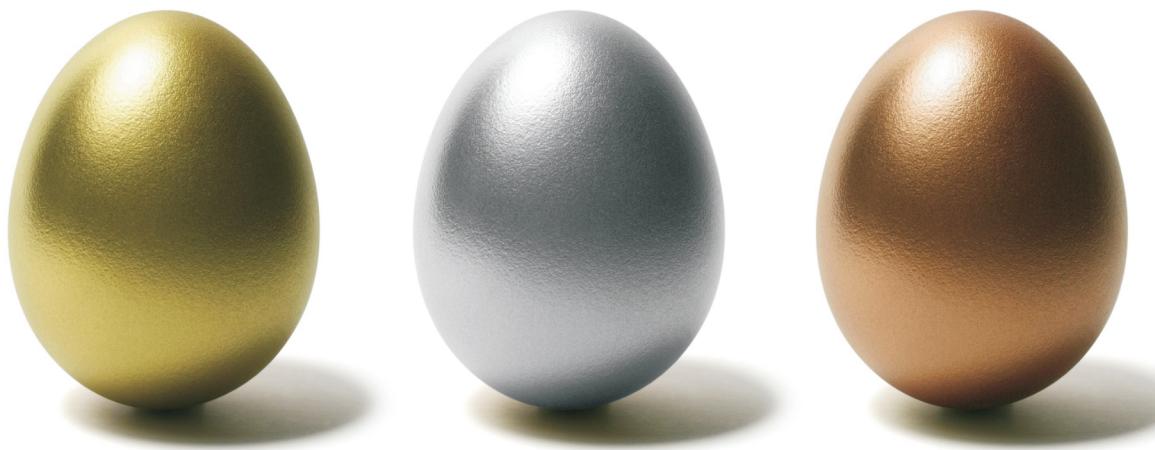


Software Engineering

Jibitesh Mishra
Ashok Mohanty



Software Engineering

Jibitesh Mishra

Associate Professor and Head
Department of Computer Science and Engineering
College of Engineering and Technology
Bhubaneswar

Ashok Mohanty

Reader
Department of Mechanical Engineering
College of Engineering and Technology
Bhubaneswar

PEARSON

Delhi • Chennai • Chandigarh

Copyright © 2012 Dorling Kindersley (India) Pvt. Ltd

Published by Pearson India Education Services Pvt. Ltd, CIN: U72200TN2005PTC057128, formerly known as TutorVista Global Pvt. Ltd, licensee of Pearson Education in South Asia.

No part of this eBook may be used or reproduced in any manner whatsoever without the publisher's prior written consent.

This eBook may or may not include all assets that were part of the print version. The publisher reserves the right to remove any material in this eBook at any time.

ISBN 978-93-325-5869-4

eISBN 978-93-325-8697-0

Head Office: A-8 (A), 7th Floor, Knowledge Boulevard, Sector 62, Noida 201 309, Uttar Pradesh, India.

Registered Office: 4th Floor, Software Block, Elnet Software City, TS-140, Block 2 & 9, Rajiv Gandhi Salai, Taramani, Chennai 600 113, Tamil Nadu, India.

Fax: 080-30461003, Phone: 080-30461060

www.pearson.co.in, Email: companysecretary.india@pearson.com

CONTENTS

| | |
|--|-----------|
| <i>Preface</i> | <i>xi</i> |
| 1. INTRODUCTION | 1 |
| 1.1 What is Software? | 2 |
| <i>1.1.1 Systems Software</i> | <i>2</i> |
| <i>1.1.2 Applications Software</i> | <i>3</i> |
| 1.2 Characteristics of Software | 3 |
| 1.3 Evolution of Software for Business | 4 |
| 1.4 Generations of Computers | 5 |
| <i>1.4.1 Generation of Computer Hardware</i> | <i>5</i> |
| <i>1.4.2 Generations of Software</i> | <i>6</i> |
| 1.5 Programming Languages | 6 |
| <i>1.5.1 Low-level Languages</i> | <i>7</i> |
| <i>1.5.2 High-level Languages</i> | <i>7</i> |
| <i>1.5.3 Fourth-generation Languages</i> | <i>8</i> |
| 1.6 Paradigm Shift in Programming Techniques | 9 |
| <i>1.6.1 Early Computer Programming</i> | <i>9</i> |
| <i>1.6.2 Control Flow-based Design</i> | <i>9</i> |
| <i>1.6.3 Structured Programming</i> | <i>10</i> |
| <i>1.6.4 Data Structure-oriented Design</i> | <i>10</i> |
| <i>1.6.5 Data Flow-oriented Design</i> | <i>10</i> |
| <i>1.6.6 Object-oriented Design (1980s)</i> | <i>10</i> |
| 1.7 Software Crisis and Emergence of Software Engineering | 11 |
| <i>1.7.1 The Software Crisis</i> | <i>11</i> |
| <i>1.7.2 Demands of Today's Business</i> | <i>12</i> |
| <i>1.7.3 Critical Problems of Software Development</i> | <i>13</i> |
| 1.8 Core Aspects of Software Engineering | 14 |
| 1.9 Salient Features of Software Development | 16 |
| <i>Summary</i> | <i>18</i> |
| <i>Exercises</i> | <i>19</i> |
| 2. SOFTWARE DEVELOPMENT PROCESS | 21 |
| 2.1 Software Processes | 21 |
| 2.2 Software Development Life Cycle Models | 22 |
| 2.3 Waterfall Model | 23 |
| 2.4 The “V” Model | 25 |
| 2.5 Prototyping Model | 26 |
| 2.6 The Iterative Waterfall Model | 28 |

| | |
|--|-----------|
| 2.7 The Spiral Model | 29 |
| 2.8 Process Standards | 30 |
| <i>Summary</i> 33 | |
| <i>Exercises</i> 33 | |
| 3. SOFTWARE REQUIREMENT ENGINEERING | 35 |
| 3.1 Requirement Engineering Process | 36 |
| 3.1.1 <i>Types of Software Requirements</i> | 36 |
| 3.2 Requirement Inception | 37 |
| 3.2.1 <i>Identification of Stakeholders</i> | 37 |
| 3.3 Requirement Elicitation | 38 |
| 3.3.1 <i>Requirement Elicitation Through Interview</i> | 40 |
| 3.3.2 <i>Requirement Elicitation Through Questionnaire</i> | 41 |
| 3.3.3 <i>Record Review</i> | 41 |
| 3.3.4 <i>Observation</i> | 41 |
| 3.3.5 <i>Collaborative Requirement Gathering</i> | 41 |
| 3.3.6 <i>Output of Requirement Elicitation</i> | 43 |
| 3.4 Requirement Elaboration | 44 |
| 3.4.1 <i>Initial User Requirements</i> | 44 |
| 3.4.2 <i>Initial Technical Requirements</i> | 44 |
| 3.4.3 <i>Final Functional Requirements</i> | 44 |
| 3.5 Negotiation | 46 |
| 3.6 Requirement Validation | 47 |
| 3.7 Structure of SRS | 47 |
| 3.8 Characteristics of the RE Process | 49 |
| <i>Case Study</i> | |
| <i>Summary</i> 52 | |
| <i>Exercises</i> 52 | |
| 4. SOFTWARE DESIGN APPROACHES | 53 |
| 4.1 Different Approaches to SAD | 54 |
| 4.2 Overview of the FO Approach | 54 |
| 4.2.1 <i>Model and Tools</i> | 55 |
| 4.2.2 <i>Salient Features of SSAD</i> | 56 |
| 4.3 Overview of the OO Approach | 58 |
| 4.3.1 <i>Object-oriented Analysis</i> | 59 |
| 4.3.2 <i>Object-oriented Design</i> | 60 |
| 4.3.3 <i>Object oriented Testing</i> | 61 |
| 4.3.4 <i>Object-oriented Maintenance</i> | 61 |
| 4.4 Comparison of OOAD with SSAD | 61 |
| <i>Summary</i> 63 | |
| <i>Exercises</i> 63 | |
| 5. STRUCTURED ANALYSIS | 65 |
| 5.1 Introduction to Structured Analysis | 66 |
| 5.2 Data Flow Diagram | 67 |
| 5.2.1 <i>Rules for Drawing DFD</i> | 71 |
| 5.2.2 <i>Physical and Logical DFD</i> | 71 |
| 5.3 Process Specification | 72 |

| | |
|--|------------|
| <i>5.3.1 Decision Tables</i> | 72 |
| <i>5.3.2 Decision Tree</i> | 76 |
| 5.4 Data Dictionary | 76 |
| <i>5.4.1 Data Dictionary Internals</i> | 77 |
| <i>5.4.2 Data Dictionary Types</i> | 79 |
| 5.5 Entity Relationship Model | 80 |
| 5.6 State Transition Diagram | 82 |
| <i>Summary</i> 86 | |
| <i>Exercises</i> 86 | |
| 6. STRUCTURED DESIGN 89 | |
| 6.1 Structured Design Methodologies | 89 |
| 6.2 Coupling and Cohesion | 90 |
| 6.3 Structure Chart | 92 |
| 6.4 Mapping DFD into a Structure Chart | 93 |
| <i>6.4.1 Refinement of DFD</i> | 93 |
| <i>6.4.2 Transaction Analysis</i> | 94 |
| <i>6.4.3 Transform Analysis</i> | 95 |
| 6.5 Data Design | 97 |
| 6.6 Detail Design | 98 |
| <i>6.6.1 Program Flowchart</i> | 98 |
| <i>6.6.2 Structured Flowchart</i> | 101 |
| <i>6.6.3 Pseudocode</i> | 101 |
| 6.7 HIPO Documentation | 104 |
| <i>Summary</i> 105 | |
| <i>Exercises</i> 106 | |
| 7. OBJECT-ORIENTED CONCEPTS AND PRINCIPLES 107 | |
| 7.1 Key Concepts | 107 |
| <i>7.1.1 Object</i> | 108 |
| <i>7.1.2 Class</i> | 108 |
| <i>7.1.3 Message</i> | 111 |
| <i>7.1.4 Inheritance</i> | 111 |
| <i>7.1.5 Abstraction</i> | 112 |
| <i>7.1.6 Encapsulation</i> | 113 |
| <i>7.1.7 Polymorphism</i> | 113 |
| 7.2 Relationships | 115 |
| <i>7.2.1 Is-A Relationship</i> | 115 |
| <i>7.2.2 Has-A Relationship</i> | 115 |
| <i>7.2.3 Uses-A Relationship</i> | 117 |
| 7.3 Some More Concepts | 117 |
| <i>7.3.1 Object Identifier</i> | 117 |
| <i>7.3.2 Object References</i> | 117 |
| <i>7.3.3 Object Persistence</i> | 118 |
| <i>7.3.4 Metaclasses</i> | 118 |
| 7.4 Modeling Techniques | 118 |
| <i>7.4.1 Booch OO Design Model</i> | 118 |
| <i>7.4.2 Rumbaugh's Object Modeling Technique</i> | 119 |
| <i>7.4.3 Jacobson's model</i> | 120 |

| | |
|---|------------|
| 7.5 The Unified Approach to Modeling | 123 |
| 7.6 Unified Modeling Language | 123 |
| <i>Summary</i> | 124 |
| <i>Exercises</i> | 125 |
| 8. OBJECT-ORIENTED ANALYSIS 127 | |
| 8.1 Use-Case Modeling | 127 |
| <i>8.1.1 Development of Use-Case</i> | 130 |
| <i>8.1.2 Use-case Realization</i> | 133 |
| 8.2 Activity Diagram and State Diagram | 134 |
| 8.3 Interaction Diagrams | 136 |
| <i>8.3.1 Sequence Diagram</i> | 136 |
| <i>8.3.2 Collaboration Diagram</i> | 136 |
| 8.4 Types of Classes | 137 |
| <i>8.4.1 Interface Class</i> | 138 |
| <i>8.4.2 Entity Class</i> | 138 |
| <i>8.4.3 Control Class</i> | 139 |
| 8.5 Class Classification Approaches | 140 |
| <i>8.5.1 Noun Phrase Approach</i> | 140 |
| <i>8.5.2 Classical Approach</i> | 142 |
| <i>8.5.3 Function Point Approach</i> | 142 |
| <i>8.5.4 Domain Analysis Approach</i> | 142 |
| <i>8.5.5 Structural Approach</i> | 143 |
| <i>8.5.6 CRC Card Approach</i> | 143 |
| <i>8.5.7 Use-case-driven Approach</i> | 144 |
| 8.6 Relationship, Attributes and Method Identification | 145 |
| <i>8.6.1 Relationships</i> | 145 |
| <i>8.6.2 Attributes</i> | 146 |
| <i>8.6.3 Methods</i> | 147 |
| <i>Case Study-I: The ATM System of Bank</i> | 148 |
| <i>Case Study-II: The Milk Dispenser</i> | 155 |
| <i>Summary</i> | 158 |
| <i>Exercises</i> | 159 |
| 9. OBJECT-ORIENTED DESIGN 161 | |
| 9.1 System Context and Architectural Design | 162 |
| <i>9.1.1 Defining System Boundary</i> | 162 |
| <i>9.1.2 Identification of Subsystems</i> | 162 |
| <i>9.1.3 Prioritisation of Non-functional Requirement</i> | 163 |
| <i>9.1.4 Design Framework</i> | 163 |
| <i>9.1.5 Design Axioms</i> | 164 |
| <i>9.1.6 Access Layer Prototypes</i> | 165 |
| 9.2 Principles of Class Design | 165 |
| 9.3 Types of Design Classes | 167 |
| 9.4 Component Diagram and Deployment Diagram | 167 |
| <i>9.4.1 Component Diagram</i> | 168 |
| <i>9.4.2 Deployment Diagram</i> | 168 |
| 9.5 Patterns | 169 |
| <i>9.5.1 Types of Patterns</i> | 170 |

| | |
|---|------------|
| 9.5.2 Pattern Template | 171 |
| 9.5.3 Generative and Non-generative Patterns | 171 |
| 9.5.4 Antipatterns | 171 |
| 9.6 Framework | 175 |
| 9.61 Struts Framework | 175 |
| 9.62 .NET Framework | 177 |
| Summary | 178 |
| Exercises | 178 |
| 10. USER INTERFACE DESIGN | 181 |
| 10.1 Types of User Interfaces | 181 |
| 10.2 Characteristics of User Interface | 182 |
| 10.3 Textual User Interface | 183 |
| 10.3.1 Command Language-based Interface | 184 |
| 10.3.2 Menu-based TUI | 185 |
| 10.4 Graphical User Interface | 185 |
| 10.4.1 Menus in Graphical Interface | 186 |
| 10.4.2 Mode-based Interface and Mode-less Interface | 187 |
| 10.4.3 Window Management System | 188 |
| 10.5 Widget-based GUI | 189 |
| 10.6 User Interface Design | 194 |
| Summary | 196 |
| Exercises | 197 |
| 11. CODING AND DOCUMENTATION | 199 |
| 11.1 Coding Standards | 200 |
| 11.1.1 Formatting Features | 200 |
| 11.1.2 Naming Convention | 202 |
| 11.1.3 Coding Norms | 202 |
| 11.2 Coding Guidelines | 203 |
| 11.3 Software Documentation | 204 |
| 11.4 Documentation Standard and Guidelines | 206 |
| 11.4.1 Structure of User Documentation | 206 |
| 11.4.2 Usage Mode | 207 |
| 11.4.3 Information Content of Software User Documentation | 207 |
| 11.4.4 Format of Software User Documentation | 209 |
| 11.4.5 Format for Instructions | 210 |
| 11.5 CASE Tools | 210 |
| 11.5.1 Upper and Lower CASE Tools | 211 |
| 11.5.2 CASE Workbenches and Environments | 211 |
| 11.5.3 Advantages of CASE Tools | 212 |
| 11.5.4 Pitfalls of CASE Tools | 214 |
| Summary | 214 |
| Exercises | 215 |
| 12. SOFTWARE TESTING | 217 |
| 12.1 Testing Fundamentals | 217 |
| 12.1.1 Verification and Validation | 218 |

| | |
|---|------------|
| 12.1.2 Testing Process | 218 |
| 12.2 Black Box Testing | 220 |
| 12.2.1 Decision Table-based Testing | 221 |
| 12.2.2 Cause–Effect Graphs in Black Box Testing | 221 |
| 12.2.3 Equivalence Partitioning and Boundary Value Analysis | 223 |
| 12.3 White Box Testing | 225 |
| 12.3.1 Statement Coverage | 226 |
| 12.3.2 Branch Coverage | 227 |
| 12.3.3 Path Coverage | 228 |
| 12.3.4 McCabe’s Cyclomatic Complexity | 228 |
| 12.3.5 Data Flow-based Testing | 228 |
| 12.3.6 Mutation Testing | 232 |
| 12.4 Unit Testing | 233 |
| 12.5 Integration Testing | 233 |
| 12.5.1 Big Bang Integration Testing | 234 |
| 12.5.2 Incremental Integration Testing | 234 |
| 12.5.3 Bottom-up Integration Testing | 234 |
| 12.5.4 Top-down Integration Testing | 235 |
| 12.5.5 Sandwiched Testing | 236 |
| 12.5.6 Backbone Integration Testing | 236 |
| 12.5.7 Thread Integration Testing | 236 |
| 12.6 Object-oriented Testing | 236 |
| 12.6.1 Issues in OO testing | 239 |
| 12.6.2 State Transition Testing | 239 |
| 12.6.3 Transaction Flow Testing / Scenario-based Testing | 241 |
| 12.7 System Testing | 242 |
| 12.8 Usability Testing | 243 |
| Summary | 244 |
| Exercises | 245 |
| 13. SOFTWARE METRICS | 247 |
| 13.1 Software Metrics and Its Classification | 247 |
| 13.2 Software Size Metrics | 248 |
| 13.2.1 LOC Metrics | 249 |
| 13.2.2 Function Point Metrics | 249 |
| 13.2.3 Feature Point Metrics | 251 |
| 13.2.4 Bang Metrics | 252 |
| 13.2.5 Halstead’s Metrics | 252 |
| 13.3 Quality Metrics | 253 |
| 13.4 Process Metrics | 255 |
| 13.4.1 Halstead’s Metrics | 256 |
| 13.4.2 Defect Estimation | 256 |
| 13.5 Design Metrics | 257 |
| 13.5.1 High-level Design Metrics | 257 |
| 13.5.2 Component-level Design Metrics | 258 |
| 13.6 Object-oriented Metrics | 259 |
| 13.6.1 CK Metrics Suite | 259 |
| 13.6.2 Metrics for Object Oriented Design | 260 |
| Summary | 262 |

Exercises 262

| | |
|---|------------|
| 14. SOFTWARE PROJECT ESTIMATION | 265 |
| 14.1 Software Project Parameters | 265 |
| 14.2 Approaches to Software Estimation | 266 |
| 14.3 Project Estimation Techniques | 267 |
| 14.3.1 <i>Estimation by Expert Judgement</i> | 267 |
| 14.3.2 <i>Estimation by Analogy</i> | 268 |
| 14.3.3 <i>Estimation by Available Resources</i> | 268 |
| 14.3.4 <i>Estimation by Software Price</i> | 268 |
| 14.3.5 <i>Estimation by Parametric Modeling</i> | 269 |
| 14.3.6 <i>Limitations of Estimation Techniques</i> | 269 |
| 14.4 Classification of Software Projects | 270 |
| 14.5 Constructive Cost Estimation Model | 271 |
| 14.5.1 <i>Basic COCOMO</i> | 271 |
| 14.5.2 <i>Intermediate COCOMO</i> | 273 |
| 14.5.3 <i>Complete COCOMO</i> | 275 |
| 14.6 COCOMO II | 276 |
| 14.6.1 <i>Early Design Model</i> | 276 |
| 14.6.2 <i>Post-architecture Model</i> | 278 |
| 14.7 Conclusion | 279 |
| Summary | 280 |
| Exercises | 281 |
| 15. SOFTWARE PROJECT MANAGEMENT | 283 |
| 15.1 Introduction to Software Project Management | 283 |
| 15.1.1 <i>Salient Features of Software Project</i> | 284 |
| 15.1.2 <i>Responsibilities of Software Project Manager</i> | 285 |
| 15.1.3 <i>Qualities of Software Project Manager</i> | 285 |
| 15.2 Project Planning | 285 |
| 15.2.1 <i>Types of Plans</i> | 286 |
| 15.2.2 <i>Software Development Plan</i> | 286 |
| 15.3 Work Breakdown Structure | 287 |
| 15.4 Project Scheduling | 290 |
| 15.4.1 <i>Activity Network</i> | 290 |
| 15.4.2 <i>Critical Path Method (CPM) and Program Evaluation Review Technique (PERT)</i> | 294 |
| 15.5 Execution, Monitoring and Control | 296 |
| 15.5.1 <i>Earned Value Monitoring</i> | 297 |
| 15.5.2 <i>Gantt Chart</i> | 297 |
| 15.5.3 <i>Causes of Project Failure</i> | 298 |
| 15.6 Risk Management | 299 |
| 15.6.1 <i>Types of Risks</i> | 300 |
| 15.6.2 <i>Risk Management Activities</i> | 301 |
| 15.6.3 <i>Risk Management Strategies</i> | 301 |
| 15.7 Configuration Management | 302 |
| 15.7.1 <i>Need for Configuration Management</i> | 303 |
| 15.7.2 <i>Configuration Management Process</i> | 303 |
| 15.7.3 <i>Software Version and Revision</i> | 305 |

| | |
|---|------------|
| <i>Summary</i> | 305 |
| <i>Exercises</i> | 306 |
| 16. SOFTWARE QUALITY MANAGEMENT | 307 |
| 16.1 The Concept of Quality | 307 |
| 16.2 Evolution of Quality Management | 308 |
| 16.3 Some Thoughts of Quality Gurus | 309 |
| 16.4 Process Quality Models | 312 |
| 16.4.1 <i>The Deming Prize</i> | 312 |
| 16.4.2 <i>Baldrige (MBNQA) Model</i> | 312 |
| 16.4.3 <i>European Foundation for Quality Management (EFQM) Business Excellence Model</i> | 314 |
| 16.5 Quality Assurance | 315 |
| 16.5.1 <i>Concepts and Definition</i> | 315 |
| 16.5.2 <i>Standards and Procedures</i> | 315 |
| 16.5.3 <i>Quality Assurance Activities</i> | 316 |
| 16.6 Process Improvement and Six Sigma | 317 |
| 16.7 Process Standard: ISO 9000 | 320 |
| 16.8 Process Standard: ISO 12207 | 321 |
| 16.8.1 <i>Primary Processes</i> | 321 |
| 16.8.2 <i>Support and Organizational Processes</i> | 323 |
| 16.9 Capability Maturity Model | 323 |
| <i>Summary</i> | 325 |
| <i>Exercises</i> | 326 |
| 17. WEB ENGINEERING | 327 |
| 17.1 General Web Characteristics | 327 |
| 17.1.1 <i>Evolution of Web Software</i> | 328 |
| 17.1.2 <i>Emergence of Web Engineering</i> | 329 |
| 17.2 Web Engineering Process | 330 |
| 17.3 Web Design Principles | 332 |
| 17.4 Web Metrics | 333 |
| 17.4.1 <i>Web Metrics Based on Response and Predictor Variable</i> | 336 |
| 17.4.2 <i>Web Metrics Based on Web Characteristics</i> | 336 |
| 17.5 Mobile Web Engineering | 337 |
| 17.6 Web Engineering Security | 338 |
| <i>Summary</i> | 339 |
| <i>Exercises</i> | 340 |
| Appendix A Objective-type Questions | 341 |
| Appendix B Frequently Asked Questions with Short Answers | 359 |
| Appendix C Software Maintenance | 373 |
| Appendix D Component-based Software Engineering | 379 |
| Index | 383 |

PREFACE

Software Engineering was developed in response to the problems of building large, custom software systems for defence, government and industrial applications. There are two approaches to the study of Software Engineering—the traditional Function Oriented (FO) approach and the contemporary Object Oriented (OO) approach. Many books on Software Engineering follow either of the two approaches, whereas students are required to know both the approaches and be clear about the similarities and differences between the two. This becomes easier when both the approaches are discussed together in a single book.

This book covers both FO and OO methodologies of SE, making it a complete book on the subject. We provide an overview of both approaches in Chapter 4 and elucidate their distinct software engineering methodologies in the subsequent chapters. Aspects that are common to both approaches are dealt with together.

The structure of this book may be divided into four modules:

Module 1: Chapters 1 to 4 cover the fundamental concepts of Software Engineering.

Module 2: Chapters 5 to 9 examine Software Engineering techniques. Techniques of the traditional FO approach are described in Chapters 5 and 6, while Chapters 7, 8 and 9 provide an insight into the OO approach.

Module 3: Chapters 10 to 13 deal with topics on User Interface, Software Coding and Documentation, Software Testing, and Software Metrics, respectively.

Module 4: Chapters 14 to 16 describe the management aspects of software engineering, namely, Software Estimation, Software Project Management and Software Quality Management, respectively.

Web Engineering, an emerging area of Software Engineering, is explored in Chapter 17. In addition, topics on Software Maintenance and Component-based Software Engineering are covered separately in the appendices.

Replete with diagrams, the book has an adequate number of questions at the end of each chapter. There are also about 140 multiple-choice objective-type questions and about 120 frequently asked questions with short answers given separately in the appendices. It expounds on the concepts of Software Engineering through case studies and examples to which the students are familiar, such as software for educational and banking systems. It emphasizes the interdisciplinary nature of Software Engineering with focus on topics such as Project Management, Estimation, Quality Management, Quality Standards and Metrics that fall under this grey area. The book is designed as a comprehensive text for undergraduate students, which gives a deep insight into software engineering methodologies.

The content of this book conforms to the syllabi of leading universities and is laid out as per the generally accepted body of knowledge on software engineering. We have tried to keep the text simple and complete and hope that the book will be useful to students.

ACKNOWLEDGEMENTS

Continuous motivation from our colleagues and staff members at the College of Engineering and Technology, Bhubaneswar, has helped us in writing this book. The knowledge we had gained from our association with Infosys, Tata Consultancy Services (TCS) and National Productivity Council (NPC) has made the book more qualitative. The feedback we had received from the staff members of Xavier's Institute of Management Bhubaneswar (XIMB) and Indian Institute of Technology Kharagpur (IITKGP) has helped us in improving the contents of this book.

Jibitesh Mishra
Ashok Mohanty

ABOUT THE AUTHORS

Jibitesh Mishra is currently Associate Professor and Head of the Department of Computer Science and Engineering, College of Engineering and Technology, Bhubaneswar. He has taught for over 16 years in various universities across the world and authored four books of repute. His first book as co-author, *Design of Information System: A Modern Approach*, was published in India, the UK and China in the year 2000. He has published more than 20 research papers in international journals and conference proceedings. He has also edited proceedings of international conferences and reviewed journal papers. His research interests lie in Fractal Graphics, Software Engineering and Web Engineering.

Ashok Mohanty is currently Reader in the Department of Mechanical Engineering, College of Engineering and Technology, Bhubaneswar. A graduate in mechanical engineering and a post-graduate in industrial management, he has co-authored the book *Design of Information System: A Modern Approach* with Dr Jibitesh Mishra. He has over 8 years of industrial experience in a public sector undertaking and has taught for about 15 years in a government engineering college. He has also taught at Xavier Institute of Management as guest faculty. He has published about 15 research papers in journals and conferences. His areas of specialization include Project Management, Quality Engineering and Management Information System.

INTRODUCTION

This chapter provides a preamble to the software engineering discipline. It briefly describes the following topics:

- *Software and their types*
- *Characteristics of commercial software*
- *Evolution of software for business*
- *Generation of computers*
- *Overview of different programming languages*
- *Developments in programming techniques*
- *Factors that led to the emergence of the software engineering discipline*
- *Core aspects of software engineering*
- *Salient features of software development*

After going through this chapter a reader will understand the scope of software engineering and its importance in software development.

Today, every sector of the economy depends on computers to a great extent. Software is a key element of any computer-based system. Industries have realized that quality software is an important means to improve performance and to gain competitive advantage. The development and supply of software is one of the fastest growing industry segments in India and abroad.

The software developed as commercial products are generally much different from computer programs written for academic or research purposes. A great deal of effort, time and money are required to design and develop any commercial software. Development of even a trivial piece of software requires many activities to be performed and it is regarded as a project. Computer programming is just one part of the software development process. Like any other economic endeavor, development of software requires a systematic approach that includes comprehensive methods, better tools for efficient execution of these methods and procedures for quality assurance, coordination and control. It also involves people and their management. All the engineering aspects relating to software development have combined

together to evolve as a discipline called software engineering. It is concerned with building quality software within time and resource constraints.

1.1 WHAT IS SOFTWARE?

A computer has two aspects: (1) hardware and (2) software. The term hardware describes the physical aspects of computers and related devices. Detailed instructions in a form understandable to computer hardware need to be given for operating a computer and getting meaningful output from it. A set of instructions is called a computer program. The programs are often stored on computer storage devices so that they can be invoked when required. Software is a general term for the various programs and data stored on computer storage devices to operate the computer and related devices. Software controls the operation of computer hardware. It has three principal functions:

- (i) It helps in operating and managing the computer resources.
- (ii) It serves as an intermediary between the organization and its stored information.
- (iii) It helps in problem solving by using computer hardware.

Without the instructions provided by software, computer hardware is unable to perform any of the tasks associated with computers.

Software can be categorized into two types:(1) systems software and (2) applications software.

Systems software consists of generalized programs that manage computer resources such as the central processing unit (CPU), printers, terminals, communication links and peripheral equipment. In other words, systems software serves as the intermediary between the software used by end users and the computer itself. Applications software consists of programs designed for applying the computer to solve a specific problem. The programs for payroll processing or sales order entry are examples of applications software. Systems software provides the platform on which applications software runs. The relationship between user, applications software, systems software and computer hardware is shown in Figure 1.1. The user sends instructions to the applications software, which “translates” the instructions for the systems software, which in turn forwards them to the hardware. Information flows in two directions: from the user working on the computer to the hardware and back again to the user.

1.1.1 Systems Software

The extensive use of systems software began with second-generation computers in the early 1960s. Before this, the operation of a computer was controlled primarily by human operators. For example, the operator had to activate each peripheral device when that device was needed by the computer. This type of human intervention wasted large amounts of computer time and human resources. These functions were automated by software called operating systems.

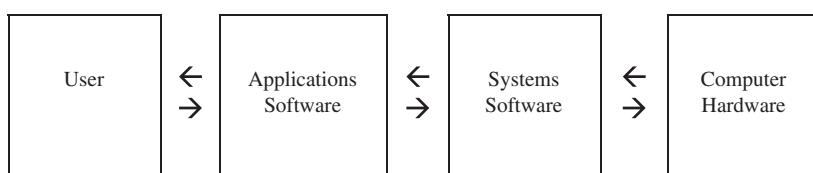


Figure 1.1 Relationship among user, software and hardware

An operating system is a set of integrated programs that controls the execution of computer programs and manages the storage and processing resources of a computer system. These programs are stored partly in primary storage and partly in direct-access secondary storage. With operating systems, a queue of jobs that are awaiting execution is fed onto a disk. The operating system starts each job as and when the system resources are available for its execution. Since human intervention is eliminated, computer idle time is significantly reduced.

Systems software consists of three types of programs.

- (i) System control program: It controls the execution of programs, manages the storage and processing resources of the computer and performs other management and monitoring functions. The set of these programs constitutes the operating system.
- (ii) System support program: It provides services to other computer programs and to computer users for carrying out routine functions such as copying, deleting, merging or sorting files, directory checking etc. Most of these programs are also included in the operating system.
- (iii) System development program: It assists in the creation of application programs. Examples are language translators such as a BASIC interpreter and compilers such as FORTRAN, C/C++, Java etc.

1.1.2 Applications Software

It applies the power of the computer to solve problems by performing specific tasks. It can support individuals, groups and organizations. Companies can customize applications software, buy existing programs off-the-shelf or use a combination of customised and off-the-shelf applications software.

Electronic spreadsheet, word processing, graphics software and integrated software are general-purpose applications software. Computer programs for payroll processing, inventory management and sales analysis are specific-purpose applications software. These are customized to meet specific requirements of user organizations.

1.2 CHARACTERISTICS OF SOFTWARE

Software is often the single largest cost item in a computer-based application. Though software is a product, it is different from other physical products.

- (i) Software costs are concentrated in engineering (analysis and design) and not in production.
- (ii) Cost of software is not dependent on volume of production.
- (iii) Software does not wear out (in the physical sense).
- (iv) Software has no replacement (spare) parts.
- (v) Software maintenance is a difficult problem and is very different from hardware (physical product) maintenance.
- (vi) Most software are custom-built.
- (vii) Many legal issues are involved (e.g. intellectual property rights, liability).

As stated earlier, commercial software are generally different from computer programs written for academic or research purposes. The characteristics of real-world software are given in Box 1.1.

Development of software is also different in some way from building a physical object such as a bridge, house or factory. Problems solved with software solutions are complex. Finding solutions requires some ingenuity and good planning. The progress made in a software project is not visible to the

Box 1.1: Characteristics of real-world software

- It is generally developed by software firms for their clients under formal business contracts.
- Like any product, software is designed based on some software specification.
- It is usually developed in teams and not by individuals.
- It generally includes clear and detailed documentation (i.e. design manual and users' manual).
- It is meant for users who need not have good knowledge of computers.
- It generally has user-friendly interfaces so that users having limited expertise in computers can operate the system.
- Generally, software is designed to be run on different platforms.
- It has a lifetime in years, after which it becomes obsolete. Hence, software is designed keeping its intended life and cost in view.
- It generally requires some modification from time to time to accommodate changes taking place in the organization and the environment.
- It is developed under formalized product reviews (quality assurance) and formalized testing procedures.
- The cost of software failure may amount to an economic catastrophe. Hence, software is designed for utmost reliability.
- A computer system is prone to misuse or sabotage by persons having ulterior motives from within or from outside the organization. Hence, software is designed to be tamper-proof and protected from misuse or damage.
- Ethical issues (like protecting privacy) are also taken into consideration in designing software.

eyes as in the case of other projects. Due to the uncertainty and uniqueness of each project, it is difficult to estimate project duration and cost accurately. During the course of a project, there can be changes in the requirements and environment. Since software is not a physical product and its development is not visible to the eyes, customers often feel no compunction to put pressure for incorporating some changes at the last minute. There is no universal method for software development that can be followed in all situations. The method used in a well-managed software project may not be suitable for a similar project when the project size is different. However, keeping in view the above characteristics of software, some systematic approaches, engineering principles, tools and techniques have been devised to produce quality software within time and resource constraints.

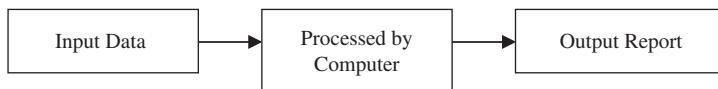
1.3 EVOLUTION OF SOFTWARE FOR BUSINESS

Computers can do repetitive calculations at a very fast rate. Only, steps for calculations need to be specified through a program. Hence in the earlier days, computers were used only for doing various repetitive operations (calculations) on input data to get output reports.

Examples:

- (i) payroll applications for calculation of gross salary, deductions, net salary etc. of employees from their basic salary.
- (ii) calculation of incentive to be paid to workers based on wage rate, hours worked, overtime worked, quantity produced etc.
- (iii) calculation of interest on bonds, amount of dividend on shares etc.

All the above examples require a large number of repetitive calculations. These calculations, which used to take days, could be done in hours with the help of computers. Hence, computers were widely used for processing of data, which is popularly known as “electronic data processing (EDP)”.



While technological advancement led to the development of high-capacity secondary memory devices at cheaper cost, computers were also used for storing data. Storing data on a magnetic medium of computer requires much less space. It is also cheaper as compared to storing data on paper. Data stored on magnetic medium facilitates easy retrieval and further data processing as and when required.

This led to the development of systems for providing information support for various managerial activities and functions. The system consisted of a high-capacity computer with a large number of peripherals such as terminals, printers and magnetic storage devices that stored large volume of data. Software was developed to operate such large systems to manage computer resources, to update data in real time and to retrieve information. Thus, the complexity of software began to increase.

A number of software packages called Data Base Management Systems (DBMS) were developed to solve the problems of managing large volume of data. INGRE, ORACLE, SYBASE, DB2 etc. are examples of some popular data management software. Simple PC-based DBMS packages like Dbase, FoxBASE, FoxPro were also developed to meet the requirements of small organizations. These DBMS packages greatly simplified the task of processing queries, generating reports and application programming.

As the computing power of a computer increased, it permitted the development of more sophisticated software that used various decision-making models, operation research techniques, neural network, fuzzy logic and various mathematical and statistical tools.

Today, computers are widely used in government administration, business management and in a wide range of service sectors such as banking, transport, education, entertainment, healthcare etc. This has been made possible due to rapid developments in the field of Information Technology.

1.4 GENERATIONS OF COMPUTERS

Developments in the field of computer technology have taken place through some distinct stages called generations of computers.

1.4.1 Generation of Computer Hardware

Since the times computers were first invented around 1950, there has been spectacular growth in processing speed, memory size and reliability of computers. At the same time, there has been a steady decrease in their cost and size.

The early computers, referred to as first-generation computers, were built on vacuum tubes. These computers were big machines that consumed a lot of electric power. They also required large-capacity air-conditioners to dispel considerable amount of heat generated by the vacuum tubes. Second-generation machines were built by using transistors. The use of transistors reduced the size of computers. The third generation made use of integrated circuits (ICs). Present-day computers are fourth-generation computers. They use very large scale integration (VLSI) technology so that the whole computer circuit is contained in a chip. As a result, they are not only cheaper and smaller in size but are also much more

powerful than earlier computers. Research is going on to further improve the architecture of computers. Currently, research efforts are aimed at finding a better non-silicon material as the replacement for silicon, which forms the base of electronics components.

1.4.2 Generations of Software

The sophistication and range of problems that can be addressed by programming languages can be attributed to the increased capacity of computer hardware. Like hardware, computer software also evolved over time in stages, which is referred to as generations of software.

The machine language was the first-generation computer software. It consisted of strings of binary digits 0 and 1. It was the only way to communicate with the primitive computers of the 1940s. It took highly trained, specialized programmers to understand, think and work directly with the machine language of a particular computer. Machine language instructions must specify the storage location for every instruction and data item used. Consequently, writing software in this language was extremely slow and labor-intensive; very few problems could be addressed this way.

The second-generation software, which began in the early 1950s, consisted of assembly language. Assembly language is considered a symbolic language because it consists of language-like acronyms and words such as add, sub (subtract) and load. The programs that translate assembly language into machine code are called assemblers. Today, assembly language has limited use in software development.

Third-generation computer software, which prevailed from the early 1960s to the late 1970s, are also called high-level languages. These are less machine-dependent and can be used on different types of computers. Popular third-generation languages include FORTRAN (FORmula TRANslator) for scientific and mathematical problems, COBOL (COmmon Business-Oriented Language) for business problems requiring extensive manipulation of data files and BASIC (Beginner's All-purpose Symbolic Instruction Code), a generalized programming language popular for microcomputers. The language "C" was developed in the early 1970s. It has features to manipulate memory storage locations of a computer. It is a structured language and is very versatile so as to handle a wide range of problems. Third-generation software is still in wide use today.

Fourth-generation software was developed in the early 1980s and is used widely along with third-generation software for application development. Fourth-generation software consists of query software, report generators, graphics software, application generators and other tools that substantially reduce the effort and time required in software development.

The first three generations of software languages were procedural. Program instructions had to detail a sequence of steps, or procedures, telling the computer what to do and how to do it. In contrast, fourth-generation software has nonprocedural features. Program instructions need only specify what has to be accomplished rather than provide details about how to carry out the task. Consequently, the same process can be accomplished with fewer program steps and lines of program code as compared to third-generation languages. Thus, fourth-generation software makes the task of programming much easier.

1.5 PROGRAMMING LANGUAGES

Each of the major kinds of software consists of programs written in specific programming languages. Each programming language was designed to solve a particular class of problems. It is important to understand the strengths and limitations of each of these languages in order to select appropriate software.

1.5.1 Low-level Languages

Assembly language was developed to overcome some of the difficulties of machine language. Mnemonic (easy to remember) codes and symbols are used to represent operations (such as adding or moving) and storage locations. For example, “A” stands for “Add” and “L” stands for “Load”. Since assembly-language instructions correspond closely to machine-language instructions, it is machine oriented for a specific computer and specific microprocessor. Each assembly-language instruction corresponds to a single machine-language instruction. This is illustrated through an example of Machine Language Code and Assembly Language Code given below.

Purpose of code: Add the Value of B to A

Machine Code: 111110100101001010010000000000001001000000001100

Assembly Code: AP TOTAL A, VALUE B

However, it was necessary that instructions in both machine language and assembly language specified the storage location for every instruction and data item used. Hence, these were called low-level languages. These languages emphasize efficient use of computer resources, require minimal memory and require minimum CPU activity for processing. Thus, these are used for writing basic instructions that are required for starting the computer.

1.5.2 High-level Languages

Assembly language is easier to use than pure machine language. It is used primarily for writing operating systems software. However, it is still extremely difficult to learn and requires highly skilled programmers. Thus, it is not suitable for writing application programs. The next choice is the high-level languages that are more powerful, easier to use and directed toward specialized classes of problems. A list of some high-level languages and their features is given in Box 1.2.

Box 1.2: List of some high-level languages and their features

FORTRAN: It was developed in 1954 to facilitate the writing of scientific, mathematical and engineering software. Its great strength lies in its facilities for mathematical computations. It does not have strong facilities for input/output activities or for working with lists. Thus, it would not be appropriate for business problems that involve reading massive amounts of records and producing reports. On the other hand, for business problems requiring sophisticated computations, such as forecasting and modeling, FORTRAN has been used successfully.

COBOL: It was introduced in the early 1960s and is still being used for business applications. It was designed to process large data files with alphanumeric characters (mixed alphabetic and numeric data), which are characteristic of business problems. It can read, write and manipulate records very effectively. Business specialists also find it easier to learn than most other programming languages. It uses relatively English-like statements, is easily readable and supports well-structured programs. It does not handle complex mathematical calculations well, however, and its programs tend to be wordy and lengthy.

PL/1: It was created by IBM in 1964 as a general-purpose programming language to support both business and scientific problem solving. It is very powerful but not widely used. Computer programmers who were accustomed to COBOL and FORTRAN and companies that had already invested heavily in those software did not want to convert to another language.

(Continued)

BASIC: It was developed in 1964 to teach Dartmouth College students how to use computers. It has become an extremely popular programming language for microcomputers and for teaching programming in schools. It is easy to learn and has minimal memory requirements for conversion into machine code.

PASCAL: Named after Blaise Pascal, the seventeenth-century mathematician and philosopher, it was developed by the Swiss computer science professor Niklaus Wirth in the late 1960s. Wirth wanted to create a language that would teach students structured programming techniques. PASCAL programs consist of smaller subprograms, each of which is a structured program in itself. PASCAL has limited features for input and output; hence, it is not well suited for most business problems.

ADA: It was developed in 1980 to provide the U.S. Defense Department with a structured programming language to be the standard for all its applications. It is also useful for business problems and is used in some non-military applications. It can operate on microcomputers, is portable across different brands of computer hardware and promotes structured program design. It also supports concurrent tasks and real-time programming. It was named after Ada, Countess of Lovelace and an able nineteenth-century mathematician, who developed the mathematical tables for an early calculating machine.

C: The language C was developed under the auspices of AT&T's Bell Laboratories in the early 1970s. It is the language in which most of the UNIX operating system is written. It has much of the tight control and efficiency of execution of assembly language, yet is easier to learn and is portable across different microprocessors. Much commercial microcomputer software has been written in C, and it is starting to be used for business, scientific and technical applications on larger computers.

C++: It is an object-oriented language (OOP). It is distinctly different from the other languages mentioned above. Basically, the approach is to use objects that are self-contained units that contain both data and related facts and functions. Functions are the instructions to act on the data. An object encapsulates both data and their related instructions and a specific occurrence of an object is called an instance. At this point, students are not expected to understand exactly how OOP works.

JAVA: It is another OOP language widely used on the Internet today. It is developed both as a compiler and a language. It is an interpreter language. There is a school of people who are sure that JAVA is going to be the language of choice and the main language for computers.

1.5.3 Fourth-generation Languages

Fourth-generation languages offer two major advantages:

- (i) They allow end users to develop software on their own with little or no technical assistance.
- (ii) They offer substantial productivity gains in software development.

Fourth-generation languages tend to be less procedural than high-level languages, making them more suitable for end users. Thus, these languages have created the technical platform for users to play a larger role in solving problems with little assistance from computer specialists. Fourth-generation languages are easier to learn and apply. These languages are not rigid on syntax. These have features by which a major portion of computer code can be automatically generated. Programs developed through these languages generally have fewer errors. These languages also have tools to detect and repair errors. Hence, fourth-generation languages improve the productivity of computer programmers. Studies have shown that productivity gain is as much as 3 to 5 times over conventional languages.

1.6 PARADIGM SHIFT IN PROGRAMMING TECHNIQUES

Starting from early years of computers to present times, there have been significant changes in characteristics of problems, issues and computing practices.

1.6.1 Early Computer Programming

During the 1950s, programs were written in assembly language. Programs were limited to about a few hundreds of lines of assembly code. Every programmer developed his own style of writing programs according to his intuition (exploratory programming).

Introduction of high-level languages such as FORTRAN, ALGOL and COBOL in the early 1960s greatly reduced software development efforts. However, software development style was still exploratory. Typical program sizes were limited to a few thousands of lines of source code.

1.6.2 Control Flow-based Design

As the size and complexity of programs increased further, exploratory programming style proved to be insufficient. Programmers found it very difficult to write these big programs correctly. Programs written by others were also very difficult to understand and maintain. To cope up with this problem, certain norms evolved for good programming practice. Particularly, much attention was given to the design of the program's control structure.

A program's control structure indicates the sequence in which the program's instructions are executed. Flow-charting technique was developed to help programmers to design a good control structure of programs. Flow-charting technique was also useful to make others understand a program by mentally simulating the program's execution sequence. A program having an untidy flowchart representation is difficult to understand and debug.

It was found that:

- GO TO statements make the control structure of a program untidy.
- GO TO statements alter the flow of control arbitrarily.

In March 1968, Dijkstra wrote a letter to editor "GO TO Statement Considered Harmful" in Communications of the Association for Computing Machinery (ACM) and a subsequent article "The structure of the THE-multiprogramming system". Many programmers were unhappy to read his article. It was difficult for many programmers to change their programming style. Many of them were so used to the "GO TO" statement that they considered the use of these statements inevitable. They published several counter articles highlighting the advantages and inevitability of these statements. However, the need to restrict the use of these statements was universally recognized. It was conclusively proved that only three programming constructs are sufficient to express any programming logic:

1. Sequence: This consists of assignment statements.

e.g. $a = 2; b = 5; c = a + b$

2. Selection: This consists of testing of a condition followed by assignment statements to be executed based on whether the condition is true or false.

e.g. if ($c = 7$) $i = 0$ else $i = 1$

3. Iteration: This consists of testing of a condition followed by set of assignment statements to be executed repeatedly or not based on whether condition is true or false.

e.g. while ($i > 0$) $i = j - i$

It was accepted that any programming problem can be solved without using “GO TO” statements. This formed the basis of structured programming.

1.6.3 Structured Programming

The concept of structured programming evolved in the 1970s. A program is called structured when it uses only the following types of constructs:

- (1) Sequence (2) Selection (3) Iteration

In structured programs unstructured control flows are avoided. The program consists of a neat set of modules. It uses single-entry, single-exit program constructs. However, violations to this feature are permitted due to practical considerations such as premature loop exit to support exception handling.

Structured programs:

- are easier to read and understand,
- are easier to maintain and
- require less effort and time for development.

Research experience shows that programmers commit fewer errors while using structured “if-then-else” and “do-while” statements as compared to “test-and-branch” constructs.

1.6.4 Data Structure-oriented Design

Later, during the middle of 1970s, it was discovered that it is important to pay more attention to the design of data structures of a program than to the design of its control structure.

Techniques that emphasize on designing the data structure and on deriving program structure from it are called data structure-oriented design techniques. Michael Jackson, in the 1970s, developed a data structure-oriented design technique called Jackson’s Structured Programming (JSP) methodology. Here, the program code structure corresponds to the data structure. Hence, in this methodology, a program’s data structures are first designed using notations for sequence, selection and iteration. Then data structure design is used to derive the program structure.

1.6.5 Data Flow-oriented Design

Data flow-oriented techniques evolved during the early 1980s. They advocate that the data items input to a system must first be identified and then the processing required on the data items to produce the required outputs is determined.

Data flow technique identifies different processing stations (functions) in a system and the items (data) that flow between processing stations. It is a generic technique. It can be used to model the working of any system and not just software systems. Its major advantage is its simplicity.

1.6.6 Object-oriented Design (1980s)

Object-oriented technique is an intuitively appealing design approach. In this approach, natural objects (such as employees, pay-roll-register etc.) occurring in a problem are first identified. Then, the relationships

among objects such as composition, reference and inheritance are determined. It simplifies programs by placing logically related parts (objects and their elements) close together and by having similar parts of the program behave in a similar manner. Its salient features are:

- Simplicity
- Reuse possibilities
- Lower development time and cost
- More robust code
- Easy maintenance

Because of this, object-oriented techniques have gained wide acceptance as the standard techniques for developing software.

In the modern approach to software development, programs are written by focusing on the interfaces first and adding details later. An interface is designed using standard Graphical User Interface (GUI) components. The programs are written as responders to external and internal stimuli. These stimuli change the state (properties) of objects and their components in a defined way. The change in state is called an event. Thus, object-oriented (O-O) software generates and handles events. The programs are more understandable due to common frameworks and conventions followed by all programmers.

Recently, a large number of techniques and buzzwords have evolved in the field of software. Some of these are: distributed programming, database programming, client/server programming, intelligent programming, multi-threaded programming, safe programming, security programming, robust programming, functional programming, ego-less programming, extreme programming etc. All of these, in some way or the other, help in making better software in terms of meeting the present and anticipated future requirements of users, greater reliability and lesser cost.

1.7 SOFTWARE CRISIS AND EMERGENCE OF SOFTWARE ENGINEERING

As explained earlier, software that are developed as commercial products are generally much different from computer programs written for academic or research purposes. There is a requirement of great amounts of effort, time and money to design and develop any commercial software. It also involves people and their management. Computer programming is just a part of the process for making any commercial software. Even the most trivial pieces of software require many activities to be performed and can be considered as a project. The task of programming has become easier due to developments made in programming languages and programming techniques. Significant developments have also taken place in the way software are designed and developed. This is due to three main factors, which are:

- (i) The software crisis
- (ii) The demands of business
- (iii) Critical problems of software development

1.7.1 The Software Crisis

In the early years of computer applications, the focus of development and innovation was on hardware. Software was largely viewed as an afterthought. Computer programming was considered as a state of art known to a gifted few. Programmers followed their own personal style and did not follow any standardized approach. In this approach, design was implicitly performed in one's mind, and documentation was often non-existent.

This way of doing things was adequate for developing simple programs. However, as rapid developments took place in the field of information technology, computer programs became too complex and sophisticated to be manageable by this approach. Software soon took over more and more functions, which were hitherto done manually. As “software houses” emerged, software began to be developed for widespread distribution. A typical software project produced about 10,000 program statements. With no tools or methods of managing this increasing complexity, the number of errors began to accelerate.

For example, in a study by the Comptroller General of the United States (1979) it was found that

- 2% of software worked on delivery
- 3% worked only after some corrections
- 45% were delivered, but never successfully used
- 20% were used, only after major modification/rework
- 30% were paid for, but never completed/delivered

Due to the absence of any standardized way of programming, it was extremely difficult to find out and correct these errors. The cost of developing the software often exceeded the estimated budget. Software projects were taking a lot longer than initially envisaged. In a survey done by IBM in 1994, it was reported that

- 55% of systems cost exceeded the initial estimated budget
- 68% of software overran delivery schedules
- 88% of the software had to be substantially modified

It was observed that software projects mostly failed to meet cost and time requirements. It was also observed that the failures were caused more due to mundane human factors than due to lack of technical expertise. Many of the problems with software development could be traced to faulty methodology. These problems of software development were extensively felt in the beginning of the 1970s and are collectively referred to as the “software crisis”.

1.7.2 Demands of Today's Business

The basic paradigm governing the application of computers has always been to use general-purpose computer hardware and customized software for application-specific needs. Thus, the quality of software that drives computer-based solutions differentiates a company from its competitors. Hence, for computerization of their systems, organizations often spend much more on software than they do on hardware. The development and supply of software has thus been recognized as an industry. Presently, it is one of the fastest growing industry segments in India and abroad.

Fast developments that have taken place in computer hardware have also increased the scope for development of software. The present day computer hardware has more memory capacity. It is also much cheaper and faster as compared to earlier computers. Hence, due to the development of these powerful computer hardware, it is now possible to develop software that can use this storage and processing power of computer hardware for solving any kind of complex problem.

Due to increasing competitiveness, today's business environment has become very dynamic. The average life cycle of products has reduced significantly. Organizations need to respond very fast to changing needs of the market. To cope up with this environment, organizations make frequent changes in their business strategies, policies and procedures. The changes require frequent updates to existing software. However, poor design and haphazard practices followed in software development create problems in the

maintenance of existing software. The conditions demand that software be developed quickly and meet the tight delivery schedule required by customers.

1.7.3 Critical Problems of Software Development

Absence of a structured and methodical approach caused problems like:

- Errors made in one phase of development carried over to subsequent phases, causing rework and therefore delay.
- Difficulty in estimating effort and measuring progress caused schedule slippage and budget overruns.
- Assuring quality of delivered software in terms of reliability and meeting the customers' requirements became difficult.
- There was difficulty in incorporating design modifications and installing correct versions of software.

These prompted corporate managers, software professionals and researchers to analyze the causes of these problems. They felt the need to adopt a systematic approach to software development. Application of principles of engineering for systematic development of software gave rise to a new discipline called "Software Engineering".

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; i.e., the application of engineering to software (IEEE Standard Glossary of Software Engineering Terminology).

Software engineering is concerned with building good and reliable software. Any branch of engineering uses the following for designing and building a product:

- Methods and techniques
- Tools
- Procedures and standards

Just as in any other branch of engineering, the basic principle of software engineering is to use structured, formal and disciplined methods for building software. Software engineering provides a set of methods, tools and procedures for development of quality software within the constraints of cost and time.

Methods provide the rules, steps and techniques for carrying out software engineering tasks, such as project planning and estimation, system and software requirement analysis, design of data structure, program architecture and algorithm procedure, coding, testing and maintenance. There are mainly two approaches to software engineering, namely, (i) *function-oriented approach* and (ii) *object-oriented approach*.

Function-oriented approach is the traditional approach of software development. In this approach, a system is viewed to perform some set of functions. Hence, first, the basic functions performed by a system are identified. Based on these basic functions, a system is broken down into subsystems. For each basic function, the corresponding lower-level functions are determined and accordingly subsystems are broken down into further subsystems. Finally, each lower-level function is studied in detail to determine how it is performed and the corresponding software component is designed accordingly. Object-oriented approach is a modern approach. It supplements function-oriented method by trying to identify basic objects, their properties and behaviors.

Tools provide support for methods. They consist of various graphical representations, tables and charts that help in analysis of software requirements and its design. Data Flow Diagram (DFD), Data Dictionary, Entity-Relation (ER) diagram, Decision Table, Structured Charts etc. are some of the important tools used in the function-oriented approach. In the object-oriented approach various tools are integrated into a single unified language called “Unified Modeling Language” (UML). This language is now widely used by professionals for software development.

The tools are now integrated into system development software to automate software engineering methods. These software tools are called Computer-Aided Software Engineering (CASE) tools.

Procedures bind methods and tools into a framework. They specify the software development plan such as:

- the methods and the sequence in which methods will be applied
- the deliverables (documents, forms, reports etc.) that are required
- the controls that ensure quality and proper coordination
- the milestones that help a manager to assess progress

Software engineering provides certain paradigms that encompass the above methods, tools and procedures. These paradigms may be viewed as models of software development. Also, certain standard practices and principles for software development have evolved through consensus to be adopted by all software professionals.

Software engineering is the establishment and use of sound engineering principles, management practices, methods, tools and procedures to produce high-quality software within time and resource limitations. Thus, it is not just concerned with technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production. The developments that took place in these areas revolutionized the way software are designed and developed. Some of these developments are given below.

- Life cycle models
- Specification techniques
- Project management techniques
- Debugging techniques
- Testing techniques
- Software measurement techniques
- Quality assurance techniques
- CASE tools etc.

Software engineering has introduced a rigor and structure into the software development process—one that integrates comprehensive methods for all stages in software development, better tools for automating these methods, more powerful techniques for software quality assurance and an overall philosophy for co-ordination, control and management. These have been explained in some detail in subsequent chapters.

1.8 CORE ASPECTS OF SOFTWARE ENGINEERING

The subject “Software Engineering” is inter-disciplinary in nature. It has emerged as a discipline very recently. Due to rapid growth of knowledge in this field, software professionals and academicians felt

the need to have a consistent view of software engineering worldwide. To achieve this objective the IEEE Computer Society's Professional Practices Committee has published a guide to Software Engineering Body of Knowledge (SWEBOK) in 2004. The guide is based on a generally accepted portion of the Body of Knowledge. The material that is recognized as being within this discipline is organized into ten knowledge areas as follows:

1. Software requirements
2. Software design
3. Software construction
4. Software testing
5. Software maintenance
6. Software configuration management
7. Software engineering management
8. Software engineering process
9. Software engineering tools and methods
10. Software quality

The guide lists eight subject areas from which the software engineering knowledge has been derived. These are:

1. Computer engineering
2. Computer science
3. Management
4. Mathematics
5. Project management
6. Quality management
7. Software ergonomics
8. Systems engineering

Software engineering is concerned with both technical as well as managerial aspects of software development. There are four important core aspects of software development. These four aspects are: (1) Product, (2) Process, (3) People and (4) Project.

Product: Software, as a product, has to perform certain specific functions required by users (customers). Determination of correct functional requirements and features of software to be produced is a very critical activity of software development. For this, various stakeholders and users of software are identified to elicit information for determining functional specification of software. Sometimes requirements of one class of users may conflict with those of another. Finalization of functional specification is often a balancing act of satisfying requirements of different stakeholders within cost and time constraints.

Process: It refers to methodologies to be followed for developing the software. It is the framework for establishment of a comprehensive plan and strategies for software development. The process specifies the policies, procedures, tools and techniques to be used for software development. A number of models are available. CMM (Capability Maturity Model) is a widely used standard model for software development process.

People: Software development requires creativity and knowledge work. It is often difficult to specify the quantitative and qualitative measures of this work. Since fast technological developments are taking

place in the field of computer science, updating of knowledge is a part of computer professionals' job. A number of people having diverse expertise are required to work together for developing any software. Since work of one individual affects the work of others, quality software is mostly developed through teamwork. Hence, developing motivation, morale and teamwork among people and upgrading their professional expertise are important aspects of software engineering.

Project: As stated earlier, there is a requirement for great amounts of effort, time and money to design and develop any commercial software. A number of interrelated activities have to be performed in a planned schedule for completing the software within time constraints. Hence, development of any software can be considered as a project. Thus, project management aspects such as planning and monitoring of activities, schedule, resources and expenditure are important for software development.

1.9 SALIENT FEATURES OF SOFTWARE DEVELOPMENT

A typical software product may consist of a million rows of code. Due to its large size and complexity, it is difficult to avoid, detect and repair errors in software. A general statement of objectives is not sufficient to begin writing programs. Poor definition and documentation is one of the major causes of failed software efforts. A formal and detailed description can be determined only after thorough communication between customer and developer. Maintaining the schedule for software completion is also a critical success factor. It has been observed that unlike other projects, adding (engaging) more people does not help to crash duration of software projects. Hence, if a software project gets behind schedule, it is very difficult to catch up. Constructing, testing and demonstrating the software after removing all errors is only half the job. Usually, many new problems surface during operations, which take much time and effort to repair. Some real operational and behavioral problems are also encountered during implementation. There are several factors that impact quality and productivity significantly. Some of these factors are given below.

A) Factors relating to project characteristics

Each software project is different from others. Hence, development of software is a creative activity. Due to the novelty of each software project, some amount of uncertainty exists about its execution.

Complexity: Owing to different levels of complexity, it is difficult to fix a standard of performance. For example, it is common for programmers to write 25–100 lines of code per day for application programs. However, for utility programs and system programs this may be less than 10 lines per day. Development and maintenance efforts are nonlinear functions of product size and complexity.

Sequential nature of work: Software development involves a large number of activities that need to be performed by different individuals. For example, the blueprint of design cannot be done until the requirements of the customer are understood and finalized. A program cannot be tested until its coding is complete.

Accommodate for hardware deficiencies: Hardware and software together constitute a total system. There may be instances where hardware is not up-to-date or it may have some deficiencies. However, instead of replacing the hardware, sometimes compromises are made in the software design to accommodate for hardware deficiencies. This affects the quality of software and productivity of software development.

Performance requirements: Software can be developed to meet different levels of reliability, security and response time. Productivity of software development varies greatly based on the level of performance required by the customer.

Available time: Productivity of a software project is sensitive to time available for development. It has been found that engaging more people does not result in proportional decrease in developmental time. Productivity also decreases when a project is extended beyond its nominal duration.

B) Factors relating to development methods

Use of systematic methods, tools and procedures improves quality and productivity of software development.

Level of technology: System environment, programming practices and tools greatly facilitate software development. Many of the tasks related to software development can be automated by using these development tools. These tools also help in generating computer codes (programs), designing interfaces, designing data files, detecting and rectifying errors etc. Thus, the level of technology used for software development has a significant impact on quality and productivity.

Problem understanding: Software is developed to solve certain problems. Hence, as a first step, it is important to understand the problems that are to be solved, the constraints under which the software has to function and the overall goal that the software has to achieve. Poor initial understanding of customers' requirement causes problems in later stages of software development. It is one of the major reasons of software failure.

Appropriate notations: Notations such as symbols, graphical charts, diagrams etc. are frequently used by software professionals for communicating knowledge/ideas. These play a very important role in understanding the functional requirements of software. Many of these notations have been standardized. Use of informal/nonstandardized notations causes misunderstanding and misrepresentation of ideas. Informal notations are also not supported by software development tools (CASE tools).

Change control: The requirements for a project do change. Therefore, notations and procedures are necessary to assess the impact of proposed changes. Software consists of a number of modules, programs, data files and interfaces. Hence, any change made in one element affects other components of software. Thus, there is need for plan for change, formalized mechanisms for making changes and mechanisms for change control.

C) Human factors

Software development is a knowledge work. Hence, quality of the software product and productivity of its development is dependent to a large extent on competence and motivation of people. The characteristics of people engaged in software development have a significant effect on productivity.

Technical competence: In software development, there is no substitute for technical competence. For example, for a certain task that requires a great amount of programming skill, a skilled programmer cannot be substituted by engaging a number of less-skilled programmers. In addition to computer skill, familiarity with the work for which the software is being developed is also important for a software professional.

Ability to work in a team: Software development is a team work. It is estimated that a team member of a typical software project spends only about 30% of his time working alone. The output he produces must be compatible and integrate with the work done by other members. Hence, co-ordination and interaction among team members is very important. Thus, it is common for software professionals to spend about 50% of their time in interacting with other team members.

Communication: Poor communication is a source for misunderstandings. Hence, organizational climate free from communication barrier is conducive to software development. Communication is necessary to exchange ideas for working in a team. Communication is also necessary for eliciting information about users' problems and their requirements. Software professionals need good communication skills to present the features of their software to customers.

Motivation: Due to the nature of software projects, the responsibilities and tasks assigned to people cannot be defined and structured, as in routine jobs in manufacturing industries. Hence, it is difficult to determine performance standard relating to quantity and quality of output. Thus, enforcing external control is not as effective as self-control and motivation.

Understanding the nature of software projects and problems associated with them is essential for managing quality and productivity. Software quality is never achieved by accident. Advanced computers and software tools alone cannot ensure quality of software. It is a result of thorough planning and concerted effort. In accordance with modern concepts of quality management, all the ingredients such as people, infrastructure facilities, processes, policies and procedures contribute to quality of any software. Knowledge of software engineering equips a manager to identify the metrics to evaluate different aspects of software development, apply effective methods of control and adopt effective tools to solve problems relating to software projects.

SUMMARY

Software is a general term for the various programs and data stored on computer storage devices to operate computer and related devices. Software can be categorized into two types: (1) systems software and (2) applications software. Systems software consists of programs that manage computer resources. Applications software consists of programs designed to solve a specific problem.

Computers were first used in business for processing of data called EDP. Development of DBMS software helped in the use of computers for data storage. The increase in the computing power of computers permitted the development of more sophisticated software. Today, they are widely used in almost every field.

The first generation of computers were built on vacuum tubes. Second-generation machines were built by using transistors. The third generation made use of ICs. The present day's fourth generation of computers use VLSI circuits.

The machine language was the first-generation computer software. The second-generation software consisted of assembly language. High-level languages are the third-generation software. Fourth-generation software developed in the early 1980s. Presently, it is used along with the third-generation software for application development.

Programming techniques were developed to cope up with the size and complexity of programs. The concept of structured programming evolved in the 1970s. Data flow-oriented techniques evolved during the early 1980s. Object-oriented programming is the latest programming technique.

In the beginning of the 1970s, it was observed that large numbers of software projects were failing mostly due to human factors. This problem was referred to as the "software crisis".

The need was felt to adopt a systematic approach to software development. Application of principles of engineering for systematic development of software gave rise to a new discipline called "*Software Engineering*". Software engineering provides a set of methods, tools and procedures for development of quality software.

There are mainly two approaches to software engineering, namely, (i) function-oriented (FO) approach and (ii) object-oriented (OO) approach. The FO approach is the traditional approach for software development. In this approach, a system is viewed to perform some set of functions. OO approach is the latest approach. It focuses on the entities or objects of the system.

DFD, Data Dictionary, ER diagram, Decision Table, Structured Charts etc. are some of the important tools used in the FO approach. UML is the standard tool used in the OO approach. The tools are now integrated into system development software to automate software engineering methods. These software tools are called CASE tools.

The subject “Software Engineering” is inter-disciplinary in nature. Product, Process, People and Project are the four important core aspects of software development.

EXERCISES

1. What is software? How are these classified?
2. Explain how commercial software is different from computer programs written for academic purposes.
3. Explain how software, as a product, is different from other physical products.
4. Compare software development with manufacturing functions required for any physical product.
5. Describe some present-day applications of computers in business. Explain how these present-day applications are different from data processing.
6. Describe the stages in the evolution of hardware and software, called “generation of computers”.
7. Distinguish between low-level, high-level and fourth-generation programming languages.
8. Describe how modern programming techniques are different from the old techniques used in the past.
9. Explain the term “software crisis”. State the reasons for software crisis.
10. Describe the events that led to the evolution of software engineering as a discipline.
11. Describe the features of software engineering as a separate discipline.
12. Enumerate various factors that affect productivity of software. How can the quality of software be ensured?

This page is intentionally left blank.

SOFTWARE DEVELOPMENT PROCESS

This chapter describes the type of work involved in software development. It covers the following topics:

- Software processes
- Software development life cycle models
- Description of some popular models
- Process standards

After going through this chapter a reader will get an idea as to how software is created in practice.

Development of commercial software requires a great amount of effort, time and money. Different types of activities are required to be performed for development of software. Software development also involves people and their management. Hence, software development can be considered as a project. A successful project is one that satisfies the expectations of three major goals, viz., (1) low cost, (2) completion on schedule and (3) quality. According to software engineering principles, if the process for development of any software is right, the chance of success of the software project is greatly increased. Hence, software projects utilize some standard and proven processes for organization and execution of tasks to achieve these goals.

2.1 SOFTWARE PROCESSES

The term process means “a particular method of doing something”. It generally involves a number of steps or operations. In software engineering the term software process refers to the methods of developing software. Software process can be categorized into four major types of component processes as given below.

- (i) Software development process
- (ii) Software project management process
- (iii) Software configuration management process
- (iv) Software process management

Software development process is a structured set of activities that transform the users' requirements into a quality software product.

Planning and scheduling of tasks and monitoring their execution fall in the domain of project management process.

The business environment and needs of customers are never static. These keep changing with time. Hence, during the execution of the software development project, it may become imperative to make changes in the given software. Development processes generally do not focus on evolution and changes. To handle these, another process is used, which is called software configuration management process. The objective of this process is primarily to deal with managing changes in software requirements and its environment.

An organization needs to keep improving to remain competitive. Hence, to enhance its capabilities, an organization must adopt a process by systematically reviewing and improving all its processes. The process of understanding the current process, analyzing its properties, determining how to improve and then effecting the improvement is dealt with by the process management process. The basic objective of the process management process is to improve the software processes.

2.2 SOFTWARE DEVELOPMENT LIFE CYCLE MODELS

A software development process is a structure imposed on the development of a software product. There are several models that describe the approaches or the way different tasks are done during the software development process. These models are called "software development process" models.

In the physical sense, software does not wear out. It has no replacement (spare) parts. However, still the software has a lifespan. The system for which the software is made and the system's environment is never static. These keep changing due to developments in technology and market conditions. When the system is changed significantly, it may not be possible to update or maintain the software any longer. Hence, the software may be considered to have lived its lifespan. This kicks off development of new software and this cycle continues. Hence, software development process models are also popularly called as "software development life cycle (SDLC)" models.

The software development process deals with management and technical issues of software development. It comprises many types of activities that are performed according to some plan. These activities are: (1) requirements engineering, (2) software design, (3) implementation or coding, (4) testing or inspection and (5) maintenance or adaptation.

- The requirement engineering activities are concerned with understanding the purpose and environment of the system for which the software is meant and to determine its desired functional requirements and attributes. These activities may take up about 7–15% of the software developmental effort.
- The design activities are concerned with finding the technical solution, developing the modules and the blue print of software and developing logic for coding. These activities take up about 10–15% of the total developmental effort.
- The implementation activities are concerned with actual programming or coding in a computer language. These activities take up about 7–15% of the total developmental effort.
- The testing activities are normally performed at different stages of software development to detect if there is error in any work product. These activities are done to ensure that the software is developed without any error. Testing activities take up about 13–20% of the developmental effort.
- Even after software is installed and running, it needs modifications whenever there is any change in the larger system of which the software is a part. Hence, the software requires support activities

for correcting errors that may be detected in actual operation and for adaptation of software to meet business needs. Studies have shown that the maintenance activities may constitute more than 50% of cost and effort.

Software development process models can be categorized mainly into two types:

- (i) Sequential process models
- (ii) Iterative process models

In sequential process models the software is developed in a sequence of stages. The typical stages are analysis, design, coding, testing etc. Hence, after all the activities of analysis are complete, the design activities are performed and so on.

In iterative process models a prototype or a small part of the software is developed using the sequential process. After one part is completed, all the activities of the sequential process are repeated for developing the next part. Hence, the software is developed iteratively until the software being developed is completed satisfactorily.

2.3 WATERFALL MODEL

The waterfall model proposed in 1970 is a traditional model for software development. It is a sequential process model. According to this model the software development project is executed in some steps or phases. The phases of software development process are as under.

1. System engineering
2. Requirement analysis
3. Design
4. Implementation or coding
5. Verification or testing
6. Maintenance

The software development process proceeds from one phase to the next in a purely sequential manner. This model is called the “waterfall model” because the flow of the process from one phase to the next resembles the flow of water falling from steps. The software to be developed is always a part of a larger system. Hence, system engineering is the first stage in which requirements for the larger system are established. This stage ensures that the software will interface properly with people, hardware and other software in the system. The scope of the software project and project estimation is done in this stage. In the next stage all the activities of requirement analysis are done to determine software requirements. When the requirements are fully determined and the software specifications are finalized, the design phase begins. In the design phase, all the design activities are performed on the basis of requirement specification finalized in the requirement phase. Software design has mainly two parts: (1) high-level design and (2) low-level design. In high-level design the system is factored into logical building blocks and interfaces between them are specified. Low-level design consists of specifications of functions along with logic and algorithms. It is also called “detail level design” or “program specifications”. When the design is fully completed, its implementation is done by coding (programming). Generally, software is factored into a number of components (modules, functions or objects). Program codes of these software components are generally produced by different teams. Towards the later stages of the implementation phase these program codes are integrated together. After the implementation is complete, the software product is tested and debugged in the testing phase. Any faults introduced in earlier phases are removed here. Then, the software product is installed

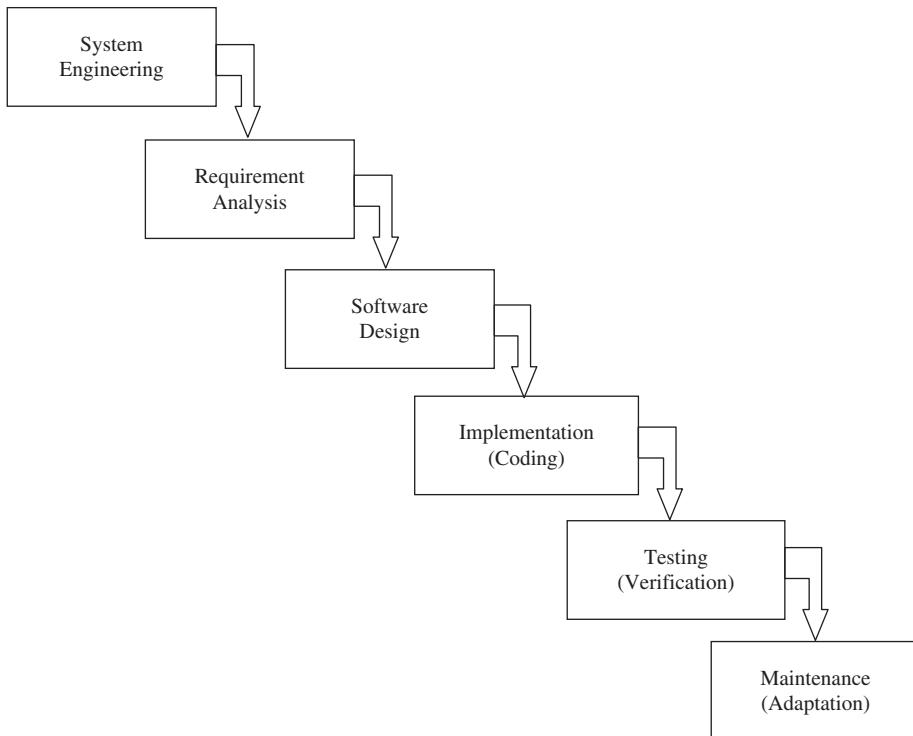


Figure 2.1 The waterfall model

and maintained in the maintenance phase. Maintenance means making changes to the existing software. It essentially consists of reapplying each of the above stages to the existing software rather than developing a new one. The block diagram of the waterfall model is shown in Figure 2.1.

Thus, the waterfall model maintains that one should move to a phase only when its proceeding phase is fully completed. Phases of development in the waterfall model are thus discrete, and there is no jumping back and forth or overlap between them.

The waterfall model has the following advantages:

- The way of structuring different activities in a software project appears simple and logical.
- Every stage produces a concrete deliverable such as requirement documents, code etc.
- It facilitates software contracting. Requirement specifications help in establishing contract. The deliverable can be tested against specifications.
- It provides a simple measure of the development status. Progress from one phase to the next indicates achievement of an important milestone.

It has some drawbacks as well.

- It is difficult to accurately and completely specify requirements prior to any implementation.
- Once the requirement specifications are established in the analysis stage, it is very difficult to make any changes in subsequent stages. Hence, the users are locked in to features that may not reflect their true needs.

- A working version of the software is not available until late in the development process and the users are completely cut off from the development until the acceptance-testing stage.
 - The phase of development does not necessarily reflect progress toward completion. Entering the test stage may mean anything from 20% to 80% completion.

The waterfall model is the earliest software process model and it is still very popular.

2.4 THE “V” MODEL

The “V” model is a recognized standard for development of IT systems. It lays down what is to be done, how it is to be done and what tools are to be used to develop an IT solution. Hence, accordingly, there are three levels of the “V” model.

- Software lifecycle process model
 - Methods to be used
 - Tool requirements

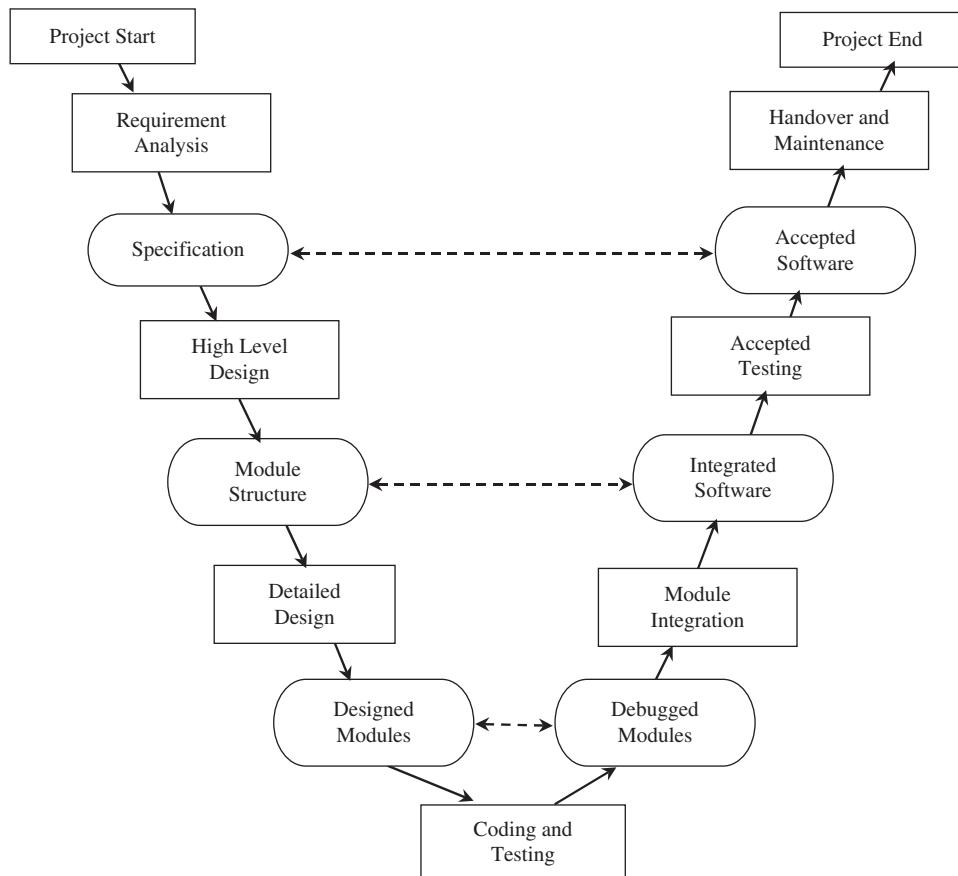


Figure 2.2 The “V”model for software development

The “V” model is structured into four sub-models. They are closely integrated and influence one another. They are:

- Project management (PM): It comprises project plans, monitoring and control mechanism and cost management.
- Software/system development (SD): To develop the total IT system or the software.
- Quality assurance (QA): It specifies quality requirements, test cases and criteria, and examines products and the compliance with standards.
- Configuration management (CM): It is a system to manage changes.

The “V” model for software development can be considered as an extension of the waterfall model. However, it gives more emphasis on testing. It recognizes various kinds of testing, like unit (module) testing and integration testing. The information used for testing comes not only from the coding phase but also from the earlier development phases. To indicate this information flow, the “V” model bends upward at the coding phase, thus bringing the corresponding phases opposite one another. Arrows indicate the relations between the deliverables on both sides of the V. The information flows from the development activities to the corresponding test activities.

The left arm of the “V” represents the analysis and design phases of software development. The point of the “V” is coding and testing, which is the implementation process. The right arm of the “V” represents the various stages that are required for testing and integration of the system components.

The “V” model deploys a well-structured method in which each phase can be implemented by detailed documentation of the previous phases. Testing activities like test designing start at the beginning of the project well before coding and therefore save significant amount of the project time.

2.5 PROTOTYPING MODEL

The users generally have a general view of what is expected from the software product. They are able to tell the general objectives of the project but not the detailed requirements such as various inputs, processing needs and output. It is up to the software developers to extract the necessary information required for software development. Sometimes there is a communication gap between the users and the software developers because of differences in their professional background. This problem is solved by the prototype model. When a prototype design is shown to the users, they are able to point out defects and offer suggestions for improvement. The prototype model allows the users to interact and experiment with a working representation of the product and thus it increases the flexibility of the development process. By seeing the prototype model, the users are better able to realize what the real system will be like. The prototype is refined till the users are reasonably satisfied. From this, the software developers are able to develop software that fulfils the real needs of the users.

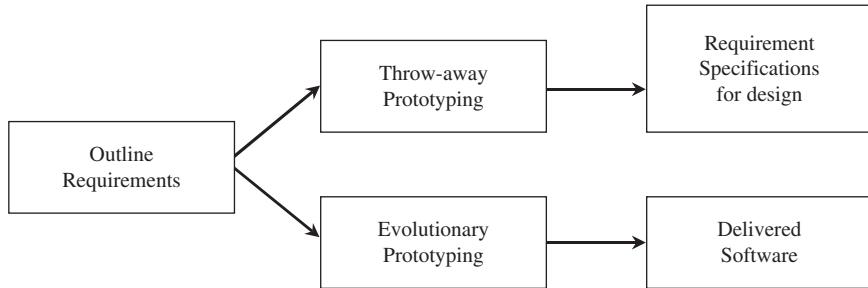
There are basically two main approaches to prototyping models.

1. Throw-away prototyping
2. Evolutionary prototyping

In throw-away prototyping, a prototype is produced to help in determining the software requirement specification and then discarded. The software is then developed based on software requirement specification that was determined with the help of the prototype.

The throw-away prototypes are used in two ways.

1. Prototype used for determination of requirements
2. Prototype used as the specifications for design



The prototype is used as a means to quickly determine the needs of users. The emphasis of prototype is on representing those aspects of the software that will be visible to the users (e.g. input screens and output formats). Hence, in throw-away prototyping, it does not matter whether the prototype works or not.

In the second approach, the prototype is actually used as the specifications for the design purpose. The advantage is speed and accuracy, as no time is spent on drawing up written specifications. The inherent difficulties such as incompleteness, contradictions and ambiguities associated with written specifications are also avoided.

In evolutionary prototyping, an initial working model of the software or part thereof is developed based on abstract or outline specification. This prototype is then evaluated by the users. Based on users' feedback it is refined through a number of stages to get the final product.

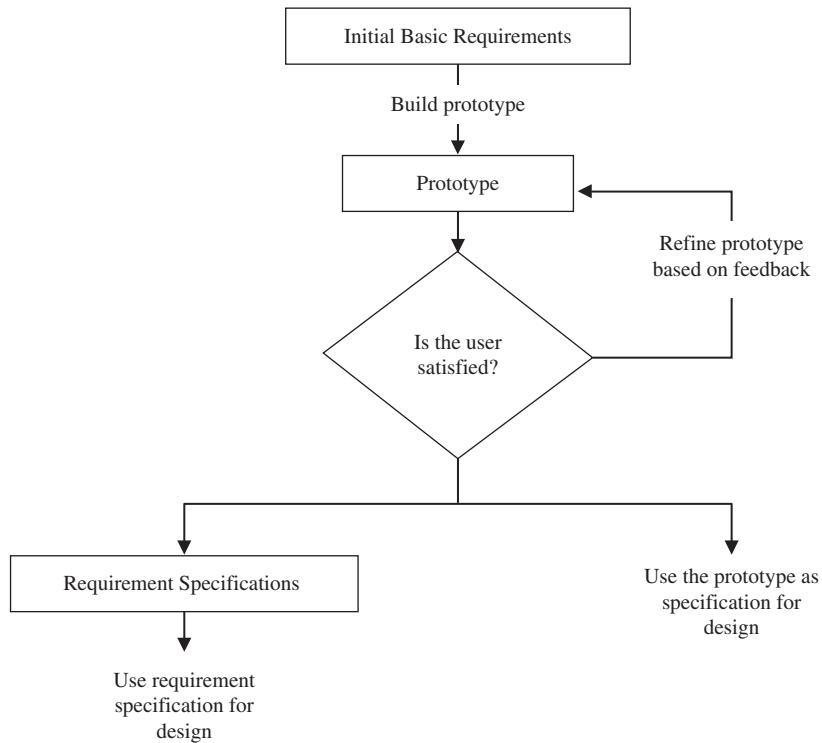


Figure 2.3 The prototyping model

Evolutionary prototype is used for systems where the specification cannot be developed in advance (e.g. AI systems and user interface systems). Verification is not possible in such cases for want of documented specifications. Hence, it may be better to develop software using an evolutionary approach rather than first build a throw-away prototype and build software the second time.

Software creation by using the prototype model has many advantages.

- Instead of concentrating on documentation, more effort is placed in creating the actual software.
- The software is created by using lots of feedbacks from users. Every prototype is created to get the views and opinions about the software from users.
- Since the users are involved in software development, there is a greater chance of the resulting software being more acceptable to them.
- If something is unfavorable, it can be changed. The software is created with the customer in mind.
- Over design could also be avoided using this model. “Over design” happens when the software has so many features that the focus on core requirements is lost. Prototyping facilitates in giving only what the users want.

Disadvantages of prototyping

- Often users feel that a few minor changes to the prototype will suffice for their needs. They do not realise that the prototype is meant only for demonstration purposes and it is not the real software. A lot of other work such as algorithm design, coding, debugging and quality-related activities are done for developing the real software.
- The developers may lose focus on the real purpose of the prototype and compromise on the quality of the product. For example, they may retain some of the inefficient algorithms or inappropriate programming codes of prototype in the real software.
- A prototype has no legal standing in the event of any dispute. Hence, the prototype as the software specification is used only as a tool for software development and not as a part of the contract.

Any proposed designs that are new involve risk. In some cases it may not be possible to decide all the features of the system in advance. Hence, unique situations, about which developers have neither information nor experience, are often selected for prototyping.

2.6 THE ITERATIVE WATERFALL MODEL

The waterfall model was widely adopted in the early days of software development. However, it is intrinsically flawed. It is very rare that requirement analysis can be entirely completed before design and design before development and so on.

The shortcomings of this model became apparent and other versions of the waterfall model called the iterative waterfall model have been developed.

This model attempts to overcome the limitations of the original model by adding an “iterative” loop to the end of the cycle. That is, in order to keep up with changing requirements the “analysis” phase is revisited at the end of the cycle and the process starts over again.

This alleviates the situation somewhat but still introduces a considerable lag between analysis and implementation. The waterfall model requires that all the steps be completed before the process is repeated again in the next cycle. If requirements change during the life of the project, the waterfall model requires the completion of a full cycle before these can be revisited.

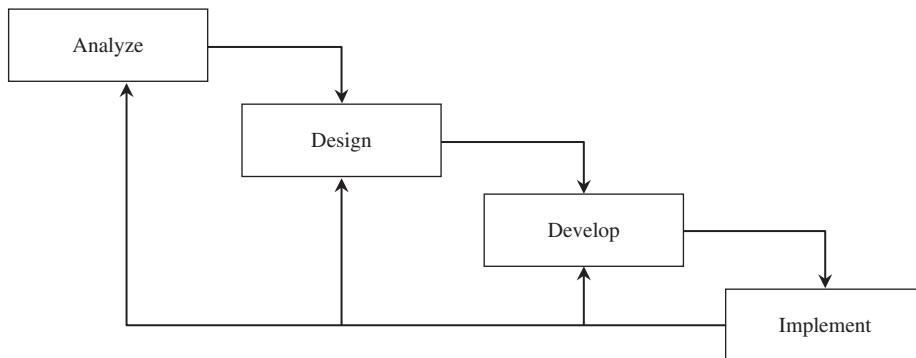


Figure 2.4 The iterative waterfall model

In the iterative waterfall model, refinement of software is done in a series of iterations. In each iteration, some functionalities are added or modified. The usual series of steps is followed in each iteration, as done in case of the waterfall model. Hence, the iterative waterfall model can be considered as a series of classical waterfall model. For this reason the iterative waterfall model is also sometimes called the incremental waterfall model.

2.7 THE SPIRAL MODEL

The spiral model was proposed by Barry Boehm in 1988. It is an iterative model. It uses iterative cycles for software development. Each cycle consists of four steps.

- | | |
|------------------|--|
| 1. Planning | Determining objectives, alternatives and constraints |
| 2. Risk analysis | Evaluating alternatives, identifying and resolving risks |
| 3. Engineering | Developing and verifying the “next level” product |
| 4. Validation | Evaluating the customer/assessing the results and suggesting modifications |

The usual procedure in the spiral model can be generalized as follows:

First, the core software requirements are understood. This usually involves interviewing a number of users and application of other requirement engineering techniques. The requirements are analyzed and various possible solutions are evaluated. Based on this a preliminary design is created. Then, the first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product. The first prototype is evaluated in terms of its strengths, weaknesses and risks.

A second prototype is evolved based on the evaluation of the first prototype. The process of defining the requirements, planning, designing, constructing and evaluating is repeated as was done in case of the first prototype. During evaluation if the amount of risk is felt to be very high, the entire project can be aborted. Risk factors might involve development cost overruns, operating cost miscalculation or any other factor that could result in a less-than-satisfactory final product.

The second prototype is evaluated in the same manner as the first. If it is felt necessary, subsequent prototypes are developed from it according to the four-fold steps outlined above. The refined prototype represents the final product. The software is constructed, based on the refined prototype. The developed software is thoroughly tested. Routine maintenance is carried out on a continual basis. The spiral model is shown in Figure 2.5.

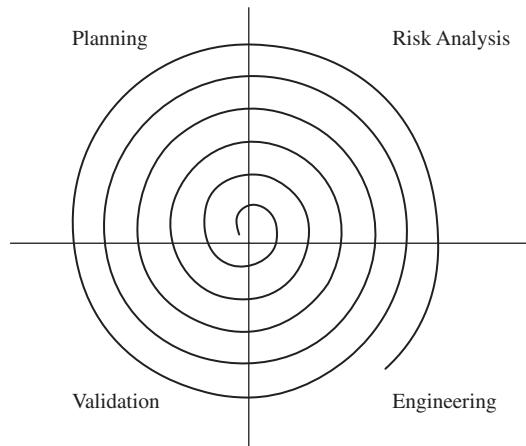


Figure 2.5 The spiral model

The radial dimension represents the evolution toward the improvement of software. In each iteration, the results of the previous iteration are analyzed for making an improved plan. This is then reviewed to determine the risk and to build a new prototype. This way the software is built progressively toward completion.

During the first cycle, the objectives, alternatives and constraints are defined. Risks are identified and analyzed. If risk analysis indicates substantial uncertainty in requirements, prototyping may be done. The engineering step may follow the life cycle approach or a prototyping approach. The users evaluate the work and make suggestions for modifications. Based on these, the next stage of planning and risk analysis is done. The software thus evolves toward a complete version.

The spiral model uses a risk management approach to software development. Some advantages of the spiral model are:

- It defers elaboration of low-risk software elements.
- It incorporates prototyping as a risk-reduction strategy.
- It gives an early focus to reusable software.
- It accommodates life-cycle evolution, growth and requirement changes.
- It incorporates software quality objectives into the product.
- It focuses on early error detection and design flaws.

The disadvantages of the spiral model include the following:

- It does not work well with contract work.
- It relies on the ability of software engineers to identify sources of risks.

The spiral model is focused on risk management. It is used most often in large complex projects.

2.8 PROCESS STANDARDS

The quest for the optimal mix of processes has resulted in different standards. It is now accepted that if an organization will manage its processes well, the products are bound to be of good quality. ISO/IEC

12207 standard is one of the popular process standards, which was proposed in 1988 and published in August 1995. It was created to establish a common international framework to acquire, supply, develop, operate and maintain software.

ISO/IEC 12207 standard consists of three types of processes:

- (i) Primary life cycle processes
- (ii) Supporting life cycle processes
- (iii) Organizational life cycle processes

This standard is applicable for all products that may be a software product, software service or the total system.

The capability maturity model (CMM) is one of the leading process models and is mostly applicable to software development. It is a framework to assess an organization's capability. The CMM covers practices for planning, engineering and managing software development and maintenance. It describes the key elements of effective software processes with an objective to gradually bring about improvements in them. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality and product quality. The CMM establishes a yardstick to measure the maturity of an organization's processes.

ISO 15504, also known as SPICE (software process improvement capability determination), is a "framework for the assessment of software processes". This standard is aimed at setting out a clear model for process comparison. SPICE is used much like CMM. It is a process model to manage, control, guide and monitor software development. This model is used to measure what a development organization or project team actually does during software development. This information is analyzed to identify weaknesses and drive improvement. It also identifies strengths that can be integrated into common practices to be followed in the organization or the team.

The process standard is useful for quality assurance. These are dealt with in some detail in subsequent chapters.

Problem for Discussion

ABC University (not the real name of the university) is a state technical university. It has four constituent colleges and a large number of affiliated colleges located at different places inside the state.

The university wants to develop software for its examination system. It is desirable that the software should also help other functional areas of the university such as academics, finance, establishment, library etc.

Some questions and suggestions:

- A) If you are engaged as a consultant for developing the software, how will you proceed?
- B) It is suggested that the following activities should be performed for developing the software:

1. Gather information about why the software is needed and who initiated the proposal for development of the software.
2. Analyze if development of the software is feasible.
3. Estimate the probability that the software development project will be successful.
4. Gather information about different disciplines and courses taught in the university.
5. Gather information about the approximate number of students enrolled in the different disciplines.

6. Estimate the order of expected budget for developing the software.
7. Estimate expected time and effort that will be needed to develop the software.
8. List various activities that should be performed.
9. Draw the Network (CPM/PERT) diagram.
10. Distribute work among programmers to write programs.
11. Develop the graphical user interfaces (GUIs).
12. Integrate the program units to build the software.
13. Test each unit program for correctness.
14. Decide on what functionalities should be built into the software.
15. Decide on what should be the different modules of the software.
16. Decide on whether the software will be implemented in a stand-alone computer or over a network.
17. Decide on whether the DBMS will be Oracle, DB2 or Sybase.
18. Decide on whether the software will be Window based or Linux based.
19. Design each module in detail.
20. Test the software for correctness.
21. Deliver the software.

C) Classify the above activities into the following categories:

- a) Requirement engineering
- b) Software design
- c) Program coding
- d) Software testing
- e) Project management

D) Suggest the order in which the above activities should be carried out.

E) What according to you are the two most critical activities where special care should be taken?

F) Discuss whether the software should be built sequentially or iteratively. There can be many views and differences of opinion on the model to be followed. Some of the views may be as under.

- (i) The classical waterfall model is the basic model. It provides a simple measure of the development status. Progress from one phase to the next indicates achievement of important milestones. It is simple to understand and follow.
However, it appears that the university authorities are not very clear about what they really expect from the software. Hence, some errors are likely to be committed during the initial phases. The waterfall model is not good at handling errors because errors committed during any of the phases are passed on to subsequent phases.
- (ii) The iterative waterfall model may be a better choice because it is good at handling errors. This model is also simple to understand and use.
However, this software project appears to be a large project and involves many risks. Hence, the iterative waterfall model may not be suitable.
- (iii) Since the purpose of the software is not very clear, it may be better to build a prototype.
However, since it is a large project and the client is a state university, it is desirable to have clear documented specification as a part of the contract before proceeding for software development.
- (iv) What about the “V model” and the “spiral model”?

SUMMARY

The term “software process” refers to the methods of developing software. It has four major component processes, namely, (1) software development process, (2) software project management process, (3) software configuration management process and (4) software process management process.

Software development process is a structured set of activities that transform the user requirements into a quality software product. Sequential process models and iterative process models are two main categories of models for software development. In the sequential process models the software is developed in a sequence of stages. The typical stages are analysis, design, coding, testing etc. In the iterative process models, a prototype or a small part of the software is developed using the sequential process. After one part is completed, all the activities of the sequential process are repeated for developing the next part.

The waterfall model is a traditional sequential process model for software development. In this model, the software development project is executed in some well-defined phases.

The V model for software development can be considered as an extension of the waterfall model. It stipulates various kinds of testing, like unit (module) testing and integration testing. The information used for testing comes not only from the coding phase but also from earlier development phases.

The prototyping model is an iterative process model. Throw-away prototyping and evolutionary prototyping are two main approaches to prototyping models. A throw-away prototype is used only for checking the software requirement specification. In evolutionary prototyping, an initial working model of the software or part thereof is developed based on the abstract or outline specification. This prototype is evaluated and refined through a number of stages to get the final product.

The iterative waterfall model attempts to overcome the limitations of the waterfall model by adding an “iterative” loop to the end of the cycle. After the end of each phase, its previous phases are revisited to modify requirements and remove errors.

The spiral model uses iterative cycles for software development. Each cycle consists of four steps, namely, (1) planning, (2) risk analysis, (3) engineering and (4) validation. The spiral model gives much emphasis on risk management.

Different process standards have developed to ensure software quality. ISO/IEC 12207 standard, CMM, ISO 9000 standards and SPICE are some popular process standards.

EXERCISES

1. Briefly describe the different types of activities that are performed in the software development process.
2. Explain the problems that might be faced by an organization if it does not follow any software life cycle model.
3. List various stages of the waterfall model. What are the advantages and disadvantages of this model?
4. Compare the V model with the waterfall model.
5. What is a software prototype? Identify the reasons for the necessity of developing a prototype during software development.
6. Distinguish between throw-away prototyping and evolutionary prototyping.
7. Distinguish between sequential process models and iterative process models of software development.
8. Compare the iterative waterfall model with the waterfall model.
9. List the four stages that a software development project goes through in each cycle of the spiral model. Is this model more suitable for small projects or big risky projects? Explain.
10. Write short notes on:
 - a. Software configuration management
 - b. Software process management
 - c. Process standards

This page is intentionally left blank.

SOFTWARE REQUIREMENT ENGINEERING

This chapter explains the need for adopting a rigorous process for determining correct software requirements. It describes the “Requirement Engineering” (RE) process. It covers the following topics:

- *Types of software requirements*
- *Methods and techniques used in RE*
- *Activities involved in the RE process such as Inception, Elicitation, Elaboration, Negotiation, and Validation*
- *Structure of Software Requirement Specification (SRS)*
- *Characteristics of the RE process*

After going through this chapter a reader will get a feel of software engineering.

Software is developed to fulfil some purposes and objectives. To fulfil the objectives, the software has to perform certain functions. Hence, to develop any software, it is first necessary to understand two things:

1. What is the purpose for which the software is being developed, and what are the objectives it has to fulfil?
2. What are the functions that the software has to perform to fulfil its objectives?

Software requirements stipulate what must be accomplished, transformed, produced, or provided. They refer to capability that must be met or possessed by a software component in order to meet contract or specification requirements, quality standards, as well as stated and implied needs of users.

Software requirements are the basis on which the software is developed. Hence, it is very important to determine software requirements correctly. It needs to be done properly so that:

- the needs and desires of customers/users about what they expect from the software are known.
- there is no gap between the perception of customers'/users' requirements and the actual requirements that the software should fulfil.

- the important requirements of software, which are implied but not stated by the customers/users, are visualized.

Rigorous methods and processes are used for determining the requirement specification of the software. The process to determine the requirement specification of the software is called the Requirement Engineering (RE) process. RE is a systematic approach for determining Software Requirement Specifications (SRSs). It is an iterative process of analyzing the problem, documenting the observations in some standard representation formats, and checking the accuracy of the understanding gained to determine the requirement specifications of the software. The task of RE is performed by software engineers, researchers, and system analysts.

3.1 REQUIREMENT ENGINEERING PROCESS

The first step in RE is to gain an understanding of the problems for which the software is being developed. For this, all necessary information about the problems are gathered. Based on the information, the initial requirement specification of the software is developed. This initial specification is further refined after discussions with the prospective users and stakeholders.

RE is among the most critical and most difficult tasks faced by software engineers. It involves transforming system requirements into a description of software requirements, performance parameters, and software configuration. It consists of two important tasks.

- (i) Finding the requirement specifications, more popularly called work product.
- (ii) Reviewing the work products (specifications) with customers/users to ensure their correctness.

What functions will be included in an application is an important decision. The decisions regarding the functions of a project are influenced by:

- the needs of the organization
- the risk (business and technical) associated with the project
- the resources (e.g. budget, staff) available for the project
- the technology available in the organization

Generally, a feasibility study is undertaken at the beginning of a project. The feasibility study is based on the outline requirement specification. The outline specification spells out the broad purpose for which the software is needed. Examples of outline requirements may be:

- The organization needs an application to comply with a new tax law.
- The organization needs an application to manage inventory more efficiently.
- The organization needs an application to manage human resources more efficiently.

However, outline specification is not sufficient for design and construction of software. For this, detailed requirement specification is needed.

3.1.1 Types of Software Requirements

As per Glossary of Software Engineering Terminology given in IEEE Standard 610.12-1990, there are six types of SRSs.

- (i) *Design requirement*: It specifies the design of a system or system component.

- (ii) *Functional requirement*: It specifies the function that a system or system component must perform.
- (iii) *Implementation requirement*: It specifies the coding or construction of a system or system component. This affects the technical effectiveness.
- (iv) *Interface requirement*: It specifies the external items with which the system or its component must interact, and it sets forth constraints on formats, timing, or other factors caused by such interaction.
- (v) *Performance requirement*: It imposes conditions on a functional requirement; for example, a requirement that specifies the speed, accuracy, or memory usage with which a given function must be performed.
- (vi) *Physical requirement*: It specifies a physical characteristic that a system or system component must possess; for example, material, shape, size, weight.

The RE process is accomplished by execution of some distinct activities. The important activities of requirements engineering process are: Inception, Elicitation, Elaboration, Negotiation, Specification, Validation, and Management. These activities are described in subsequent sections.

3.2 REQUIREMENT INCEPTION

The inception function is concerned with understanding the situation and events that have initiated the software project. This helps one to understand how the software being developed is going to be useful and thereby determine what features the software should possess. Hence, it is important to know the following:

- Has the software project been initiated as a result of problems relating to maintenance or relating to operation of some existing software system?
- Has only a single event or has a sequence of events triggered the development of a new computer-based system?
- Has the need for software evolved abruptly or has it evolved over a period of time?

3.2.1 Identification of Stakeholders

A stakeholder is defined as “anyone who benefits in a direct or indirect way from the software system being developed”. The usual stakeholders are management, marketing people, consultants, system analysts, software engineers, and support/maintenance engineers of a software firm; and buyers, and end-users of the software. These stakeholders have different needs. The quality of software is determined by the extent to which it satisfies the needs of all its stakeholders. Hence, at inception, the requirements engineer identifies different stakeholders.

These stakeholders provide inputs for determining software requirements. The initial list may grow as the requirements engineer gathers information about the system and identifies more such stakeholders. These stakeholders have different views of the system. They derive different benefits when the system is successfully developed and are exposed to different risks if the development effort should fail. Hence, it is important to identify different categories of stakeholders and examine their varied viewpoints. For the sake of convenience we have categorized them into two classes.

1. One class of stakeholders are those for whom the software is made. These are people whose jobs are affected/improved (e.g. employees); who use the software (e.g. managers, operators, clerks);

who initiate and take decision for software development (top executives) etc. We shall refer to these stakeholders as “software users” or simply “users”.

2. The other class of stakeholders are those who work to build and sell the software to the users. They include managers of the software firm, marketing people, consultants, system analysts, software engineers, and support and maintenance engineers. We shall refer to these stakeholders as “software developers” or simply “developers”.

The needs of all the stakeholders should be considered while developing the software. Hence, to determine SRSs, all the stakeholders should be involved in the RE process.

Example:

Let us consider the development of an e-commerce website for a travel/tour operator firm. The firm provides various kinds of services to tourists. The important services are:

- Booking of railway and air tickets to different places
- Hotel booking at different places
- Guided tour in luxury buses at various locations
- Car rental
- Various standard package tours for different categories of people
- Customized package tours
- Providing travel information about places of interest

We can identify various stakeholders for this e-commerce website. A probable list is given in Table 3.1.

Can we identify more stakeholders? Presently, the firm is booking hotels for its customers through its tour agents. However, in future the travel firm may want to include the facility of hotel booking directly through the website. Hence, can we include hotels as future stakeholders? This way the scope of software may expand. Though software is developed keeping in view the future needs, we have to limit the scope of the software at some point to complete it in time.

3.3 REQUIREMENT ELICITATION

Elicitation is about seeking information about the software, the system, and the business. The following information can be sought from the customers and the users:

- What are the objectives of the software product?
- What is to be accomplished by the software?
- How does the software fit into the needs of the business?
- How is the software to be used on a day-to-day basis?

Elicitation of information about software requirement is similar to how a journalist gathers information to analyze and report a news item. To gather the necessary information, journalists sometimes use a technique called “5Ws and an H”. The “5Ws and an H” referred to are:

- | | | |
|------------|------------|--------------|
| (i) Who? | (ii) What? | (iii) Where? |
| (iv) When? | (v) Why? | (vi) How? |

The questions based on “5Ws and an H” can be asked to elicit information about software requirements. Table 3.2 below shows how these questions can be used to elicit information about users’ requirements.

Table 3.1 Stakeholders of an E-commerce Website

| | Stakeholders | How benefited/affected |
|---|---|---|
| U | Owner of the firm | Likelihood of more business and profit Bear the cost of software development Bear the risk if the project fails |
| S | Employees of the firm | Work becomes easier and workload is reduced Way of doing work changes Career growth of employees is affected More financial incentives |
| E | Tour agents | Some employees may become redundant Likelihood of more business and profit |
| R | Bankers of the firm | Likelihood of being subjected to greater control by the firm Some of their services may not be required |
| D | Customers of firm (tourists) | Must provide e-banking facilities |
| E | Management of the software firm | They get better service |
| V | Marketing people | Likelihood of more profit |
| E | Consultants, system analysts and software engineers | Likelihood of more orders for software |
| L | Support/maintenance engineers | They develop the software and get paid by the software firm |
| O | | |
| P | | |
| E | | |
| R | | |

However, getting answers to the above questions is not that simple. Three main problems are faced during requirement elicitation.

- (i) **Problems of scope.** The boundary of the system is ill-defined. The users may specify unnecessary technical details that can confuse, rather than clarify, overall system objectives.
- (ii) **Problems of understanding.** A user generally has limited knowledge of:
 - what is needed from the software
 - the capabilities and limitations of the computing environment of the organization
 - the scope and domain of the problem that the software is supposed to solve
 Also, the users sometimes have trouble communicating their needs to the system engineer. They specify requirements that are sometimes ambiguous and not verifiable. A user may omit certain important information that may seem obvious to any internal member of the organization. However, the same information may be quite unknown to the system engineer because he is not conversant with the organization's problems. Requirements specified by one user may conflict with the needs of other users.
- (iii) **Problem of volatility.** Requirements change over time. This is due to the dynamic environment in which most organizations operate. The problem of volatility of business requirement is an important aspect that needs to be considered during software development. Otherwise the software may become obsolete before it is implemented.

Table 3.2 Questions for Requirement Elicitation

| Question | Manifestation in user requirements |
|----------|---|
| Who? | To know about organizational unit, type of job, or client to which the requirement relates |
| What? | To know about tasks that need to be performed; i.e. functional requirement |
| Where? | To know about type of machine (e.g. client, server etc.) on which the processing is required to be done |
| | One can know several aspects of the requirements through this question. Some examples are: |
| When? | <ul style="list-style-type: none"> – Performance: After how much time must the task be completed after its initiation? – Order: What is the sequence in which different tasks must be initiated and completed? – Period: What is the time period of transaction data time to which the processing logic should be applied? |
| Why? | To know the rationale behind the requirement of why the software is needed |
| How? | To know how the tasks are initiated and performed and how the results are delivered. Asking “how” determines the methods used to deliver the functionality. It is concerned with understanding data entry, data processing, and mechanisms for dissemination of information. |

Hence, some standard techniques are used for elicitation of users' requirements. Some of these techniques are discussed below.

3.3.1 Requirement Elicitation Through Interview

For elicitation of users' requirement, both qualitative and quantitative forms of information are necessary. Interviews are often the best means to elicit qualitative information such as opinions, policies, and narrative descriptions of activities or problems. Many people, who are reluctant or not able to express themselves well in writing, feel free to discuss their ideas verbally. Furthermore, it is often easier to schedule an interview with senior managers than to have them fill up questionnaires.

The analyst can interview people one at a time or in groups. Depending on the requirements, the interviews can be either structured or unstructured. Structured interviews are more formal and standardized. The questions to be asked in the interview are generally framed in advance. The structured interview ensures uniform wording of the question for all respondents. It facilitates objective evaluation and analysis of responses. It is easy to administer and requires much less time. Though it requires some amount of thinking and effort to frame the questions, little skill is required to conduct interviews. However, a structured interview may not be suitable for all situations. A high degree of structure may restrict spontaneity and clarity of responses and thus not be liked by some people (specially the top executives). Hence, when the analyst wants objective information on specific items, structured interviews are preferred.

An unstructured interview uses free question and answer sessions. The open and free atmosphere provides greater opportunity to learn about the feelings, ideas, and beliefs of the respondents. The interviewer can cover a wider area. The spontaneous discussion during the interview may elicit issues that were overlooked or were thought to be unimportant. The interviewer can also ask clarification about certain points that are not well understood or are ambiguous. Unstructured interviews require less time to set up because the precise wording of questions is not needed in advance. However, it takes more time to gather essential

facts through unstructured interviews. Analysis and interpretation of results also takes more time than for a structured interview. Because of the subjective nature of assessment, the interviewers may introduce their biases in reporting results.

3.3.2 Requirement Elicitation Through Questionnaire

Due to time constraints, only a limited number of persons can be interviewed. A questionnaire is a convenient way for the analyst to contact a large number of persons to get their views about various aspects of the system. When the system is large or the scope of study covers several departments, questionnaires can be distributed to all appropriate persons inviting them to furnish necessary facts about the system. Wide distribution ensures that the respondent has greater anonymity and encourages more honest answers. Standardized questions can also yield more reliable data. However, getting an adequate response to questionnaires is often a big problem. Even though distribution may be made to a large number of individuals, total response is very rare. In business organizations, people are generally busy and they take their own time in completing questionnaires. Generally, they do not give much priority to filling questionnaires.

3.3.3 Record Review

A good volume of records and reports are often available in many organizations, which provide useful information about the existing system. The term “records” refers to the written policy manuals, regulations, and standard operating procedures that most organizations maintain as a guide for managers and employees. Manuals are documents that describe existing procedures and operations of the organization. However, manuals and standard operating procedures available in most organizations are often old and out-dated and they differ from normal practices being followed in those organizations. Records enable analysts to become familiar with current practices and it gives an idea about the volume of transactions. Careful study of how the various types of forms are used provides a better understanding about current practices being followed in the organization. Reports of previous studies, consultant briefs, and management reports also provide insight into various finer points. It provides the rationale behind some of the things that the analyst may find difficult to comprehend.

3.3.4 Observation

Observing people in the act of executing their jobs is an effective technique of gathering facts about a system. Observation as a fact-finding technique is widely accepted by scientists, sociologists, psychologists, and industrial engineers for studying various activities being performed in an organization.

Observation provides first-hand information about how the activities are actually carried out. It enables the analyst to find out whether specific steps as prescribed in the procedure manuals are actually followed in performing various activities. For example, an analyst who wants to find out how decisions are taken by senior-level managers can observe the type of information that are sought, how soon these are provided, and from where these come from.

3.3.5 Collaborative Requirement Gathering

In order to absorb risk and avoid future dispute, all stakeholders need to be involved in the requirement elicitation process. Hence, collaborative requirement gathering is a popular approach for requirement elicitation.

Collaborative requirement gathering is nothing but a team-oriented approach of gathering requirements. For this, a meeting is scheduled and members of the software team and other stakeholders of the organization are invited to attend the meeting. Usually, a facilitator/coordinator is chosen to organize and conduct the meeting in a planned way for getting fruitful results.

The inception activities establish the scope of the problem and the overall perception of a solution. Based on the understanding gained out of these initial meetings, the stakeholders make a write-up of the system/software product request. The product request is distributed to all attendees well in advance before the meeting date. During the meeting, each attendee is asked to make a list of the following:

- Identify the objects (entities or elements) that are parts of the system's environment.
- Identify the objects that are produced by the system.
- Identify the objects that are used by the system to perform its functions.
- Identify the services (processes or functions) that manipulate or interact with the objects.
- Identify the constraints (e.g., cost, size, business rules).
- Identify the performance criteria (e.g., speed, accuracy).

The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

We can summarize the process of collaborative requirement phase as follows:

- Meetings are conducted and attended by software engineers, customers, and with other interested stakeholders.
- Rules for preparation and participation are established.
- An agenda that covers all important points is suggested for the meeting. However, free flow of ideas is encouraged in the meeting in a planned way.
- A "facilitator" can be a customer, a developer, or a consultant who coordinates the proceedings of the meeting.
- The meeting is conducted: (i) to identify the problem; (ii) to propose various possible solutions; (iii) to negotiate different approaches; and (iv) to arrive at a preliminary set of solution requirements

The collaborative process tries to involve different stakeholders in determining the functional requirement of the software. Teamwork and cooperation of all stakeholders is very important for software development.

Example

Let us consider an example of a collaborative requirement gathering for a software project, where a banking company wants to replace its standard teller counters for money withdrawal with Automatic Teller Machines (ATMs).

The first step is to determine various functional requirements of an ATM. The requirement gathering is done in a team that comprises representatives from marketing, software engineering, hardware engineering, and manufacturing. An outside consultant/facilitator may also help in the requirement gathering process.

"Justification for the new system/product" is usually the first agenda or topic of discussion in the requirements gathering team meeting. Everyone should be convinced about the necessity of the new system/product being developed. Once agreement has been reached, each member presents his list for discussion. Each member of the team develops the list of objects that will comprise the ATM system. Objects described for the ATM might include the note counting machine, printing machine, card reader,

and control panel with display device etc. The list of services might include loading the note counting machine, loading paper to the printing machine, and programming the control panel. In a similar fashion, each member of the team then develops a list of constraints. For example, some of the constraints for the ATM might be: the need for the system to be user friendly, the need for the system to be interfaced directly to a central server, the need for the system to reject the cards inserted by customers if these are not proper etc. The list for performance criteria might include: time duration (say 1 second) within which the system should recognize the card, optimum manner in which the notes should be counted for disbursement to customers. The lists can be either pinned to the walls of the room using large sheets of paper or posted on an electronic bulletin board.

After individual lists are presented, a combined list is created by the group. The combined list eliminates redundant entries, adds any new ideas that come up during the discussion. However, usually no items are deleted from the list. After combined lists for all topic areas have been developed, the facilitator coordinates the discussion. The combined list is shortened, lengthened, or reworded to properly reflect the system/product being developed. The objective is to develop a consensus list in each topic area (objects, services, constraints, and performance). The lists are then set for action.

Once the consensus lists have been completed, the team is divided into smaller sub-groups. Each of these sub-groups works to develop the specifications for one or more items of the lists. For example, the item specification for the object “Control Panel” can be as follows:

“The control panel is a unit that is approximately 8×8 inches in size and is put near the bottom of the display device. The control panel’s interaction with users occurs through a keypad. The keypad contains numeric keys from 0 to 9 and two control keys “Cancel” and “Enter”. A “ 14×14 ” inch LCD display shows different menus and screens. The software provides interactive prompts, messages, and similar functions.”

Each sub-group then presents its item specifications in the team meeting for discussion. Additions, deletions, and further augmentation are made to the specifications. In some cases, the development of item specification uncovers new objects, services, constraints, or performance requirements of the software. These are then added to the original consensus list. Many issues may be raised during discussions in the team meetings. Some of these issues may remain unresolved during the meeting. For this an “issue list” is usually maintained so that the issues raised can be reviewed and acted on later.

After the item specifications are completed, each team member makes a list of validation criteria for the product/system and presents the list to the team coordinator. A consensus list of validation criteria is then created. Finally, any one or more participants are assigned the task of writing a complete draft specification using all inputs from the meeting. This draft specification or work product is usually reviewed in the team meeting and finalized.

3.3.6 Output of Requirement Elicitation

The work product of the requirement elicitation process contains initial user requirements. It usually includes the following:

- A statement of need and feasibility
- A statement of scope for the system or product
- A list of customers, users, and other stakeholders who participated in requirements elicitation
- A description of the system’s technical environment

- A list of requirements (preferably organized by function) and the corresponding constraints for each requirement
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions
- Any prototypes developed to define requirements

However, the work products produced as a consequence of requirements elicitation may vary depending on the size of the system or product being built.

3.4 REQUIREMENT ELABORATION

The information obtained from the users during inception and elicitation is expanded and refined during elaboration. These requirement engineering activities focus on developing a refined technical model of software functions, features, and constraints. Elaboration is driven by creation and refinement of software requirements in users' perspective. It describes how the end-user will interact with the system.



The output of elaboration is an analysis model. It defines the informational, functional, and behavioral domain of the problem.

For illustration purposes, let us consider a model with three categories of requirements.

- (i) Initial User Requirements
- (ii) Initial Technical Requirements
- (iii) Final Functional Requirements

3.4.1 Initial User Requirements

This represents requirements that users felt prior to their closer interaction and discussion with the software developers. It may have one or more of the following characteristics.

- *Incomplete*: Initial user requirements may lack functions necessary for referential integrity.
- *Lack utility functionality*: Users may assume certain requirements as obvious and may forget to list such essential requirements. For example, certain essential validation reports or inquiries may be missing in initial user requirements.
- *Impossible/difficult to implement*: A user may ask for an on-line inquiry that may be difficult to implement due to constraints of CPU processing time.
- *Too general*: The requirement is ambiguous and does not specify necessary details for its implementation.
- *Varying functional needs*: Requirements of one user may be different from those of another user.
- *Beyond application boundaries*: The requirement is beyond the specified scope of current and/or future application.
- *Expressed in a different context*: Initial user requirements may refer to the physical or manual aspects of the system, which is not applicable in the context of a computerized system.

Example

A user (manager) in the Human Resource (HR) department of a corporation expresses his requirements as:

“Whenever I am working with an employee, I want to be able to view the employee’s information by entering his or her name.”

This requirement implies the development of an inquiry screen and a group of data items on employees. Functions of initial user requirements might be identified as:

1. Query on a specific employee
2. Employee data file
3. Functions to maintain a data file such as create, update, and delete employee record

3.4.2 Initial Technical Requirements

This step represents the software developers’ view of requirements. The initial technical requirements may include elements that are necessary for the implementation, but are not stated in the initial user requirement. Continuing with the same example, the software developer may feel that an index is necessary to speed up the retrieval of specific employees. Thus, the initial technical requirements might be identified as:

1. Query on a specific employee
2. Employee data file
3. Functions to maintain a data file such as create, update, and delete employee record
4. Index on the employee data file

In this step, the software developers review initial users’ requirements based on technical considerations. However, in the process, the software developers might make some error in determining the needs of users due to following reasons:

- The software developers may give too much emphasis on technical constraints in determining the requirements.
- The software developers may be unfamiliar with certain terminology of users and vice versa.
- The software developers may make certain assumptions about users that are not true.

Since neither the initial users’ requirements nor the initial technical requirements are complete, it is necessary to integrate both to get final functional requirements.

3.4.3 Final Functional Requirements

The third step of determining final functional requirements is usually decided jointly by the users and software developers. The joint sessions are necessary to achieve consistent and complete functional requirements. This step is the final version of the functional requirements after which the design activities are initiated. The desirable characteristics of final functional requirements are as follows:

- Contains terminology that can be understood by both users and software developers
- Provides integrated descriptions of all user requirements, including requirements from different users

- Is complete and consistent enough to estimate the amount of effort needed to develop the software
- All the processes and data aggregates connected with software are approved by the users
- The feasibility and usability are approved by the software developers

Consider our example where the user's requirement was to view the employee's information by entering his or her name. However, many employees may have the same name. Hence, in joint session it may be suggested and agreed upon to include "an on-line employee list" (name, department, and employee number) from which to select an employee.

For the above functional requirement, an index will also be necessary to speed up the retrieval of a specific employee. However, it is a technical solution and thus it cannot be considered as a part of final functional requirements. These technical specification/requirements are dealt with during subsequent phases.

Software is developed based on final functional requirements.

3.5 NEGOTIATION

The stakeholders are asked to prioritize their requirements and then discuss conflicts in priority. The "MoSCoW" rule is a widely used method for prioritizing stakeholder needs. MoSCoW is derived from the first letters of the following prioritizing criteria:

- Must have (Mo)
- Should have (S)
- Could have (Co)
- Want to have but will not have this time round (W). For most practitioners, the "W" actually stands for "Won't have".

Every stakeholder has a different viewpoint of the system. Hence, there can be conflict due to multiple viewpoints.

- *Conflicting viewpoints:* A software project has many stakeholders. These stakeholders may have different viewpoints about a system's requirement. Hence, different viewpoints may conflict with one another. For example, the marketing manager may be interested in lower cost so that the new software is easy to sell, whereas the product manager may have less concern for cost aspects.
- *Multiple viewpoints:* The requirements of the system should be explored from the different viewpoints of all its stakeholders. For example, the software sales executive may be interested in functions and features that will excite the potential market. Hence, he may insist on features such as colorful and friendly user interfaces. However, the product manager may be more interested in the performance and reliability of the software and the business managers may be interested in features required for a defined market category that can be built within cost constraints.
- The requirements engineer reconciles these conflicts through a process of negotiation.

Risks associated with each requirement are identified and analyzed. For each such specific requirement, rough estimates of development effort, project cost, and delivery time are made. Using an iterative approach, some requirements are eliminated, combined, and/or modified. SRS is generally determined by making a trade-off in some features to satisfy varied and conflicting needs of different stakeholders.

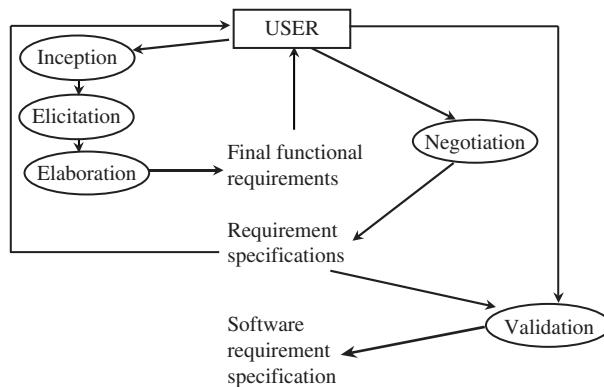


Figure 3.1 Requirement engineering process

3.6 REQUIREMENT VALIDATION

The work products (requirement specification) produced as a consequence of requirements engineering are assessed for quality during a validation step. The validation process ensures that all software requirements have been stated unambiguously, there are no inconsistencies or omissions, and that the work products conform to the standards established for the process, the project, and the product. This ensures that the right work product is built. A formal technical review is the primary mechanism for requirements validation. This is usually done in a team. The team includes members from software developers, customers, users, and other stakeholders. The final output is the SRS. The different activities of the RE process to determine SRS is shown in Figure 3.1.

3.7 STRUCTURE OF SRS

SRS is the final work product produced by the requirements engineer. It serves as the foundation for subsequent software engineering activities. It describes the function and performance requirements of software. It also lists the constraints that will affect its development. A typical structure (template) of an SRS document is given Table 3.3. The requirements of software may be described against each heading and sub-heading of this template.

SRS is a formal document. It uses natural language, graphical representations, mathematical models, usage scenarios, prototype model, or any combination of these to describe the software to be developed. There are standard templates for presenting requirement specifications in a consistent and more understandable manner. However, there is no hard and fast rule. Sometimes it may be preferable to remain flexible and use one's ingenuity to document requirement specification in a better way. The characteristics of a good SRS document are as follows:

- (i) **Structured:** The SRS document should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

Table 3.3 Typical Structure of an SRS Document

1. Introduction: It provides an overview of the system, the software being developed, and the document.
 - 1.1. Purpose of software and name of system for which it is being developed
 - 1.2. Scope of software project, its benefits, objectives, and goals (i.e. how the software is related to the business goals)
 - 1.3. Types of audience for whom the document is intended (e.g. programmers, users etc.)
 - 1.4. Brief summary about the content of SRS and how it is organized
2. Overall Description: High-level (summary) description of the software and its features are given in this section.
 - 2.1. Product Perspective:
 - 2.1.1. Context and origin of the software, about how it is initiated
 - 2.1.2. A simple block diagram of the overall system to which the software relates
 - 2.2. Product Features:
 - 2.2.1. Major features and significant functions that the software performs
 - 2.2.2. Organization of different functions/modules of the software
 - 2.3. Category/types of users who will use the software
 - 2.4. Performance Requirements
 - 2.5. Operating Environment: Minimum and recommended system requirements for running the software such as hardware platform, operating system, database system, other mandatory and optional software components etc.
 - 2.6. Design and Implementation Constraints: List the items or issues to be taken care of in development of the software. These might include the organization's policies, government regulations, hardware limitations, compatibility to other applications, language requirements etc.
 - 2.7. Security, safety, and privacy requirements
 - 2.8. User Documentation: Specify the requirements of user manuals, on-line help that will be delivered along with the software. Specify if any design document and source code are also to be delivered.
 - 2.9. Assumptions: List of assumed factors that could affect software requirements.
3. Software Features: Detailed software features are described in this section. For each software feature:
 - 3.1. State the name of the feature (in few words)
 - 3.2. Describe the feature and its priority
 - 3.3. Sequence the user actions and system responses
 - 3.4. Functional requirements: Itemize detailed functional requirements associated with each feature. Each requirement may be numbered for unique identification. It is a standard practice to use various types of specification tools, charts, diagrams, and graphs for describing the specification.
4. External Interface Requirements: Detailed description of different interfaces with the users, hardware devices, communication devices, and other software components are given here.
 - 4.1 User Interfaces: Description of each input screen and their standard buttons and functions
 - 4.2 Hardware Interfaces: Supported hardware devices and communication protocols
 - 4.3 Software Interfaces:
 - 4.3.1. Connections with other software components including databases, operating systems, tools, and libraries
 - 4.3.2. Data that will be shared across software components
 - 4.4 Communication Interfaces
 - 4.4.1. Requirements associated with any communication functions including e-mail, web browser, network server communications protocols, electronic forms, and so on
 - 4.4.2. Requirements relating to communication security, data encryption, data transfer rates, and synchronization mechanisms

5. Other Non-functional Requirements:
 - 5.1. Performance Requirements
 - 5.2. Safety and Security Requirements
 - 5.2.1. Requirements relating to security or privacy
 - 5.2.2. User identity authentication requirements
 - 5.2.3. Statutory requirements due to any external policies or regulations
 - 5.3. Software quality attributes such as adaptability, flexibility, interoperability, maintainability, portability, reliability, reusability, testability user-friendliness etc.
 - 5.4. Other requirements not covered elsewhere in the SRS
 6. Appendix:
 - 6.1. Glossary of terms
 - 6.2. List of issues
 - 6.3. Figures and diagrams
-

- (ii) **Concise:** The SRS document should be concise, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase the chances of error.
- (iii) **Conceptual integrity:** The SRS document should have conceptual integrity so that it can be easily understood by the reader without any ambiguity.
- (iv) **Viewed as black box:** The SRS document should only specify the functions that the software is required to perform and not how it should do the required functions. Hence, the SRS document views the software as a black box. For this reason, the SRS document is also called the black box specification of a system.
- (v) **Response to exceptional conditions:** The SRS document should list all conceivable exceptional conditions or events and specify software response to such conditions.
- (vi) **Verifiable:** All requirements of the software as documented in the SRS document should be verifiable. This means that it should be possible to determine whether the requirements stated in the SRS document have been met in an implementation or not.

3.8 CHARACTERISTICS OF THE RE PROCESS

The key characteristics of the “RE” process are given below.

- (i) *RE is essential for large/complex projects:* One high-level customer requirement may translate to a large number of detailed software requirements. Large projects have thousands of requirements. The requirements have interrelationships among one another. It is necessary to verify that requirements have in fact been met.
- (ii) *RE is premised on the reality of change:* The requirements change with time, and it is difficult to know all of them at the start of a project. It may be noted that new requirements will emerge as knowledge is gained during software development.
- (iii) *RE emphasizes on requirement traceability:* Sources of requirements are also the sources of requirement changes. Backward traceability to the requirement source facilitates the identification of requirement changes. For example, when requirements are based on some standards, and if the standard changes, the software needs to be analyzed with respect to the new standard.

- (iv) *RE requires a substantial allocation of budget:* Successful projects have allocated from 10 to 30% of the budget for requirement engineering activities across the project life cycle.
- (v) *RE continues throughout the project:* RE should be implemented throughout the life cycle of the project. Ideally, it begins at project inception and ends with project closeout. This aspect has led to popularity of the iterative process model.
- (vi) *RE requires skilled personnel:* RE is a technical function and is one of the most difficult and crucial tasks of software development. Hence, this job needs to be done by skilled people to ensure greater accuracy of the requirements and to minimize project risks. In addition to technical skill, the requirements engineer should have good communication and behavioral and management skill.

The main difficulty in the RE process comes from the fact that no single person has the complete picture. Hence, the requirements have to be obtained from multiple sources. The only way to understand all perspectives involved is to work effectively as a coherent team. Hence, team members' attitude, and the users' participation and communication are critical for the success of the RE process.

Case Study

Consider the case of software development for ABC University (not the real name of the university), which was discussed in the previous chapter. The university wants to develop software for its examination system. It is desirable that software should also help other functional areas of the university such as academics, finance, establishment, library etc.

Some Questions for Discussion:

- A) If you are engaged as a consultant for developing the software, whom will you include as stakeholders of this software?
- B) Probing the users' mind: For successful development of software, it is important to understand the stated as well as implied needs of the users. For example, the Controller of Examination of the university may state the objectives of the proposed software as under.
 - The software for the examination system should facilitate proper conduct of the examination.
 - The software for the examination system should facilitate timely conduct of the examination.
 - The software for the examination system should facilitate timely declaration of results.

Discuss what the user might mean by "proper conduct of examination".

- C) Based on a discussion with Controller of Examination and other senior officials, an overall idea about the software is given below. The software will operate on a client/server architecture through the university website and will perform the following functions:

| Function | Description |
|---|---|
| Student Registration: To get a list of students who will appear in the examination | Affiliated colleges will enter the student's data through the webpage |

Question Setting:

To facilitate the process of question setting

The system will have a database of all subjects. For each subject there will be two or more subject experts. The related subjects will be grouped into a number of domain areas. Each subject domain will have two to three domain experts.

For question setting, the system should help subject experts and domain experts to collaborate in their respective subject/domain areas through the Internet.

Internal Marks:

To facilitate tabulation of internal marks

The system will have a database of all teachers of different affiliated colleges. Each teacher will have a unique id.

The teacher will enter the internal marks of his students through the webpage.

External Examination:

To facilitate conduct of examination and the process of evaluation

The question papers will be uploaded in an encrypted form. The affiliated colleges will be given access to download and print the respective question paper just one hour before the examination.

All answer scripts will be codified.

Online Examination:

Conduct online examination via the Internet

The system should permit eligible students to appear in an online test in different subjects through the Internet. On completion of the test, evaluation should be done automatically and record should be maintained in the system for audit/rechecking purposes.

Discuss in a group and prioritize the requirements

D) How do you think the following category of people will be affected/benefited by the software?

- a) Controller of Examination
- b) Clerks of Examination Section
- c) Question Setters
- d) Examiners
- e) Students
- f) Teachers
- g) Affiliated Colleges

E) The software development team decides to first hold a meeting of stakeholders to understand their expectations and to get their views about the examination system.

- a) List the people who should be invited to the meeting.
- b) List a few problems that may be faced in organizing and conducting the meeting.

F) The next activity that has to be decided is to elicit information by asking the concerned persons. Make a list of people who should be interviewed. Prepare lists of questions to be discussed in each such interview.

G) Do you think getting a response through a questionnaire will be useful? Design a questionnaire to elicit information.

H) Going through official records is a good source for getting information. Hence, what records would you like to see? What problems/obstacle do you anticipate in accessing the records?

SUMMARY

Software is developed to perform certain functions. Software requirements stipulate those functions. They provide the basis for software development.

RE is a systematic approach for determining SRSs.

There are six types of SRSs: (1) Design requirement, (2) Functional requirement, (3) Implementation requirement, (4) Interface requirement, (5) Performance requirement, and (6) Physical requirement.

The RE process is accomplished through the execution of some distinct activities. Some of these activities are: Inception, Elicitation, Elaboration, Negotiation, Specification, Validation, and Management.

The inception function is concerned with understanding the situation and events that have initiated the software project. At inception, the requirements engineer identifies different stakeholders.

Elicitation is about seeking information about the software, the system, and the business. Requirement elicitation may be done through interview, questionnaire, record review, observation, and collaboration of users. Requirement elicitation provides initial user requirements.

In requirement elaboration, the initial users' requirements are reviewed in the developer's perspective to get initial technical requirements. The final functional requirements are derived by integrating initial users' requirements and initial technical requirements.

A software development project has different stakeholders. SRS is generally determined by making a trade-off in some features to satisfy varied and conflicting needs of different stakeholders.

The requirement specifications are assessed for quality during a validation step.

SRS is a formal document that describes the function and performance requirements of software. A good SRS document should be structured, concise, viewed as a black box, and verifiable. It should also list all possible exceptional conditions and should have conceptual integrity.

EXERCISES

1. Explain the need for the software RE process.
2. List various activities that are carried out in the RE process.
3. Write in brief about various types of software requirements.
4. What do you mean by stakeholders? Why is identification of stakeholders important for RE?
5. List various techniques of requirement elicitation. Compare technique of requirement elicitation through interview with that through questionnaire.
6. Describe the process of collaborative requirement gathering for determining software requirement.
7. Distinguish between "Initial Users' Requirement", "Initial Technical Requirement", and "Final Functional Requirement".
8. Explain the importance of requirement elaboration activity.
9. A requirements engineer should have good behavioral and negotiation skill. Comment.
10. Write the characteristics of a good SRS document.
11. Write the characteristics of the RE process.

SOFTWARE DESIGN APPROACHES

There are two main approaches to software analysis and design, namely, Function-Oriented Approach and Object-Oriented Approach. Both these approaches are covered in some detail in subsequent chapters of this book. The basic concepts and an overview of both these approaches are given in this chapter to serve as an introduction to subsequent chapters. It is meant to make it easier for the readers to draw similarity and distinction between these two approaches when they go through the details in subsequent chapters.

As discussed in Chapter 2, the software development process comprises many types of activities that are performed according to some plan. These activities are:

- (1) System Analysis or Requirement Analysis
- (2) Software Design
- (3) Implementation or Coding
- (4) Testing or Inspection
- (5) Maintenance or Adaptation

System Analysis or Requirement Analysis is concerned with the collection of information about the system. The information helps in understanding various aspects of operations and in determining the functional requirements of the system and the software. The output of these activities is a Software Requirement Specification (SRS) that specifies what the software should do. SRS specifies what tasks the software should perform. It specifies the inputs to the system and the outputs that are required. SRS does not specify the procedure of how the tasks should be performed. It does not specify how the inputs are to be processed to get the required output. Software Design produces a specification of how the input-output relationships specified in the SRS document are obtained. The software design document (design specification) is the blueprint of the proposed solution to the problem at hand. Hence, both system analysis as well system design are important activities for software development and are collectively referred to as ‘System Analysis and Design (SAD)’. A definite method comprising a set of standard tools and techniques is used in SAD (i.e. for both analysis as well as design) to model a problem.

4.1 DIFFERENT APPROACHES TO SAD

Commercial software is generally very complex. To develop any software, it is broken down into a few major parts. Each part is again broken down into more parts. The process is continued till each part is small enough to be manageable and designed. In this way, software is organized into a hierarchy of parts. Breaking a bigger entity into smaller parts and organizing these in a hierarchy is a convenient way of handling any complex problem. This way of organizing any system is called the top-down approach. It is widely used in engineering software systems. It is also followed in project management to create work breakdown structure (WBS) of the project by breaking the project into a number of smaller work packages. There are two popular approaches that are followed in SAD:

- (i) Function-Oriented (FO) Analysis and Design
- (ii) Object-Oriented (OO) Analysis and Design

In both the above methodologies, the system for which software is to be developed is first broken down into smaller parts using the top-down approach. However, the difference between them lies in the basis on which the system is factored.

In the FO Approach, the focus is on functions or the processes that are performed by the system. First, the few major functions of the system are identified. Then, each of these functions is again factored into a number of sub-functions. This way each sub-function may be further broken down into more number of sub-functions. Here, the emphasis is on ‘what the system does (verb)’ and not on ‘what the system consists of (noun)’.

Let us consider the example of an academic institution. An academic institution consists of various elements or entities and these entities carry out various functions. Some of the functions that are performed in an academic institution and the entities that carry out the functions are as follows:

| Functions | Entities that carry out the function |
|---------------------------------|--|
| Impart teaching on theory | Teachers, Classroom, Syllabus |
| Impart teaching on practical | Teachers, Laboratories, Syllabus |
| Conduct examination | Examination section, Classroom, Question setter, Invigilators etc. |
| Evaluate students' performances | Examination section, Examiners etc. |

The FO Approach concentrates on various functions performed in the system. In contrast, the OO Approach focuses on objects or entities that are associated in carrying out the functions of the system. Hence, the OO Approach can be considered as complementary to the FO approach.

In the OO Approach, the major entities/objects of a system are first identified. Each major object is further broken down into a number of child objects or sub-objects. They are identified by their properties. They perform the functions. An overview of both these approaches is given in the following sections.

4.2 OVERVIEW OF THE FO APPROACH

Usually some basic principles and standardized methods are followed to develop any commercial software. The FO Approach is the traditional way of developing software. It supports the concepts of structured programming.

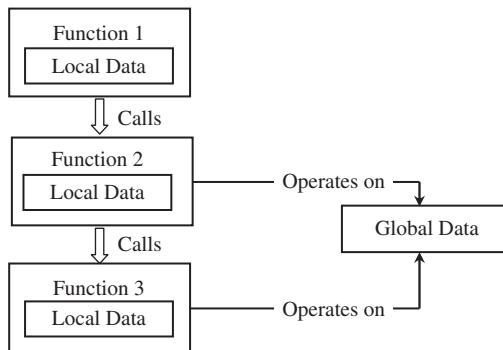


Figure 4.1 Procedural calls in structured programming

The concept of structured programming evolved in the 1970s. In structured programming, the complexity is managed by decomposing a program into a number of major functions. Then, each of these major functions is decomposed into a number of sub-functions. This process is continued till each function is small and manageable enough for coding. A structured program consists of a set of functions or modules organized in a hierarchy. A higher-level function invokes the lower-level function. The data can be local to a function and accessible only to that function. Data can also be global so that they are accessed by all functions of the program. The concept of structured programming is shown in Figure 4.1.

The methodology used in the FO Approach is quite structured and is based on the top-down approach. It is followed both in system analysis as well as in design. Hence, the methodology is called ‘Structured System Analysis and Design (SSAD)’.

The purpose of ‘*Structured Analysis*’ is to analyze users’ requirements, to carry out functional decomposition of system and to represent the same through some standard graphical tools. Structured analysis is done to determine what the system is required to do by identifying:

- functions to be performed
- data to be manipulated
- constraints on functions and data

Functional specification specifies ‘what the software is expected to do’. It provides the basis for design of software.

‘*Structured Design*’ is a class of activities that are performed to develop design specification. Design specification is the blue print for constructing the software. Software program/codes are developed by translating the design specification in a programming language. The design techniques focus primarily on the principles of functional decomposition to develop different modules of software.

4.2.1 Model and Tools

Development of any software is generally a group activity and involves a number of persons. Hence, it is important that both functional specifications as well as design specifications are described in a manner that can be easily understood by all concerned. To facilitate common understanding, some standard graphical tools, techniques and models are used to describe the specifications.

The elements of SSAD are generally composed of three essential models:

1. Environmental Model
2. Behavioral Model
3. Implementation Model

Environmental Model defines the scope of the proposed system and the boundary and the interaction between the system and the outside world. It is composed of (1) Statement of purpose of software; (2) Context diagram that depicts the main processes and the external entities of the system and (3) List of significant events that take place in the system (Event List). An event is any happening that takes place in a system. It causes some change in the system.

Behavioral Model describes the functional requirements, the internal behavior and data entities of the system. Some standard graphical tools are used for this purpose. The Data Flow Diagram (DFD), Data Dictionary, Entity Relationship Diagram (ER Diagram), Process Specification Tools and State Transition Diagram (ST Diagram) are some popular documentation tools used in the Behavioral Model. A brief description is given below:

| Tools | Purpose |
|--------------------------------|--|
| 1. DFD | Process mapping of system |
| 2. Data Dictionary | List of different data items used in the system |
| 3. ER Diagram | Data modeling |
| 4. Process Specification Tools | Procedures used in the system |
| 5. ST Diagram | Chronological events and corresponding system states |

Implementation Model describes the design specification of the software so that it can be coded in a programming language. It comprises the following parts: (1) Software architecture, (2) Data design, (3) Interface design and (4) Component design.

Software architecture is depicted by a graphical tool called the '*Structure Chart*'. The chart is used to depict the modular design of a program. It is derived from the DFD. The ER Diagram forms the basis of data design. Interface design is derived from the DFD and ST Diagram. It describes the communication of the system with other software elements, other systems and the users.

Component-level design is the detailed design of software. It is created by transforming the structural elements defined by the software architecture into procedural descriptions of software components. In this stage, the algorithms and the data structures are developed for the software modules. The algorithms are described for each module of the system in structured English/pseudocode. Software coding involves translating module specification given in pseudocode into source code in a programming language. Structured methodology for the FO Approach to software development is shown in Figure 4.2.

4.2.2 Salient Features of SSAD

SSAD methodology follows certain basic principles and guidelines. The salient features are given below.

Functional Decomposition: The FO design strategy relies on decomposing the system into a set of interacting functions. Each subroutine performs some function of a program. This permits subroutines to be developed and tested separately and then integrated into the main program.

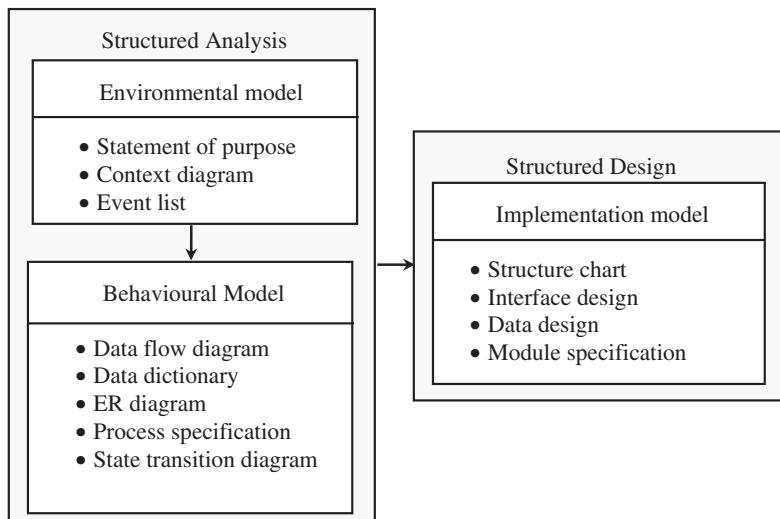


Figure 4.2 Function oriented approach to software development

Top-Down Hierarchical Structure: In structured methodology, the software is structured as a top-down hierarchy of modules. A module is a group of instructions, subprograms or subroutines. The top-down structure of these modules is developed according to some design rules and guidelines.

Cohesiveness: Cohesion refers to the degree to which a module's instructions are functionally related. In highly cohesive software, each lower-level module should accomplish a single specific function. This makes the modules reusable in future programs. The lower-level modules are grouped together based on similarity of purpose and linked to their higher-level module. Modules should be highly cohesive. Programs that are implemented with highly cohesive modules are easier to understand, modify and maintain.

Coupling: Coupling refers to the level of dependency that exists between modules. Modules should have minimum dependence on one another. This minimizes the effect that future changes in one module will have on other modules. Modules should be loosely coupled.

Separation of Data from Program: In structured analysis techniques, data requirements and functional requirements are usually organized separately. Similarly, in structured design data modules and processing modules are also kept separate. Modules that access the same data may be widely dispersed in the design. The documentation tools used for program modules are also different from those used for data structures.

Testing at Multiple Levels: Testing is performed at multiple levels. Usually it is done at three levels, namely, (1) Unit level, (2) Integration level and (3) System level. Errors detected at higher levels of testing require unit-level changes and re-test at all lower levels. Responsibilities for testing at each level are usually entrusted to different groups.

4.3 OVERVIEW OF THE OO APPROACH

The world is made up of entities or objects. Hence, if we analyze and design a system in terms of objects and how the objects interact with one another, then we are closer to the real-world modeling. The concepts underlying object orientation as a programming language evolved about 20 years back. It is only in the last few years that Object-Oriented Analysis and Design (OOAD) method has become quite popular and is now being widely used for development of commercial software. This methodology is more suitable for a software project that requires complex algorithm/processing logic.

The OO Approach uses OO programming concepts. This approach uses data structures combined with relevant functions (method) to create reusable objects. This improves the maintainability of the software to a large extent.

An object is invoked by this method. Invocation of a method may change either the properties of the object or invoke some other object. By this the messages are passed between objects (See Figure 4.3). We will discuss more about objects and messages in Chapter 7.

The OO Approach focuses on identifying the real-world objects associated with the problem. Models are built to represent the real-world objects. The problem solution involves implementing the models. Data and behavior of a real-world object are treated as an interrelated whole. The software solution maps more closely to the real-world problem.

In the OO Approach, a software system is viewed as a set of interacting objects. The characteristics of the OO Approach are:

- Objects are an abstraction of the real world. The objects can be considered as interacting entities that constitute a system.
- Objects are identified by their properties. The properties of objects can change by specific events according to set procedure (code). The code (program) that is applicable to an object is contained in that object.
- System functionality is expressed in terms of operations or services associated with each object.
- Shared data areas are eliminated. Objects communicate by calling on services offered by other objects rather than sharing variables.
- Objects may be distributed and may execute either sequentially or in parallel.

The OO methodology for a problem is a step-by-step procedure of arriving at systems engineering, analysis and design. Like SSAD methodology, the OO methodology also uses some graphical tools to represent specification. However, the tools used in OOAD are different from what are used in SSAD.

Unified Modeling Language (UML) is a documentation tool used in OOAD methodology for specifying, constructing, visualizing and documenting the artifacts of a software-intensive system. UML has

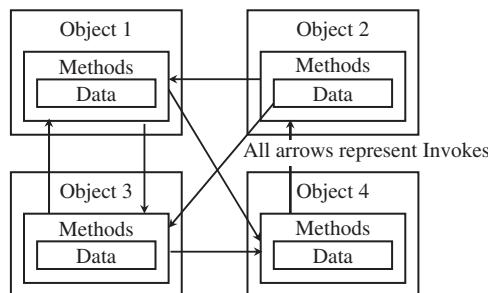


Figure 4.3 Messages passing within objects

evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobsen and others at Rational Corporation in 1997. At present, the UML has almost become an industry standard documentation tool. UML comprises a number of documentation tools as listed below:

1. Use Case diagram
2. Static Structure diagram (Class diagram and Object diagrams)
3. State diagram and Activity diagrams
4. Interaction diagrams (Sequence diagram and Collaboration diagram)
5. Implementation diagrams (Component diagram and Deployment diagrams).

A number of Computer Aided Software Engineering (CASE) tools are now available to assist in OOAD (e.g. for drawing UML diagrams). Rational Rose is one such popular software.

The OOAD comprises the Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD). However, in the OO approach, it is difficult to make a boundary between these two sets of activities. For example, the Class diagram and Object diagram are included in the OOA. However, these are also useful for OOD. It may be seen in Figure 4.4 that there is overlapping of both these phases.

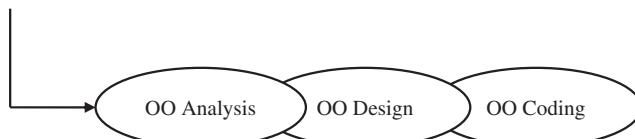


Figure 4.4 Object oriented approach of analysis and design

In fact, there is overlapping of the OOD phase with OO coding and implementation as well.

4.3.1 Object-oriented Analysis

Object-oriented analysis (OOA) or specifies WHAT the system is to do in terms of the real-world objects and not in terms of separate functions and data. The objects are present in the problem domain. OOA is performed in the following steps:

1. Define User View of Requirements
2. Identify Analysis Objects and their Characteristics
3. Determine Object Dynamics
4. Determine Object Interactions and Relationship

In OOA, requirements are expressed via the following diagrams:

| Name of diagram | Purpose |
|-----------------------|--|
| Use Case diagram | It describes how the users (actors) interact with processes (use cases). |
| Class diagram | It is used to refine the Use Case diagram and define a detailed design of the system. |
| State diagram | It represents the different states that the objects in the system undergo during their life cycle. |
| Activity diagram | It describes the process flow of the system. |
| Sequence diagram | It represents the interaction between different objects in the system. |
| Collaboration diagram | It groups together the interactions between different objects. |

Solution space maps naturally to the problem space. The requirements are specified in user-recognizable terms.

4.3.2 Object-oriented Design

During the OO analysis the main focus is on ‘what needs to be done’. The resulting classes from OO analysis can serve as the framework for the design. Designing any OO system poses the fundamental questions:

- What objects do we need?
- What behaviors are required?
- How do we distribute behavior over the set of objects?

In object-oriented design (OOD), the classes, attributes, methods and associations identified during the OO analysis phase are designed for implementation. First, the objects (classes, fields) and their relationships (inheritance, association, aggregation) are designed by using some model. This is the Model-Centric perspective of design. The objects are expressed as some data-type in an implementation (programming) language. This is the Language-Centric perspective of design. These two perspectives focus predominately on the static or structural aspects of design. Hence, this is called ‘*Static Modeling*’.

Objects also have some dynamic aspects. They interact with each other by sending messages to one another. They also perform some action. Their dynamic aspect is called object behavior. Defining the behavior (methods) of objects is called ‘*Dynamic Modeling*’’. This is the ‘Responsibility-Centric’ perspective of design.

In OOD, the structural aspects are laid out first by specifying the classes and their relations. In the next step, the behavior or methods are distributed over different laid-out classes. All the diagrams discussed in OOA besides Use Case diagram contribute to OOD. Therefore, the two types of design models that are part of OOD are as follows:

1. The static models describe the static structure of the system in terms of the system object classes and their relationships.
2. The dynamic models describe the dynamic structure of the system. Dynamic models show the time-dependent behavior of the objects.

Analysis objects are refined during design. Resulting design objects serve as the system’s design modules.

In OOD, the design specifications are expressed via the following diagrams:

| Name of diagram | Purpose |
|--------------------|---|
| Component diagram | It represents the high-level parts that make up the system. |
| Deployment diagram | It captures the configuration of the runtime elements of the application. |
| Package | It holds model elements such as classes, state machines and Use Cases. |
| Subsystem | It is a special package that represents a portion of a system with a crisp interface that can be implemented as a distinct component. |

4.3.3 Object-oriented Testing

In the OO methodology, testing can be conducted in a non-hierarchical fashion. Testing can be done (1) within an object; (2) between clusters of objects that work together and (3) for the entire system. The scope of testing must cover system and network architecture. System/acceptance testing techniques are basically the same as in a traditional development project.

4.3.4 Object-oriented Maintenance

Maintenance can often be performed without significantly changing the structure of an existing application. Modification can be done to a group of objects without affecting other objects. Additional objects may be added without needing to know the implementation details of existing objects.

Object orientation has several advantages such as lower development effort, lower development time and better maintainability. It is based upon objects that constitute a system. Hence, it is more close to the real world. The OO approach is supported by some very useful design principles like maintainability, extensibility, reusability, interoperability and quality. Its popularity is increasing.

4.4 COMPARISON OF OOAD WITH SSAD

In both FO and OO methodologies, the same basic problem-solving steps are used. These basic steps are:

- Understand the problem
- Specify requirements (WHAT)
- Design the solution (HOW)
- Write the code
- Test and deploy

Similar types of tasks such as elicitation of requirements from users, documenting the requirements, identifying a designing software module, acceptance tests etc are performed in both the methodologies. Many traditional project management issues such as planning, estimating, monitoring and control, communicating also remain the same.

However, there is a significant difference between the two approaches. The principles of design and programming of these two methodologies are different as given below.

Different View of the System: In the FO design the software is developed by designing functions. Here, the focus is on 'verbs' (e.g. Update, Get etc.). In the FO technique, the number of functions are grouped together to constitute a higher-level function. However, in the OO design, the software is developed by designing objects. In this approach the focus is on 'nouns' (e.g. Employee, Department etc.) instead of verbs. A group of lower-level (child) objects can be contained in a higher-level (parent) object.

Integration of Data and Method: In the FO Approach data requirements and functional requirements are usually analyzed and designed separately. The data are shared among applications. However, the OO Approach treats data as an integral part of system design. The function in OO terminology is called method. The objects contain both data and method as its integral part. Unlike the FO approach, the data are not shared among applications. Instead, the functions (called methods in OO terminology) are grouped together with the data they operate on as objects. Data can be accessed only through the methods specified in the object. In the OO approach, the design strategy is based on information hiding.

Different Documentation Tools: Diagrams and tools used by the development team to produce a solution in the FO Approach and OO Approach are different. The FO Approach follows SSAD methodologies. It uses DFD, Data Dictionary, ER Diagram, STD etc. for structured analysis. A Structured Chart is used for structured design. The OO methodology uses a different set of tools. A standardized set of tools called UML is used in the OO methodology. The UML comprises Use Case diagram, Class diagram, Object diagram, State diagram, Activity diagram, Sequence diagram, Collaboration diagram, Component diagram and Deployment diagram. The first three diagrams are used for OOA and rest are used for OOD.

Real-World Focus: The functional approach starts with analyzing a real-world problem. The real-world focus is gradually reduced as requirements are transformed into a series of DFDs during structured analysis. The real-world focus is further lost during the design stage, in the creation of structure charts and module specification. However, in the OO Approach focus remains on the real-world objects throughout the development cycle. Natural correspondence between problem domain and solution domain results in better communications between developers and customers throughout a project life cycle.

Project Life Cycle: In the FO approach, the three development steps analysis, design and coding are done serially in the following sequence. However, there is an overlap between each of the three development steps in the OO Approach. The program objects once developed can be reused. The objects are not required to be retested and redesigned every time changes are made to the data layout. Hence, in the OO approach different project life cycles may be used. Incremental/iterative life cycle is the more popular life-cycle model in the OO approach.

Program Coding: In a functional approach a major portion of the software lifetime is spent on implementing the design. However, in the OO Approach more emphasis is given on design instead of implementation. Program coding in the OO approach is usually done in an OO language such as C++, Java, Visual Basic etc. These languages offer constructs that allow direct implementation of objects. Therefore, comparably, a much smaller fraction of time is spent on implementation. Hence, more time can be devoted to design.

Component Reuse: In the FO approach programmers usually build software components from scratch even if code libraries are available. Often they feel it is necessary to know how a component works. However, component reuse is an important feature of the OO approach. Objects are appropriate reusable components. Designs can even be developed using objects that have been created in previous designs. The objects support reusability across projects and the enterprise. Library objects may be used and modified without needing access to the details of their implementation. Hence, the programmers are freed from having to know the internals of objects. Component reuse has many advantages.

- Design, programming and validation costs are reduced.
- It uses standard objects and reduces the risks involved in software development.
- Productivity of software development is increased.

Software Maintenance: Objects may be understood and modified as stand-alone entities. Changing the implementation of an object or adding a service does not affect other system objects. The effect of changes made in an object is localized to that object. OO software is very easy to maintain.

From the above comparison it may be argued that the OO approach is much better than the FO approach. Hence, most new and complex systems are now being developed by using the OO approach. However, the traditional FO approach also has its importance. Many older systems in use today were designed and documented using traditional methods. Many business applications involve large volume of data but do not require any complex data processing. The FO Approach is suited for such data-intensive projects. The OO Approach is more suitable for applications that involve complex logic/algorithms.

SUMMARY

The FO Approach and OO Approach are two main approaches to software engineering.

The functions or processes performed in the system are the main focus in the FO approach. This approach supports the concepts of structured programming. A structured program consists of a set of functions or modules organized in a hierarchy. The methodology used in the FO approach is quite structured and is called 'SSAD'.

The structured analysis is expressed by the Environmental Model and Behavioral Model. The Environmental Model defines the scope, the boundary and the interaction between the system and its surroundings. The Behavioral Model describes the functional requirements, the internal behavior and data entities of the system. DFD, Data Dictionary, ER Diagram, Process Specification Tools and STD are some popular documentation tools used in the Behavioural Model.

The Implementation Model describes the design specification of the software. It comprises (1) Software architecture, (2) Data design, (3) Interface design and (4) Component design. The software architecture is depicted by a 'Structure Chart'. The processing logic (algorithms) of modules is described by structured English/pseudocode.

In the OO Approach the focus is on objects or entities of a system. The methodology used in the OO approach is called OOAD.

UML is a standard documentation tool used in OOAD methodology. It comprises Use Case diagram, Class diagram, Object diagrams, State diagram, Activity diagrams, Sequence diagram, Collaboration diagram, Component diagram and Deployment diagrams. The first three diagrams are used in OOA and the rest are used in OOD. However, there is some overlap of scope between OOA and OOD.

The FO Approach and OO Approach differ from each other in terms of different views of the system, integration of data and method, different documentation tools, software development life cycle, program coding, component reuse and software maintenance. The FO approach is suited for data-intensive projects whereas the OO approach is more suitable for applications that involve complex algorithm/processing logic.

EXERCISES

1. What do you understand by top-down decomposition in the context of software engineering?
2. What are the main features of structured programming?
3. Write down the differences between structured analysis and structured design.
4. Write the main content of Environmental Model, Behavioral Model and Implementation Model.
5. List some tools used in structured analysis.
6. Explain the basic principles and guidelines of SSAD.
7. Distinguish between coupling and cohesiveness.

8. Distinguish between function and object. List different functions and objects of the academic system of an educational institution.
9. Write the main characteristics of the OO Approach to software development.
10. Write the purpose of OO Analysis (OOA).
11. Write down the differences between OO Analysis and OO design. Is there any overlapping among them? If yes, explain it showing a typical scenario.
12. What is UML? List some documentation tools used in UML.
13. List various tools/diagrams used in OOA.
14. Distinguish between Static models and Dynamic models used in OOD.
15. List various diagrams that are used to express OO design specifications.
16. Explain the meaning of the following phrases in the context of Object Technology.
 - a. Integration of Data and Method
 - b. Real-World Focus
 - c. Component Reuse
17. Make a comparison between OOAD and SSAD.

STRUCTURED ANALYSIS

The function-oriented approach uses a structured methodology called “Structured Analysis and Structured Design (SASD)” to produce any commercial software. Structured analysis is a part of SASD methodology. Its purpose is to analyze users’ requirement, carry out functional decomposition of a system and represent the same through some standard graphical tools. This chapter describes different tools and techniques used in structured analysis.

Quality of software is determined by three main parameters:

- Performance: Extent to which the software meets the requirements of users
- Control: How secure is the software against human error, machine mal-function and intentional sabotage
- Modifiability: Ease with which the software permits itself to be modified to suit the changing needs of users

Quality is not achieved by chance. To ensure quality, a specific set of standard tools and techniques is generally used to produce any commercial software.

The function-oriented approach is the early and the traditional approach to the development of software. A structured methodology called “Structured Analysis and Structured Design (SASD)” is used in the function-oriented approach.

SASD is a disciplined approach to computer system design. It consists of two distinct activities.

- i) Structured Analysis (SA) and ii) Structured Design (SD)

SASD methodology was developed in the late 1970s by DeMarco, Yourdon, and Constantine. It is based on structured programming. The IBM incorporated SASD into their development cycle in the early 1980s. In 1989, Yourdon publicized his book “Modern Structured Analysis”. Subsequently the SASD methodology was enhanced to improve its ability to represent real-time systems.

The emphasis in SASD is on easing the complexities of the software development project by following the principle of “divide and conquer”. The steps followed are to divide the system into smaller components and understand what each component does and how the components are related. This reduces complexities and the time needed for software development.

5.1 INTRODUCTION TO STRUCTURED ANALYSIS

Structured Analysis (SA) is a methodology for determining and documenting the requirements for a system. The purpose of SA is to gain understanding about the system and to capture the detailed structure and requirements of the system as viewed by the users.

The system analysts explore various sources and use various techniques to gather information. Generally, the development of any commercial software is a group activity and involves a number of people. Hence, in order to facilitate all team members to contribute to the software development process, the knowledge gained during system analysis needs to be properly documented and shared. To ensure development of correct software, it is important that the requirements are documented in a form so that it fulfils two purposes:

1. The users can understand and validate that the software requirements are correct and completely specified.
2. The software designer interprets the requirements in exactly the same way as the users, i.e. there is no ambiguity in the interpretation of requirements.

For this some standard graphical tools and techniques are used. Data Flow Diagram (DFD), Data Dictionary, Decision Table, Decision Trees, ER Diagram, and State Transition Diagram (STD) are some such tools that are widely used in SA. These tools and techniques facilitate easy understanding and documentation.

The SA in SASD produces a “*Structured Requirements Specification*” comprising the Environmental Model and the Behavioral Model.

The Environmental Model defines the boundary and interaction between the system and the outside world. It is composed of Statement of Purpose, Context Diagram, and Event List.

- **Statement of Purpose** is a clear and concise textual description of the purpose for the system.
- **Context Diagram** is a graph that highlights the boundary between the system and the outside world. It highlights the people, organizations, and outside systems that interact with the system under development.
- **Event List** is a list of the events that create some response in the system.

The Behavioral Model consists of DFD, Process Specification, Data Dictionary, Entity Relationship Diagram (ERD), and STD:

- **DFD** is graphical technique to depict the movement of data (information) between processes in the system. It helps in understanding various processes that are performed in the system and the interaction between these processes and the external entities.
- **Process Specification** is a description of each process/function in the DFD. It describes the purpose of the process, the input, and the output. It is usually depicted through Structured Statements, Decision Table, and Decision Tree.
- **ERD** is a graphical model of the relationships between the data entities in the system. It gives further details about the data stores that were presented in the DFD.
- **Data Dictionary** is a repository containing details about each data item that appears on the DFD.
- **STD** is a graphical technique to highlight the time-dependent behavior of the system. It describes what happens when an event occurs.

The above deliverables of SA are transformed in the design stage into a suitable form so that they can be implemented in a programming language.

5.2 DATA FLOW DIAGRAM

In the function-oriented approach, a system is viewed as consisting of processes. A process receives input data either from an external entity or some other process. The process causes some transformation to data. The output of a process can either be input to some other process or final information to an external entity.

Data Flow Diagram (DFD) is a graphical description of a system. It shows the system's processes, the various input data that originate from external sources (entity), and how these input data pass through these processes and are transformed into useful information. DFD uses four major components to depict a system. These components are

1. External entity
2. Process
3. Data flow
4. Data store

External entities are objects that interact with the system. A system comes into action based on input signal or data it receives from an external source (entity). The system generates some output or information. The recipient is again some entity that is external to the system. Hence, data flow from an external entity to the system and from the system to an external entity. An external entity is a source or sink of data flows. It exists in the system environment.

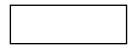
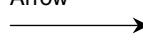
A process receives input data (data flows in) and causes some transformation to this data. A system may consist of a number of processes so that the output of one process may be the input to the other till the final output is obtained for some external entity.

A data element that goes into a process as input or out of a process as output is called data flow. It is a data structure in motion.

Data is a useful resource. Data elements stored in a data file are called data store. A data store may provide input data to a process. The output data from a process may also be stored as data store for future needs. A data store is a data structure at rest.

Some graphical symbols are used in DFD to depict the components of a system and its environment. These symbols are given in Table 5.1.

Table 5.1 Components of DFD

| System component | Description | Symbol |
|------------------|---|---|
| External entity | External source of input data or recipient of output | Square or Rectangle  |
| Process | Function or procedure that causes some transformation to data | Circle or Ellipse  or  |
| Data flow | Data structure in motion. Input or output of a process | Arrow  |
| Data store | Data file or some form of record stored for future processing | Open-end box  |

A DFD may consist of several external entities, processes, and data flows and data stores. In a DFD a process is depicted by a circle or ellipse. An arrow identifies data flow. A data flow originates from some source and after being transformed by one or more processes is received by some destinations. A data flow may be considered as data in motion. The sources and destinations of data flows are called external entities. An external entity is denoted by a rectangle. An open-end box represents a data store. A data store may be considered as data at rest. It is a repository of data such as data file or some form of record keeping. Since processes in DFD are represented by circles/ellipses that look like bubbles, DFD is also called as “bubble chart”.

A DFD is simple to draw and depicts the basic components of a system. It can be drawn in increasing level of detail, starting with a summary high-level view and proceeding to more detailed lower-level views. Thus, it supports modular, structured, and top-down approach to design the information system. The highest level of DFD (level-0 DFD) depicts the summary of a system. It is also called the Context Diagram.

Example 1 (Examination System)

Consider the case of an examination system for processing the marks of students. The system after processing the marks of students generates a report card for each individual student. The system also generates a consolidated report and sends it to the Director-of-examination. Here, the student and the director-of-examination are the external entities. The summary of the above system is shown in Figure 5.1. It is called level-0 DFD or the Context Diagram.

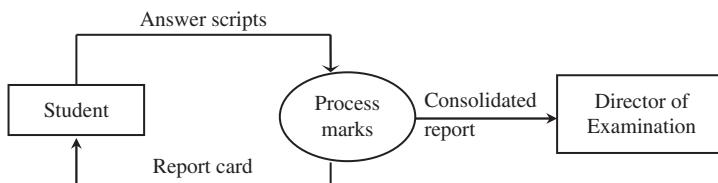


Figure 5.1 Context diagram of examination system

How actually the marks are processed is not clear from the above Context Diagram. The same can be depicted in greater detail by level-1 DFD as shown in Figure 5.2. Level-1 DFD contains four constituent processes of the “Marks Processing” process. These four processes are “1 check answer script”, “2 prepare mark sheet”, “3 prepare report card & update data file”, and “4 prepare consolidated report”. The process “check answer script” performs the evaluation of answer scripts. The next process is preparation of marks sheets. The mark sheets are processed to generate report cards of individual students and to update the student files. The fourth process is preparation of consolidated report for Director-of-examination. All these processes are part of “Process Marks”, which is shown earlier in the Context Diagram.

Example 2 (Order Processing System)

Consider another example of “Order Processing System”. The Customers send in orders to the system. The system acts on these orders to dispatch goods and the invoices to the customers. The system determines the demand of goods by consolidating various orders received from customers. It checks the stock availability, and conveys additional requirement of goods to the production shop. The level-1 DFD of this system is shown in Figure 5.3.

The *Customer* and the *Production shop* are two major external entities of this system.

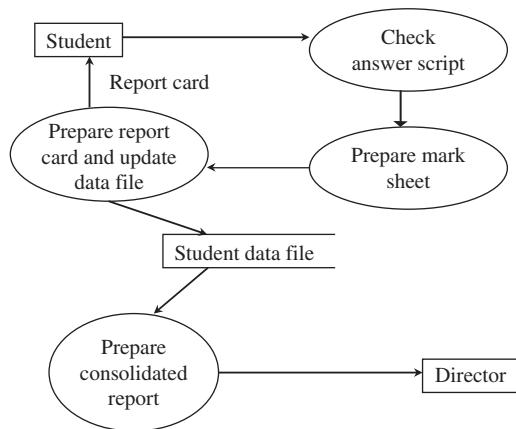


Figure 5.2 Level-1 DFD of examination system

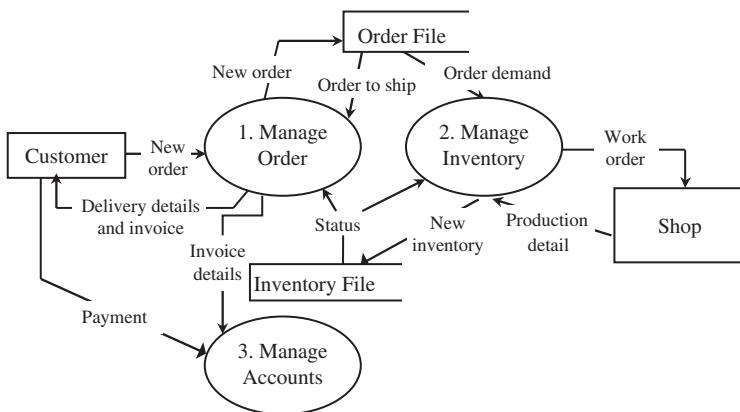


Figure 5.3 Level-1 DFD of order processing system

The system performs three major processes, namely, (1) manage orders, (2) manage inventory, and (3) manage accounts. The *Manage Order* process keeps track of orders received from the customer. This process ensures that the goods are shipped against orders when they are due. The *Manage Inventory* process is responsible for ensuring that inventory is available at the requested ship date. In order to accomplish this, the *manage inventory* process examines the *order demand* and compares it with the status of inventory availability. If there are not enough inventories to satisfy the demand, a *work order* is sent to the production shop. When the production shop sends the ordered items, the *new inventory* is entered into the *Inventory file* data store.

The *manage order* process sends an invoice for customer payment. The *manage accounts* process ensures receipt of payment against goods delivered to customers. When the order is shipped to the customer, the detail is sent to the *manage account* process and the payment received from various customers are tallied against invoice details.

The system has two data stores. The information about an order-to-ship is contained in the *Order file* data store, and the status of available inventory is contained in the *Inventory file* data store.

Example 3

Let us consider another example of a bank system. The customer of bank (account holder) deposits money and gets the payment receipt. The Context Diagram is shown in Figure 5.4.

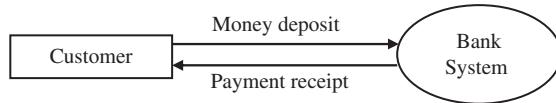


Figure 5.4 Context diagram of bank system

Level-1 DFD of the above Context Diagram that depicts the process in some greater detail is shown in Figure 5.5.

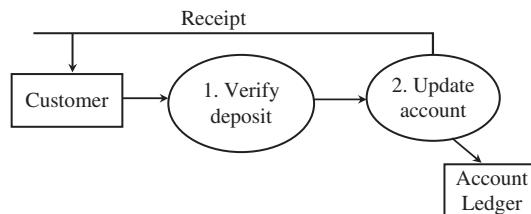


Figure 5.5 Level-1 DFD of bank system

Each process of level-1 DFD can be shown in greater detail in level-2 DFD. For example the process-2 (update account) can be exploded into a number of sub-processes. The level-2 DFD of process-2 (update account) is shown in Figure 5.6.

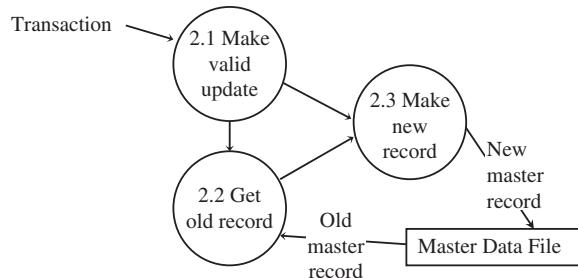


Figure 5.6 Level-2 DFD of process-2 (update account) of bank system

A higher-level DFD forms a parent-child relationship with its lower-level DFDs. The process-2 of DFD level-1 has three sub-processes in DFD level-2. Note that these three processes are numbered hierarchically as 2.1, 2.2, and 2.3. This form of numbering scheme facilitates people to link processes of a lower-level DFD to its higher level. Numbering of processes in hierarchical manner is called leveling of DFDs. The process of leveling of DFDs is illustrated in Figure 5.7.

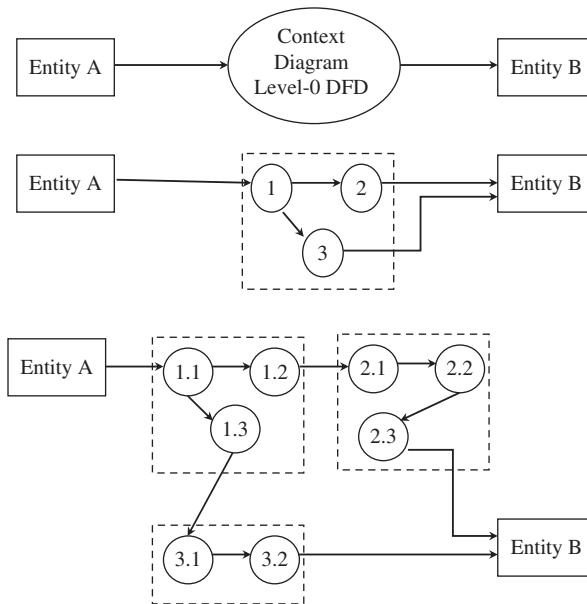


Figure 5.7 Leveling of data flow diagram

5.2.1 Rules for Drawing DFD

The following seven rules govern construction of DFD:

- (i) The squares/rectangles, circles/ellipses, arrows, and open-end boxes representing entities, processes, data flows, and data stores, respectively, must be clearly labeled.
- (ii) Leveling should be done as per convention so that parent-child relationships between DFDs are clearly depicted.
- (iii) No two data flows, data stores, entities, and processes can have the same name/label.
- (iv) Meaningful names should be chosen for labeling data flows, processes, and data stores. Use strong verbs followed by nouns.
- (v) Arrows representing data flows should not cross each other.
- (vi) The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD. Data flows in DFDs must be balanced, i.e. all data flows on the decomposed lower-level diagram must reflect flows in its parent diagram.
- (vii) For ensuring neatness and easy understanding of process, control information such as record counts, passwords, error routines, and validation requirements may not be shown in a data flow diagram.

5.2.2 Physical and Logical DFD

DFD are of two types, i.e. Physical DFD and Logical DFD. The DFDs considered so far are examples of Logical DFD. A Logical DFD specifies various logical processes performed on data. It does not specify information such as: who does the processing, where the processing is done, or on which physical device data are stored. The above facts are specified by Physical DFD.

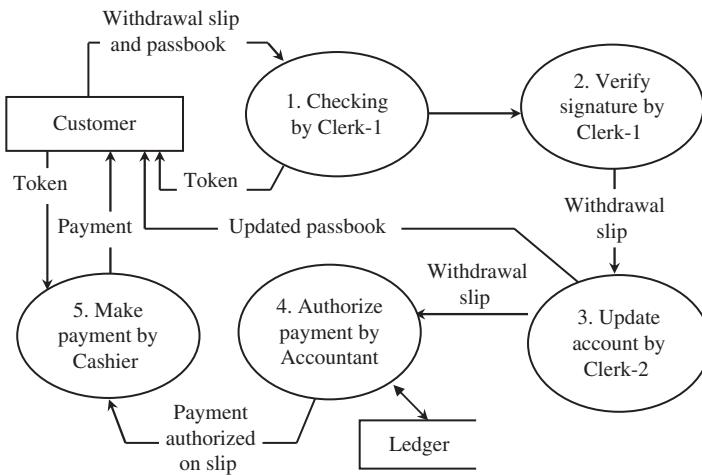


Figure 5.8 Physical DFD of withdrawal system of bank

An example of physical DFD is shown in Figure 5.8. It shows the process for withdrawal of money from a bank by an account holder against a withdrawal slip. The DFD shows that the withdrawal slip is processed at four different desks (two clerks, one accountant, and one cashier).

The Physical DFD is very helpful for understanding the existing system. It is often the starting point for identifying the transform processes and drawing the Logical DFD of the existing system. The Logical DFD of the existing system is refined and modified for improvement of processes of the system. The physical requirements of modified processes are again specified by Physical DFD. Hence, the DFD is not only an effective tool for system documentation but it can also be used to make improvement in processes of a system.

DFD is a very popular tool because it is simple and it depicts the data flows and data stores as well as the processes that transform data. Another strong advantage is the balancing feature that facilitates error detection in DFD documentation. For example, if a parent data flow diagram shows three inputs and two outputs, the leveled child diagrams taken together must have three inputs and two outputs. If there is an imbalance between parent and child data flow diagrams, it indicates that there is an error either in the parent or the child diagram. The leveling feature of DFD helps in SD of the information system.

5.3 PROCESS SPECIFICATION

A process specification describes the purpose of processes depicted in the DFD, the inputs to the process, and the output. It describes what the process should do and not how it should do it. However, if the process relates to some decision making then the decision rules used in the process may be described. The decision rules are better depicted by documentation tools such as Decision Table and Decision Tree.

5.3.1 Decision Tables

The Decision Table is a chart that defines a logical procedure by means of a set of conditions and related actions. It consists of four sections:

- Condition stub
- Condition entry

- (iii) Action stub
- (iv) Action entry

In addition to the above, the Decision Table also contains space for showing process identification, name of the process, title/description, special comments etc.

| | |
|---------------------|-----------------|
| Process ID: | _____ |
| Process Name: | _____ |
| Title/-Description: | _____ |
| CONDITION STUB | CONDITION ENTRY |
| ACTION STUB | ACTION ENTRY |

Figure 5.9 Different sections of a decision table

The *Condition Stub* displays all the necessary tests or conditions. Generally, these conditions are of a type that is either true (yes) or false (no). The condition stub always appears in the upper left-hand corner of the Decision Table. Each condition is numbered to allow easy identification. Thus, the condition stub is a list of all the necessary tests in a Decision Table.

In the lower left-hand corner of the Decision Table is for the *Action Stub* where all actions desired in a given module are listed. Actions are also serially numbered for identification purposes. Thus, Action Stub is a list of all the actions or decisions involved in a process.

The upper right corner provides space for the *Condition Entry* for all possible permutations of “true” and “false” responses related to the Condition Stub. The “true” or “false” possibilities are arranged in a vertical column called rules. Rules are numbered 1, 2, 3, and so on. Thus, Condition Entry is a list of all “true/false” permutations in a Decision Table.

The lower right corner holds the *Action Entry*, where cross marks (X) indicate whether an action should occur as a consequence of a particular set of true/false entries. The different sections of a Decision Table are shown in Figure 5.9.

The maximum number of possibilities or rules in a Decision Table can be determined by the following mathematical formula:

$$\text{Number of rules} = 2^N \quad \text{where "N" represents the number of conditions.}$$

Thus, a Decision Table having four conditions has 16 ($2^4 = 16$) rules. Similarly, a Decision Table having six conditions has 64 rules and one having eight conditions may have 256 rules. If a large number of conditions exist (four conditions result in 16 condition entries, six conditions in 64), Tecision Tables can become unwieldy. To avoid lengthy Decision Tables, analysts must remove redundancies and at the same time take precautions not to overlook anything that is relevant. In some cases, two or more rules may be combined to reduce redundancy. To indicate redundancy, we put a dash (-) in the condition entry to show that this condition stub is irrelevant and can be ignored. A Decision Table showing the discount policy of a book publisher or a wholesaler may look like the one given in Table 5.2.

Table 5.2 Decision Table showing the discount policy of a publisher

| | 1 | 2 | 3 | 4 | 5 |
|-------------------------------|---|---|---|---|---|
| Customer is book store? | T | T | F | F | F |
| Order size 5 copies or more? | T | F | T | T | F |
| Order size 20 copies or more? | - | - | T | F | - |
| Allow 25% discount | X | | | | |
| Allow 15% discount | | | X | | |
| Allow 10% discount | | | | X | |
| No discount allowed | | X | | | X |

A Decision Table can be of two forms:

- (1) Limited Entry form
- (2) Extended Entry form

In a limited condition entry, the response is either a “yes” or “no” or the condition is not relevant. This is indicated by a “dash”. However, in an extended condition entry, the response is either descriptive or in quantified form. Similarly, a limited action entry comprises one or more crosses in each column to indicate actions, otherwise a blank that indicates no action. However, in an extended action entry, the actions are written in words.

Limited Entry: The Decision Table given earlier at Table 5.2 is an example of a limited entry table. In case of limited entry, the actions are based on conditions that are either true (yes) or false (no). In certain situations a condition may not be affecting an action. Such situations may be represented by dash (-).

Another example of limited entry Decision Table is given in Table 5.3.

The Decision Table is for the check printing process. Note that in the said Decision Table, if condition 1 (End of sorted invoice file?) is true (yes) then it results in action 7, which is “end of module”, irrespective of any entry for the other three conditions. Hence, entries for those conditions are shown by “dash” (-). This way the number of condition entries has reduced from a possible 16 entries to only five entries.

Extended Entry: To illustrate the use of Extended Entry Decision Table, let us consider the criteria that are followed to identify the employees of an organization for awarding promotions.

Promotion is awarded to an employee based on number of years served in a particular grade, his qualification, and performance rating of preceding 2 years. Let us assume that the following criteria are considered for awarding promotion in the said organization.

A person can be considered for promotion if

1. Number of years in present grade is more than 5 years irrespective of other conditions.
2. Qualification is more than adequate and performance ratings in preceding 2 years are “very good” or above.
3. Qualification is adequate and performance rating in any of the preceding 2 years is “excellent” and other is “very good” or above.
4. Qualification is slightly less than adequate and performance ratings in both the preceding 2 years are excellent.
5. In rest of the cases, the employee is not to be promoted.

The Decision Table depicting the above rules is shown in Table 5.4.

Table 5.3 A limited entry Decision Table for a check printing process

TITLE: Check entry

Date: 16/07/09

Author:

System: Accounts payable System

Comments: Two files are to be read until the end of the file

| | 1 | 2 | 3 | 4 | 5 |
|--|---|---|---|---|---|
| 1. End of the sorted Invoice file? | N | N | N | N | Y |
| 2. End of Supplier master file? | N | N | N | Y | - |
| 3. Is Supplier code in Supplier master file greater than that in the Invoice file? | N | N | Y | - | - |
| 4. Does Supplier code match? | Y | N | - | - | - |
| 1. Read a record from Supplier master file | | X | | | |
| 2. Read a record from Invoice file | X | | X | X | |
| 3. Print date | X | | | | |
| 4. Print Supplier name and address | X | | | | |
| 5. Print invoice amount | X | | | | |
| 6. Display "Supplier code not present" | | X | X | | |
| 7. End of module | | | | | X |

Table 5.4 An Extended Entry decision table

Process ID: _____

System: Human Resource Management

TITLE: Short listing employees for promotion

C = Slightly less than adequate

D = Not adequate

Performance: A = Excellent

B = Very good

C = Good

D = Satisfactory

E = Poor

Note that in the said table one of the conditions “Is number of years in present grade greater than 5?” takes two values, i.e. either “yes” or “no”. Hence, the Decision Table may be considered as an example of *mixed entry*.

In the above table we have been able to reduce the number of condition entries to a manageable size by removing redundancies. However, in many cases it may not be possible to do so. In such cases the decision rules are shown in an open-ended Decision Table.

Open-ended: An open-ended Decision Table allows an action entry that point to another Decision Table. Hence, instead of one big Decision Table, it is preferable to decompose the same into a series of smaller ones. A limited-entry Decision Table can be broken down into smaller open-ended linked Decision Tables. This is shown in Table 5.5a and Table 5.5b.

Table 5.5a Illustration of an open-ended Decision Table

| | 1 | 2 | 3 |
|--------------------------------|---|---|---|
| 1. Customer is a book seller? | 1 | 2 | 3 |
| 2. Number of copies 5 or more? | T | T | F |
| 1. Allow 25% discount | T | F | - |
| 2. Allow no discount | X | | |
| 3. Go to decision Table 6.9 | | X | |

Table 5.5b Linked table of open-ended Decision Table

| | 1 | 2 | 3 |
|---------------------------------|---|---|---|
| 1. Number of copies 5 or more? | T | T | F |
| 2. Number of copies 20 or more? | T | F | - |
| 1. Allow 15% discount | ✓ | | |
| 2. Allow 10% discount | | ✓ | |
| 3. Allow no discount | | | ✓ |

5.3.2 Decision Tree

A Decision Tree is a graphic representation of a decision process indicating decision alternatives. It is simple and easily understood even by a layman.

Decision making involves several stages and at every stage, each of the choices results in a different outcome. The combination of all possible actions and potential events are depicted in this form to get a Decision Tree consisting of nodes and branches. A Decision Tree can represent decision rules described earlier by the Decision Table. This is shown in Figure 5.10.

5.4 DATA DICTIONARY

The major elements of a system are data flow, data stores, processes, and entities. The Data Dictionary describes all these elements of a system. It is an electronic glossary of items. It contains information that

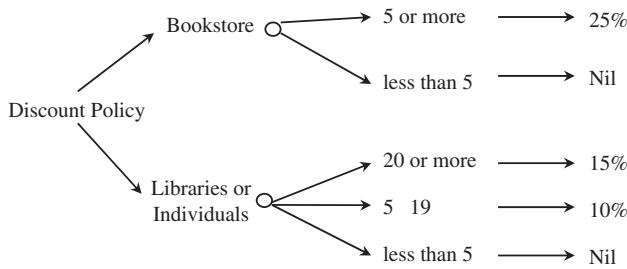


Figure 5.10 Decision Tree for showing decision rules

describes and specifies the characteristics of all elements of the DFD. It defines each element encountered during the analysis and design of a new system.

5.4.1 Data Dictionary Internals

All data in a system consist of data elements. Data elements are grouped together to make up a data structure.

The smallest unit of data that undergoes no further decomposition is known as “*data element*”. For example, data structure of an invoice may consist of elements such as invoice_number, invoice_date, customer_name, invoice_amount etc. These elements are data elements of the data structure of an invoice.

A “*data structure*” is a set of data items that are related to one another and collectively describe a component in the system. Data structures are of two types:

- (1) *Data flow* and (2) *data store*

Data flows can be considered as data structures in motion, whereas data stores are data structures at rest.

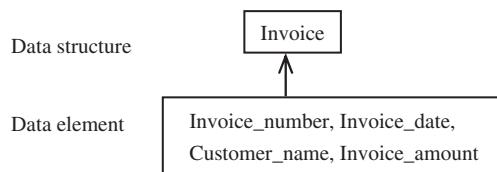


Figure 5.11 Data structure of an invoice showing various data elements

A data store consists of data structure stored in data files. Each group of data elements pertaining to an entity is a record. Data items of a record relate to different attributes of an entity. An attribute that uniquely identifies an entity is the primary key attribute.

Sometimes a data item's name may have different names in different source programs. Hence, we need to know if a data item has different names (aliases) in various source programs. We need to know the users' description of the data item in day-to-day working and also its field type and field size. Sometimes a data item or data structure is introduced purposely for system testing. Hence, Data Dictionary should also mention about this aspect.

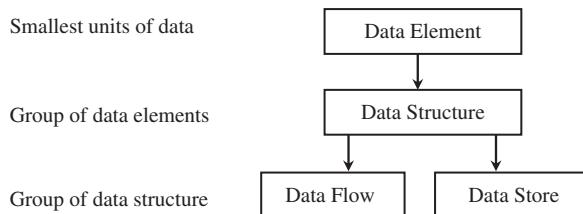


Figure 5.12 Relationship between data element, data structure, data flow and data store

The Data Dictionary might define the structure of the data file that contains details about customers as shown below:

Customer detail = Customer-number + Customer-name + Street + City + State + Pin-code +
Contact-name + Phone + Fax + E-mail

This description of Customer detail becomes a part of the Data Dictionary. Similarly all data elements used in various data flows and files (data stores) are listed in the Data Dictionary. The Data Dictionary uses the following data definition operators to define composite data items:

| | |
|-------|--|
| + | Denotes composition of two data items, e.g. “A + B” represents data A and B |
| = | Represents equivalence, e.g. “A = B + C” means that A represents B and C |
| [,] | Represents selection, i.e. any one of the data items listed in the brackets can occur. For example, [A, B] represents either A or B occurs |
| () | The contents inside the bracket represent optional data that may or may not appear; e.g. “A + (B)” represents either “A” or “A + B” occurs |
| { } | Represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data |
| /* */ | Anything appearing within /* and */ is considered as a comment |

The following rules govern the construction of Data Dictionary entries:

1. Words should be defined to stand for what they mean and not the variable names by which they may be described in the program. Hence, it is advisable to use CUSTOMER_NUMBER instead of CN. Capitalization of words helps them to stand out and may be of assistance.
2. Each word must be unique. There cannot be two definitions of the same data item.
3. Aliases, or synonyms, are allowed. For example, Customer-number may also be called as Vendor-number. However, aliases should be used only when absolutely necessary.
4. Self-defining words should not be decomposed. However, sometimes it is beneficial to decompose a data item. For instance, we might write:

Customer-address = Street + City + State + Pin-code

Data Dictionaries allow analysts to define precisely what they mean by a particular data file, data flow, or process. Some commercial software packages, usually called Data Dictionary Systems (DDS), help analysts to maintain Data Dictionaries. DDS keeps track of each term, its definition, which systems or

programs use the term, aliases, the number of times a particular term is used etc. For example, a particular data element “CUSTOMER_NUMBER” may be described in a Data Dictionary as shown in Table 5.7.

Table 5.7 Description of a data element in a Data Dictionary

| | |
|--------------|---|
| ALIASES: | Vendor |
| DESCRIPTION: | Unique identifier for customers in Account_Payable system |
| FORMAT: | Alphanumeric, six characters |
| DATA FLOWS: | Customer master; Accounts payable raw material Accounts payable adjustment; Check reconciliation |
| | Customer list; Transaction register |
| REPORTS: | Cash requirements; Customer account inquiry Check register; Customer analysis |

The Data Dictionary also contains information about various external entities. For each entity, it lists the name of all the data flows that originate from that entity. Similarly it also lists all the data flows that flow into each entity.

The Data Dictionary lists various processes and data elements used in those processes. It also gives a brief description about the function of each process. The detailed descriptions of processes are described by using process documentation tools. Decision Tables, Structured English and Decision Trees are some process documentation tools.

5.4.2 Data Dictionary Types

Data Dictionaries may either be passive, active, or in-line. Passive, active, and in-line dictionaries differ functionally as follows:

Passive Data Dictionaries: These are used for documentation purpose only. Documentation is a very important aspect of software development. A passive Data Dictionary is generally maintained as a manual rather than as a database.

Active Data Dictionaries: Besides supporting documentation, the active Data Dictionary also supports database and program development. It allows export of database definitions for creation of database and for writing program code. The IBM DB/DC Data Dictionary is an example of an active Data Dictionary.

In-line Data Dictionaries: An in-line Data Dictionary remains active during program execution. It performs tasks such as transaction validation and editing. It is associated with DBMS products such as Cullinet Software Corporation’s IDMS-R and Cin-corn System’s TOTAL.

A Data Dictionary plays a very important role in any software development process because of the following reasons:

- It provides a standard terminology for all relevant data for use in a project. In large projects different people working in the project have a tendency to use different terms to refer to the same data. This causes unnecessary confusion. Hence, a consistent vocabulary for data items is very important.
- It provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

5.5 ENTITY RELATIONSHIP MODEL

The data store notation in the DFD shows the existence of one or more groups of stored data. However, it does not give adequate information about the details of the data. Data stores of a system are mostly related with each other. However, neither DFD nor the Data Dictionary shows the relationship between different data stores. The details about the data stores and the relationships between them are shown by the ERD. The ERD is used for modeling of stored data.

The ERD is based on perception of the real world. Any real system consists of a collection of basic objects, called entities. An entity is a “thing” or “object” that is distinguishable from other objects. For example, each student of an educational institution is an entity. Entities are described by their characteristics or properties called attributes. Hence, an entity “student” may consist of some attributes such as student registration number, name, date of birth, department, class, address etc. A number of similar entities (i.e. say students) constitute an entity set. Generally, there is at least one attribute called the primary key that uniquely identifies one entity from another in a set of entities. In the above case, registration number may be the primary key (primary attribute) to identify one student from another. An entity set can have more than one such attribute to identify an entity. All such attributes are called candidate keys from which one is chosen as the primary key. An entity may not necessarily be a physical object. It can also be a conceptual object. For example, an organization may have a number of departments and each department may be considered as an entity consisting of attributes such as department_number, department_name, name_of_head, phone_number, number_of_employees, allocated_budget, etc. An attribute may be single-valued or multi-valued. For example, courses to which a student has enrolled is an attribute of “student”. However, a student can enrol in more than one course. Hence, in this example, “course” is a multi-valued attribute.

Entities of different entity sets are generally related with each other. For example, a student (of the student entity set) belongs to a department (of the department entity set). The mapping cardinality and the degree of relationship describe the way the various entities of entity sets are related with each other.

ERD uses certain notations to model stored data of a system. Entity sets are shown by a rectangular box on the ERD. An entity set represents a collection of objects (things) whose members play a role in the system being developed. Relationships are shown by the diamond-shaped boxes on the diagram. A relationship represents associations between the entity sets. Entities are identified and described by one or more facts (attributes). The attributes are represented by ellipses. A multi-valued attribute is represented by two concentric ellipses. Underlining an attribute indicates that it is a primary attribute. The important notations used in the ERD are shown in Figure 5.13.

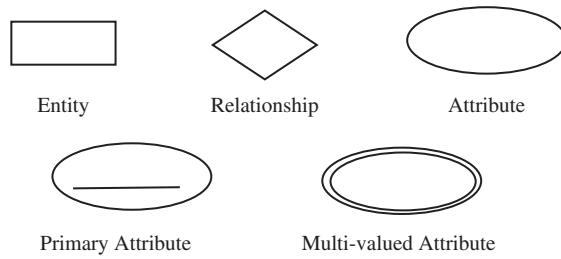


Figure 5.13 Basic symbols used in ER-diagram

Mapping Cardinalities: The binary relationship between two entity sets A and B may be described by any one of the following mapping cardinalities:

- (i) One to one: An entity of “A” is associated with one or more entities of “B” but each entity of “B” is associated with not more than one entity of “A”. It is represented by “A \leftrightarrow B”.
- (ii) One to many: An entity of “A” is associated with one or more entities of “B” but each entity of “B” is associated with not more than one entity of “A”. It is represented by “A \leftarrow B”.
- (iii) Many to one: An entity of “A” is associated with not more than one entity of “B” but each entity of “B” is associated with one or more entities of “A”. It is represented by “A \rightarrow B”.
- (iv) Many to many: An entity of “A” is associated with one or more entities of “B” and vice versa. It is represented by undirected line “A — B”.

Different types of mapping cardinality are shown in Figure 5.14.

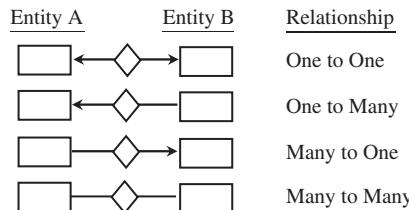


Figure 5.14 Different types of mapping cardinality

To illustrate the use of ERD, the data model of an “Order Processing System” is shown in Figure 5.15.

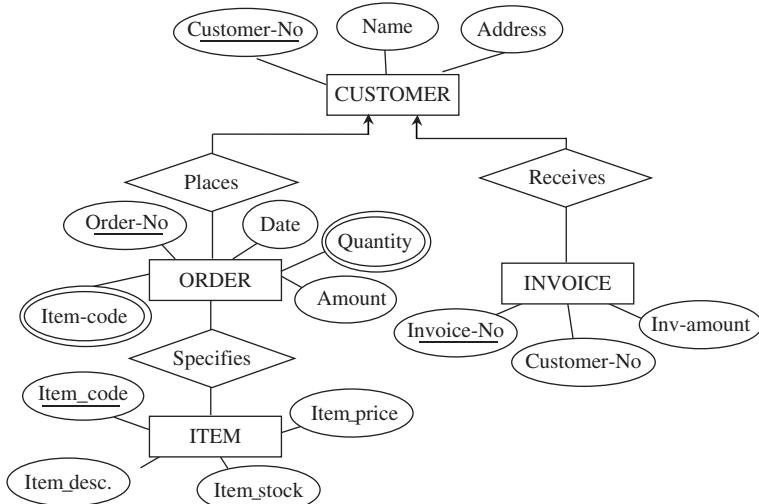


Figure 5.15 ER diagram for order processing system

The ERD shows four entity sets namely CUSTOMER, ORDER, INVOICE, and ITEM. The attributes of entities are shown in ellipses. The ellipse with the attribute name underlined indicates the primary attribute. The attribute shown in a double ellipse indicates that it is a multi-valued attribute. Note that the

attribute “item_code” is a multi-valued attribute of the entity set “Order” and a primary attribute of the entity set “Item”. All the entity sets are related with each other through some common attribute. Note the mapping cardinality. A customer can place more than one order and receive more than one invoice; the converse is not true. This is shown by arrows pointing towards customer.

Let us take another example of an educational institution. The ERD is shown in Figure 5.16. It shows four entity sets namely STUDENT, TEACHER, COURSE, and DEPARTMENT. The attribute “Specialization” shown in a double ellipse indicates that it is a multi-valued attribute. All these entity sets are related to each other.

The relationship “Belongs-to” between Teacher and Department and between Student and Department are joined by one-ended arrows. This specifies that a student or a teacher cannot belong to more than one department but a department can have more than one student or teacher.

Similarly, each course is offered by only one department but a department can conduct more than one course. The relationships between Teacher and Course and between Student and Course are joined by simple lines. This specifies that a teacher can teach more than one course and a course can also be assigned to more than one teacher. Also, students can be enrolled into more than one course and there can be more than one student enrolled in a course.

Hence, all the necessary details about entities, relationships between entities, and constraints can be easily depicted by the ERD. Thus, it is very helpful in designing the data structure of the software.

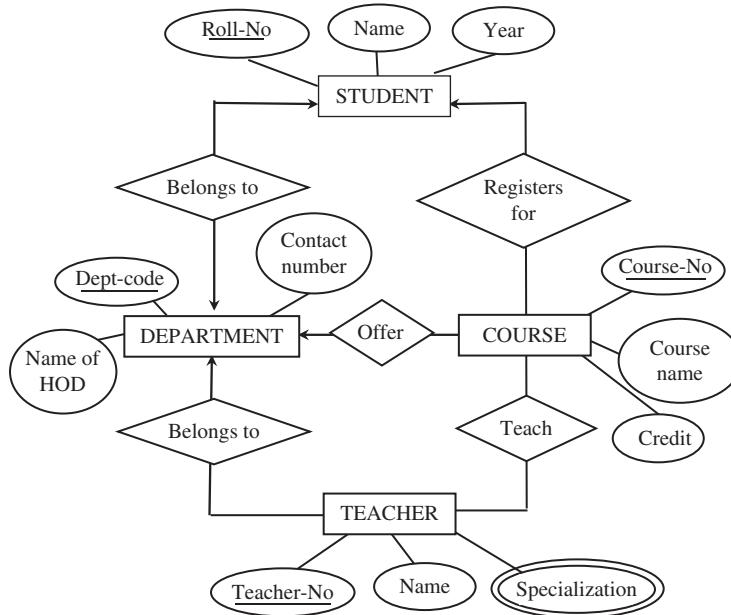


Figure 5.16 ER diagram for an educational system

5.6 STATE TRANSITION DIAGRAM

The State Transition Diagram (STD) highlights the time-dependent behavior of a system. A system's time-dependent behavior is very important for real-time systems. Telephone switching systems, high-speed data acquisition systems, process control systems are some examples of real-time systems. A real-time

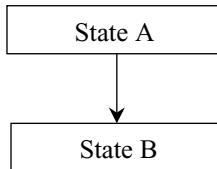
system may be passive or active. A passive system does not seek to control the surrounding environment. Many high-speed data acquisition systems fall into this category (e.g. a system capturing high-speed scientific data from a satellite). An active real-time system seeks to maintain control over some aspect of the surrounding environment. Process control systems such temperature control or pressure control in thermal power plant are examples of active systems.

A system has some properties or attributes. The state of a system is described by its properties or attributes. When any change occurs in the properties of a system, it means that the state of the system has undergone a change. The change takes place due to the occurrence of some event. A system may have a number of different states. However, only some of these states may be meaningful for modeling. A change of state of the system from one state to another is governed by certain rules.

How the transition of system takes place from one state to another is modeled by the STD. The STD has two major components. They are:

- (i) States represented by a rectangle 
- (ii) State changes represented by arrows 

Using the above notations, the change of the state of a system from state A to state B is represented as under:



Let us consider a system that has three states. The system can change from state 1 to state 2. When the system is in state 2, it can change to either state 3 or back to state 1. However, the system cannot change from state 1 directly to state 3, whereas it can change directly from state 3 back to state 1.

The initial state is typically drawn at the top of the diagram (though this is not mandatory). If an arrow that is not originating from any other state points to a state, then that state is identified as the initial state. A system can have only one initial state. Similarly, the final state is generally drawn at the bottom of the diagram (this is not mandatory). A final state is identified by the absence of any arrow leading out of state. Hence, Figure 5.17a is redrawn and shown in Figure 5.17b. It shows state 1 and state 3 as initial and final states, respectively.

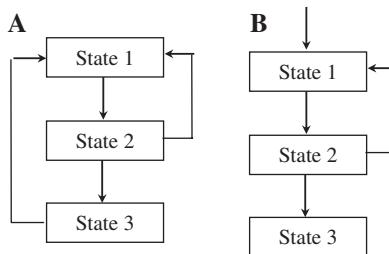
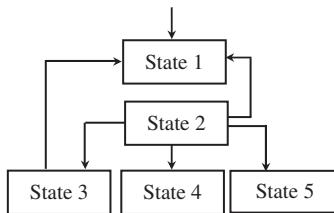


Figure 5.17 (a) STD not showing initial and final state. **(b)** STD showing initial and final state

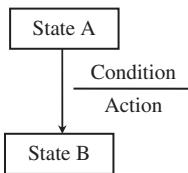
A system can have multiple final states. In such a case, the various final states are mutually exclusive, i.e. only one of the final states can occur during any one execution of the system. Figure 5.18 shows an example in which the possible final states are states 4 and 5.

**Figure 5.18** Multiple final states of a system

Two more things are needed to complete the STD. These are:

- (i) Conditions that cause a change in the state of the system
- (ii) Actions that the system takes when its state undergoes a change

In STDs, the conditions and actions are shown next to the arrow connecting two related states. This is illustrated in Figure 5.19.

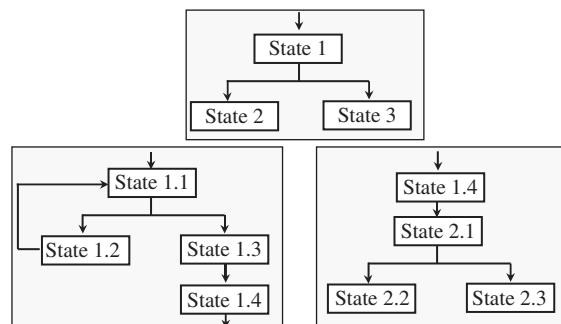
**Figure 5.19** ST diagram showing conditions and actions

A condition is an event in the external environment that the system is capable of detecting. It can be a signal, an interrupt, or the arrival of a packet of data.

However, as part of the change of state, the system typically takes one or more actions such as: produce an output, display a message on the terminal, carry out a calculation, etc. Thus, actions shown on the STDs are responses that are either sent back to the external environment or to a data store.

Partitioned Diagrams: In a complex system, there may be a number of distinct system states. To show them all on a single diagram is difficult, and it makes the diagram untidy. Thus, the STD can be partitioned and leveled in a similar manner as done in case of data flow diagrams.

Figure 5.20 shows an example of two levels of partitioned STDs. They are also called extended STDs.

**Figure 5.20** Structure of partitioned state

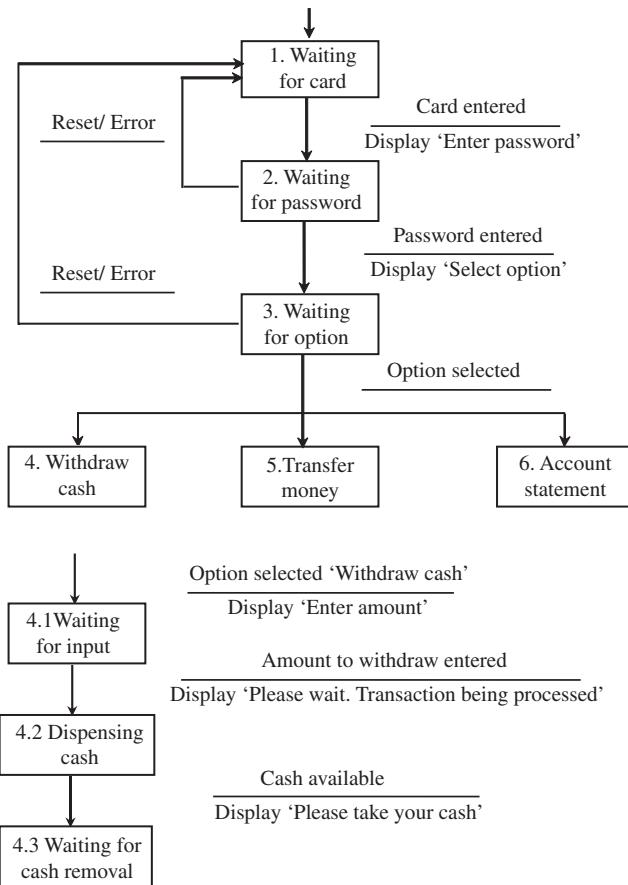


Figure 5.21 Extended state transition diagram for ATM

In a partitioned STD, any individual state in a higher-level diagram can become the initial state for a lower-level diagram. The lower-level diagrams describe the states of a higher-level diagram in greater detail. The final state(s) in a lower-level diagram correspond to the exit conditions in the associated higher-level state. An example of partitioned STD for an Automated Teller Machine (ATM) of banks is shown in Figure 5.21.

After building the preliminary STD, it should be checked for consistency. The following guidelines may be followed:

- Have all states been defined?
- Can all the states be reached? Is there any state that does not have paths leading up to it?
- Is there an exit from every state? As mentioned above, the system may have one or more final states with multiple entrances into them; but all other states must have a successor state.
- In each state, does the system respond properly to all possible conditions? This is the most common error when building an STD: the systems analyst identifies the state changes when normal conditions occur, but fails to specify the behavior of the system for unexpected conditions.

Suppose the system was modeled on the assumption that the user will press the right key to cause a change from state 1 to state 2. However, what if the user presses some other (wrong) key? If the system behavior is not specified, there is a chance that this aspect will also be overlooked while programming.

The STD is a useful modeling tool for describing the behavior of any system. It is very useful for describing the behavior of computer human interface. From the past trend, we can expect that more and more systems, whether business-oriented or scientific/engineering in nature, will take on real-time overtones. Hence, the popularity of the STD is going to increase in future.

SUMMARY

The purpose of SA is to analyse users' requirement, carry out functional decomposition of system, and represent the same through some standard graphical tools. It produces a "Structured Requirements Specification" comprising the Environmental Model and the Behavioral Model.

The Environmental Model defines the boundary and interaction between the system and the outside world. It is composed of Statement of Purpose, Context Diagram, and Event List. The Behavioral Model consists of DFD, Process Specification, Data Dictionary, ERD, and STD.

DFD is a graphical description of a system. It shows the system's processes, flow of data between processes (data flows), data store, and external entities. DFD is also called as "bubble chart". The highest level of DFD (level-0 DFD) depicts the summary of a system. It is also called the Context Diagram. The major processes of system are shown in the next lower level (level-1 DFD). Each process of the level-1 DFD can be shown in greater detail in the level-2 DFD and so on.

A Process Specification describes the purpose of processes depicted in the DFD. The decision rules are better depicted by documentation tools such as Decision Table and Decision Tree.

Data Dictionary is an electronic glossary of data items. It uses various data definition operators to define composite data items. It also contains information about various external entities and processes. Commercial software packages called DDS are available to maintain Data Dictionaries.

The ERD is used for modeling of stored data. Its major components are entity sets, attributes of entities, and relationship between entities. It uses certain notations to show these components of a system.

The state of a system is described by its properties or attributes. Any change in properties of the system signifies change of state. The change in state takes place due to the occurrence of some event. How the transition of the system takes place from one state to another is modeled by the STD.

EXERCISES

1. What is the purpose of an SA?
2. List various models used in an SA. What are their major contents?
3. Why is it necessary to use standard tools for documentation? List various standard tools used in SA.
4. What is DFD? List four major components of DFD their corresponding symbols.
5. Explain why leveling of DFDs is important.
6. Explain the term balancing of DFD.
7. Distinguish between "Physical DFD" and "Logical DFD".
8. Explain why process specification is necessary.
9. Describe the structure of a Decision Table.

10. Distinguish between “Limited Entry Form” and “Extended Entry Form” of the Decision Table.
11. What are open-ended linked Decision Tables?
12. Compare Decision Table and Decision Tree.
13. Describe the usefulness of the Data Dictionary.
14. What are the major elements of a system that are described by Data Dictionary?
15. Distinguish between Data Flow and data store.
16. List various data definition operators used in Data Dictionary to define composite data items.
17. Distinguish between passive, active, or in-line Data Dictionary.
18. What is the purpose of an ERD?
19. What are the major components of an ERD?
20. Design an ERD for a Hospital consisting of Wards, Doctors, and Patients. Clearly highlight entities, relationships, the Primary key, and the meeting constraints.
21. Identify the various entities, their attributes, and relationships between entities for a University examination system and present the design through an ERD.
22. What is the purpose of an STD? What are its major components?
23. What kind of system is most likely to use an STD as a modeling tool?
24. Define a system and its states with suitable example.
25. What are the conditions and actions in an STD? How are they shown?
26. What are the guidelines for determining consistency of an STD?
27. Draw an STD for dialling a number on phone.
28. Draw an STD for the use interface menu for Microsoft Word.

This page is intentionally left blank.

STRUCTURED DESIGN

Structured Design is a systematic methodology to determine design specification of software. The basic principles, tools and techniques of structured methodology are discussed in this chapter. It covers the four components of software design, namely, architectural design, detail design, data design and interface design. This chapter describes the following concepts, tools and techniques of structured design:

- *Coupling and cohesion*
- *Structure chart*
- *Transaction analysis and transform analysis*
- *Program flowchart*
- *Structured flowchart*
- *HIPO documentation*

Designing is a creative activity. However, certain standard procedures, tools and principles called ‘structured methodology’ are generally followed to produce a good design. The structured methodology helps to produce a system that is easy to read, easy to code and easy to maintain. It places emphasis on a structured procedure. In this chapter, we shall discuss structured methodologies and some tools used for the design of software.

6.1 STRUCTURED DESIGN METHODOLOGIES

Structured methodology comprises structured analysis and structured design. Structured design is a disciplined approach to software design.

- It allows the form of problem to guide the form of solution.
- It is based on the principle of simplifying a large complex system by partitioning it into smaller modules.
- It favours the use of standard graphic tools to aid system design.
- It offers a set of strategies for developing a solution.
- It offers a set of criteria for the evaluation of a good design.

There are four components to software design:

1. **Architectural design:** It defines the relationship between structural components (modules) of software.
2. **Detail design:** It is a component-level design that gives procedural details of software components (modules); i.e. "How is a particular piece of processing done?"
3. **Data design:** It transforms the data model represented by the ER diagram and the data dictionary into data structures required for software implementation. The purpose of the software system is to transform input data into outputs. Appropriate data representation is the key to quality software.
4. **Interface design:** A computer system is meant for different users. The users interact with the system for various purposes. Interface design describes the communications between the software and the users.

The various processes of the system and flow of data among them are depicted in the analysis models by means of Data Flow Diagram (DFD) and other graphical tools. The first step in design is to identify and define the top-level modules that the software should consist of. Each module of software should perform some group of functions and contributes to the objectives of the system.

The top-level modules of a system are decomposed into further smaller sub-modules and the process continued until the lower-level modules are of manageable size and complexity. This process is called *modularization or decomposition*. The modules are organized into a hierarchical structure. Each module accepts certain input, performs certain processes and gives certain output. The modules are designed as independent entities and then integrated with each other.

Structured design emphasizes on the hierarchical view of the system. The top level shows the most important division of work whereas the lowest level at the bottom of the hierarchy shows the details. The module at the lower level is accessed by the module at its higher level; i.e. the modules are accessed from top to bottom. This approach to system design is called the *top-down approach*. The top-down approach is generally followed throughout the entire process of software design. For example, the main menu contains several choices. Making one choice produces another menu, where more options are presented to the user. This feature permits the user to select one option at a time from the number of choices presented in the menu and makes the system more user-friendly.

Modularity is a strategy for avoiding errors in software design. In a properly modular system, the content of each module is generally designed to perform only a single, specific function. Modular design is easy to understand. It makes the system maintainable and easy to modify.

6.2 COUPLING AND COHESION

For an integrated system, it is necessary to have some exchange of data between modules. This requirement makes the modules somewhat interdependent on one another. The extent to which modules are interdependent is called *coupling*. If interdependence or coupling between two modules is more, then any modification done in one module will require corresponding changes to be made in the other module. So, if modules are highly coupled, the modification of software becomes difficult. Also, an error made in one module is no longer confined to that module but it also affects the other modules. Thus, not only are there more chances of error but debugging also becomes difficult. Therefore, well-designed software should have least coupling or exchange of data among modules.

There are different ways in which modules can be coupled. Different types of coupling are described below:

1. **Data coupling:** Data coupling is the necessary exchange of data elements between modules that are otherwise quite independent of each other. This is the best kind of coupling.
2. **Stamp coupling:** In stamp coupling, data are passed in the form of data structures or records. Exchange of data structures instead of data elements tends to make the system more complicated. So, stamp coupling is not as good as data coupling.
3. **Control coupling:** When one module passes a piece of data or signal to control the action of another module, the two are said to be control-coupled. The control information tells the recipient module what actions it should perform. Control information may also be passed from a subordinate to a super-ordinate module. Such a practice is referred to as inversion.
4. **Common coupling:** When two modules refer to the same global data area, they are common-coupled. Global data areas are possible in several computer languages, such as FORTRAN (common block) and COBOL (the data division is global to any paragraph in the procedure division). The level of module interdependence becomes quite high in such cases.
5. **Content coupling:** Content coupling involves one module directly referring to the inner workings of another module. For example, one module may alter data in a second module or change a statement coded into another module. In content coupling, modules are tightly intertwined. There is no semblance of independence. It is the worst type of coupling. Fortunately, most high-level languages do not have provisions for creating content coupling.

In a good design, all instructions in a module relate to a single function. The extent to which the instructions of a module contribute to performing a single unified task is called cohesion. Modules in an information system should be *cohesive*. There are different types of cohesion as described below:

1. **Functional cohesion:** A functionally cohesive module is the most desirable type in that all instructions contained in the module pertain to a single task or function. The name of a module such as Calculate Pay, Select Vendor, Register Order etc. may suggest that it is functionally cohesive.
2. **Sequential cohesion:** A module is sequentially cohesive if a piece of data passes through from one instruction to another to produce the desired output. In sequential cohesion, the first instruction acts on the data and the second instruction uses the output of the first instruction as its input. The output of the second instruction then becomes input to the third instruction and so on. For sequential cohesion, the instructions must appear in a logical order so that forward or backward jumps are avoided.
3. **Communicational cohesion:** For communicational cohesion, the sequence of activities is not that important but the activities must act on the same data. Each instruction either acts on the same input data or is concerned with the same output data.
4. **Logical cohesion:** In a logically cohesive module, the activities to be executed are selected from outside the module. Some frequently occurring common activities such as ADD, DELETE, SORT etc. are arranged as separate modules, which other modules use.

A good design must maintain the right balance in achieving a high level of cohesion and minimum coupling.

6.3 STRUCTURE CHART

Software architecture is represented by a structure chart. The structure chart is the primary tool used in structured design. The structure chart is used to graphically depict the modular structure of software. It shows:

- how the program has been partitioned into smaller modules
 - the hierarchy and organization of modules
 - the communication interfaces between modules

The main focus in the structure chart representation is on the module structure of software and the interaction between the different modules. Structure charts, however, do not show the internal procedures performed by the module or the internal data used by the modules.

Structure chart symbols: The form of a structure chart is shown in Figure 6.1.

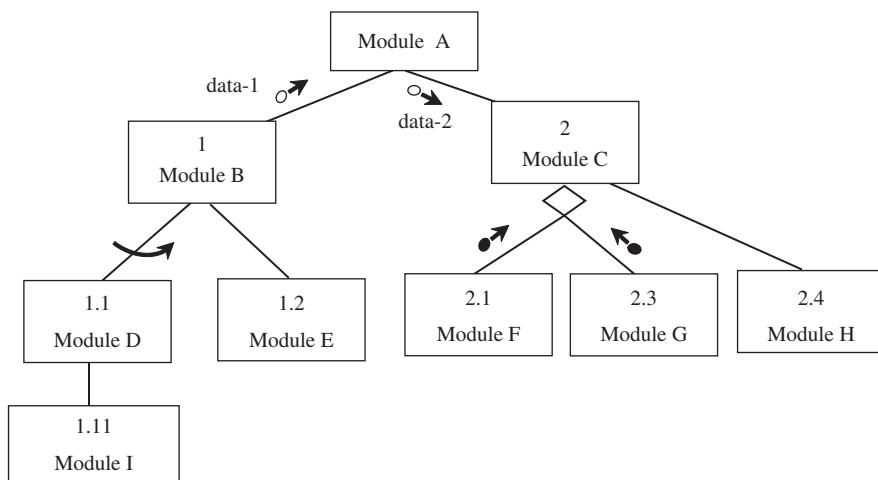


Figure 6.1 Form of a structure chart

In a structure chart, modules are depicted by rectangles. The name of the module is written inside the rectangle. The structure chart has one *main module*. This module is generally at the top of the structure chart.

In Figure 6.1, module A is the main module. Each module is connected to all the modules that it calls directly. This connection is shown by line(s) from the calling module to the modules it can call. In a structure chart, the modules are presumed to execute in a top-to-bottom, left-to-right sequence.

An arc-shaped arrow located across a line (representing a module call) indicates that the module makes iterative calls. The figure shows that module B calls module D iteratively. A diamond symbol located at the bottom of a module means that the module calls one and only one of the other lower modules that are connected to the diamond. Thus, in the said chart, only one of the two modules, F or G, is called by module C. Library modules are depicted on a structure chart as a rectangle containing a vertical line on each side. In the figure, module I is a library module.

Program modules communicate with each other through passing of data. Programs may also communicate with each other through passing of messages or control parameters, called flags. Data being passed is represented by named arrows with a small circle at one end. The direction of the arrow shows the direction of data flow. Note that data-1 is being passed “up” from module B to its parent, module A. The downward direction of the arrow for data-2 implies that the main module A is passing it to module C. The control flags are depicted by an arrow with a darkened circle at one end. As with data, the direction of the arrow indicates the source module and receiving modules.

6.4 MAPPING DFD INTO A STRUCTURE CHART

In structured design, the DFD representation of the system obtained from structured analysis is transformed into a structure chart. The modules on a structure chart are identified from processes appearing on the logical DFDs. However, as the first step, the logical DFDs obtained from the structured analysis may need to be revised/refined before these are used for design.

6.4.1 Refinement of DFD

For design of a structure chart from DFD, it is desirable that the processes appearing on the DFD should do one function. Thus, some elementary processes may need to be expanded into two or more smaller processes so that each process performs a single function. As a general rule of thumb, a process should have either one input or one output.

In order to keep DFDs from becoming overly cluttered, these are often drawn to show main processes. The processes for reading, modifying and deleting data in a data-store are not shown on the DFDs during analysis. Thus, DFDs need to be revised during design to include processes to handle data access and maintenance.

Similarly, many of the trivial business processing, exceptions and internal controls are not shown. For example, DFD may show a process receiving input data from an external entity (such as a customer), doing some processing on that data and then passing the output to another process. However, in reality, the original input data may need to be edited and proper error-handling routines performed before they are processed. To make the DFD simple, these processing details are purposely not included in the DFD at the analysis stage. So, during systems design, these necessary details (editing, error handling etc.) must be included in the revised DFDs.

The processes of a system are represented by DFD. The DFD of a system may be categorized into two types based on the main focus of its processes. These are: (1) transaction-centred system and (2) transform-centred system.

The primary function of the transaction-centred system is to send data to their proper destinations within the system. Data enter into the central module of the system called the transaction centre from where these are routed to different processes/locations based on their data type. The transaction centre is the single point of distribution of data in the system. The transaction centre does not do any transformation on data. On the other hand, the primary function of the transform-centred system is to transform input data into output. Generally, a transaction-based system has transform centres and vice versa.

Normally, we start with high-level DFD (Level-1 DFD) and convert it into module representation. This is done either by using transform analysis or by using transaction analysis. Then, we proceed further and repeat the process for the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular portion of DFD.

6.4.2 Transaction Analysis

In a transaction-centred DFD, the data enter into a central process (module) called the transaction centre from where it is routed to different sequences of processes. In the transaction-centred DFD, the data flow converges on a process and then flows away along many different paths.

The transaction-centred portion of a DFD is shown in Figure 6.2. It shows typical processes of a banking system. The central transaction centre is a process that evaluates data and routes it to other processes (modules). The transaction centre process branches into different paths. Deposits to savings would take one path and withdrawals from savings another path. Each path leads to modules designed for processing a particular type of transaction.

Each of the different ways in which input data are handled is a transaction. A simple way to identify a transaction is to trace the input data to the output. All the traversed bubbles in a particular path belong to one transaction.

All the bubbles lying on a path should be mapped to the same module on the structure chart. In the structure chart, a root module is drawn and the identified transaction modules are drawn below this module. Based on this, a structure chart is drawn as shown in Figure 6.3.

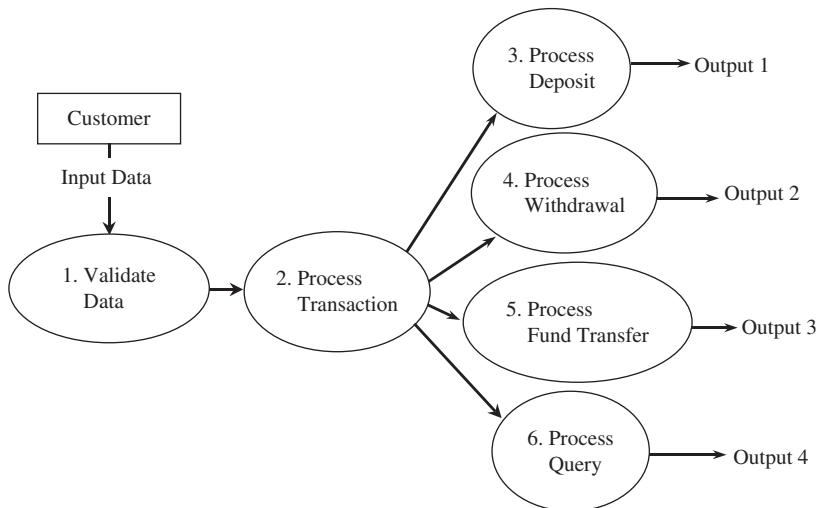


Figure 6.2 A transaction centre in a data flow diagram

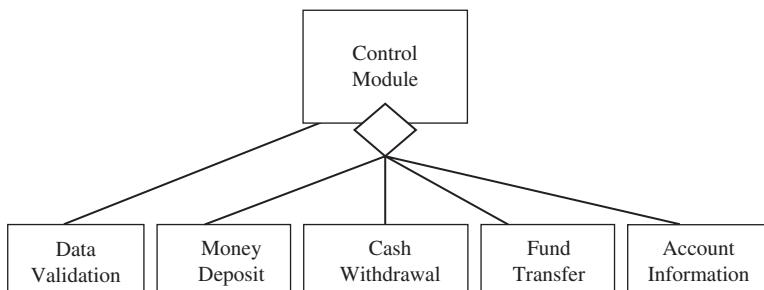


Figure 6.3 A transaction-centred system design

On-line, menu-driven systems are transaction-centred systems. In these systems, the paths selected from a menu lead to the appropriate parts of the system. Processes along a transaction path often have user interactions.

6.4.3 Transform Analysis

Data processing involves transformation of input data into output. The processes that do the data transformation are the main focus in transform-centred DFD. The transform-centred portion of DFD normally contains three types of processes:

1. input-related process
2. data transformation process
3. output-related process

The input portion of the DFD includes processes that accept, validate or edit input data from external entities or data stores. An input process is concerned with conversion of input data obtained in physical form into desired logical form (internal tables, lists etc.). This permits the data to be processed further. Each input portion is called an *afferent* branch. The output portion of a DFD is concerned with conversion of output data in logical form into physical form (e.g. printing or storing on data file). Each output portion is called an *efferent* branch. The remaining portion of a DFD does some transformation (logical processing) of input data to get output. This portion of DFD is called the *central transform*. The central transform contains the essential elements of the system where data input streams are transformed into data output streams. So, identification of the central transform is the important thing in transform analysis.

The technique for identifying the afferent branch, central transform and efferent branch of DFD is as follows. Each input data flow is traced forward through the sequence until it reaches a process that does processing (transformation of data) or an output function. All processes except the last process constitute the afferent branch. Similarly, each output data flow is traced backward through the sequence until it reaches a process that does processing (transformation of data) or an input data flow. All processes except the last process constitute the efferent branch. All processes that are neither afferent nor efferent constitute the central transform. The central transform may consist of either a single process or more than one process. Each process of the central transform may become a module in the structure chart.

In a transform-centred DFD, one or more data flows converge on a single area of the system where they are “transformed.” The data transformation may be done in one or more processes. This is in contrast to the transaction-driven portion of DFD where one of several possible paths is traversed depending upon the input data item.

Let us consider an example of transform-centred DFD shown in Figure 6.4. The first process (process-4.1) receives an account number and retrieves the relevant customer record from the data store. The next process (process-4.2) gets valid transaction-data (money to be withdrawn from the account) from the customer. The third process (process-4.3) does the data transformation by updating the customer record. So, process-4.3 is the central transform.

We can ascertain this by examining the DFD. In the figure, we see that the data-flows ‘customer record’ and ‘amount to be withdrawn’ converge on process-4.3. Further, we look at the process that follows process-4.3. We find that it (process-4.4) is a process to print the transaction detail. The primary purpose of the print process is to make the layout of information necessary for the printing. A process that prints a statement would most likely not derive any new data. So, we can infer that process-4.3 is indeed the central transform. The figure shows the data-flow diagram with the central transform process enclosed by the dotted line.

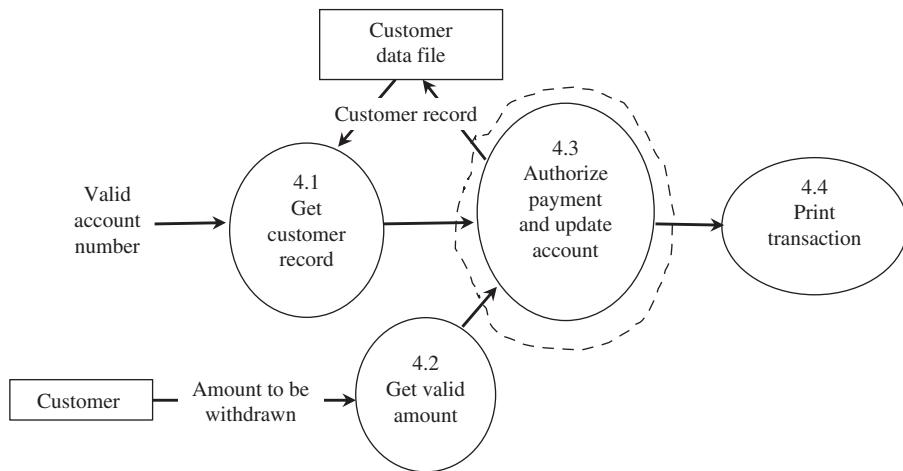


Figure 6.4 DFD showing central transform with dotted line

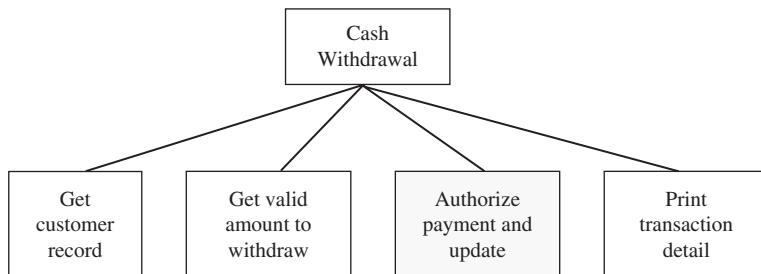


Figure 6.5 Structured chart of cash withdrawal module

In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and the efferent branches. These are drawn below the module that would invoke them. In this particular example, the central transform performs only a single function, namely, 'authorize payment and update'. This function is performed by using two data-flows, namely, customer-record and valid-amount-to-withdraw.

Process-4.1 gets the customer-record based on 'valid account number' obtained from another module. Process-4.2 gets the valid-amount-to-withdraw from the external entity, i.e. the customer. So, these two processes constitute the afferent branches. We know that the data must be formatted in a particular way before they are printed. This is done by the efferent process 'print transaction detail'. So, we can have separate modules for each of these four processes in the structure chart. The initial structure chart is shown in Figure 6.5.

Here, the transform module is shown by a shaded rectangle. The initial structure chart is also called first-cut structure chart or draft structure chart.

In the next step of transform analysis, the structure chart is refined by adding sub-functions, such as read and write modules, error-handling modules, initialization and termination process etc. This process of breaking functional components into subcomponents is called factoring.

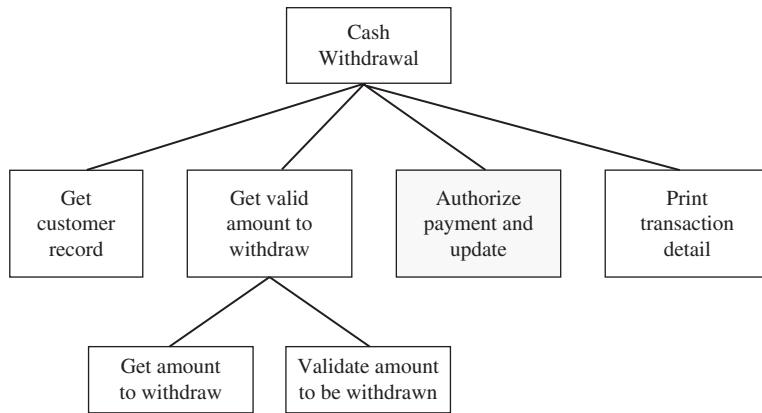


Figure 6.6 Refined structured chart of cash withdrawal module

In our example, to get valid-amount-to-withdraw, the data are first fed into the system by an external entity (customer). But, the customer may enter an amount that he is not entitled to withdraw. For example, the amount entered by the customer may be more than the balance he has in his account. The amount entered may be too high, which may be more than the permissible limit specified by the bank for safety purposes. So, the amount entered by customer has to be validated. So, we can have two modules, one for reading and the other for validating data entered by the customer. Accordingly, the structure chart is refined and shown in Figure 6.6. Any additional refinements to this particular structure chart can be done in the form of design improvements.

6.5 DATA DESIGN

Data design is done based on ER diagram and Data Dictionary. Data design may be done either as a file-based system or as a database management system (DBMS).

In the file-based approach, the data files are custom designed for each application module. There is generally no sharing of data among various application modules. The application programs are dependent on structure of data file and vice versa. This approach suffers from the following main drawbacks:

Data redundancy: Generally, many of the data items are common to different application modules. The file-oriented approach requires a separate data file for each application. So, there is unnecessary duplication or data redundancy.

Lack of data integration: In most situations, useful information is obtained by combining two or more data files. In the file-oriented approach, this task is not easy.

Program–data dependence: In file-based systems, any modification in program requires modification in the structure of data files. So, due to program–data dependence, software maintenance is difficult.

The file-based approach is not suitable for a system that involves large volume of data. In such cases, DBMS is used for organization and maintenance of data. DBMS is a software system that helps in organization, retrieval, control of integrity and security of data as well as in the development of an application

program. A number of DBMS packages are commercially available in the market. Oracle, DB2, Sybase, Ingres are some widely used DBMS software. DBMS acts as an interface between database and program modules. So, DBMS-based software are not a stand-alone system. They require additional DBMS software to be operational.

So, if volume of data is not much, file-based systems may be preferable. These are easy to design and implement. The processing speed may also be higher.

Transforming ER diagrams into data files: For example, the ER diagram given in Figure 5.16 in the previous chapter may be considered for designing of data files. The said ER diagram has four entities. It can be implemented by construction of the following Master data files corresponding to each of its entities:

STUDENT (roll-number, name, year)

TEACHER (teacher-number, name, specialization)

COURSE (course-number, course-name, course-credit)

DEPARTMENT (department-code, name-of-HOD, department-contact-number)

The above data files need to be related as depicted in the ER diagram. For this purpose, some additional data files may be created. The relationship 'Belong-to' between Student and Department can be given by the data file.

STUD-DEPT (roll-number, department-code)

Similarly, other relationships can be drawn based on common attributes as given below:

TEAC-DEPT (teacher-number, department-code)

STUD-COUR (roll-number, course-number)

COUR-DEPT (course-number, department-code)

TEAC-COUR (teacher-number, course-number)

Note that relation data files 'STUD-COUR' and 'TEAC-COUR' have a composite primary key because of many-to-many relationships.

It is very easy to transform an ER diagram into relational data files. But, it is equally critical. Data design is an important component of designing a software system.

6.6 DETAIL DESIGN

Structure charts are used to graphically depict the modular design of a program. They show: (1) how the program has been partitioned into a number of manageable modules, (2) how the modules are organized into a hierarchy and (3) the communication interfaces between the modules. But, structure charts do not show the internal data and the procedures performed by the modules. The detailed description of working procedures (algorithm) of each module is given by 'module specification'. The module specification is also called 'MSpec' and is given in graphical form by '*Program Flowchart*' and/or in textual form by *Pseudocode*.

6.6.1 Program Flowchart

The structured method for documenting the flow of a process is called the 'flow process chart'. It was introduced way back around 1920 in the field of mechanical engineering for conducting work study. It is an

effective tool for depicting an industrial process. In 1947, the American Society of Mechanical Engineers (ASME) adopted a set of symbols as the ASME standard for process charts. Later, this tool was adapted to plan computer programs and was called 'program flowchart'. The program flowchart is a popular tool for describing computer algorithms. Modern techniques such as Unified Modelling Language (UML) activity diagrams (used in the OO approach) can be considered as an extension of the flowchart.

The flowcharts are usually drawn using some standard symbols. The structure of a program can be depicted by basic flowcharting symbols as shown in Figure 6.7.

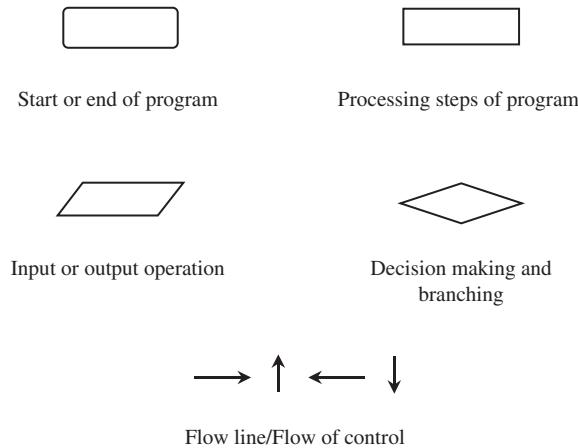


Figure 6.7 Basic symbols used in the flowchart

Start or end symbols: These are represented as circles, ovals or rounded rectangles, usually containing the word 'Start' or 'End' or some other phrase signalling the start or end of a process.

Flow of control: The arrows represent the 'flow of control' of the computer program.

Processing steps: These are represented by rectangles. The operations to be performed are written inside the rectangles. Examples: 'add 1 to X'; 'replace identified part'; 'save changes' or similar.

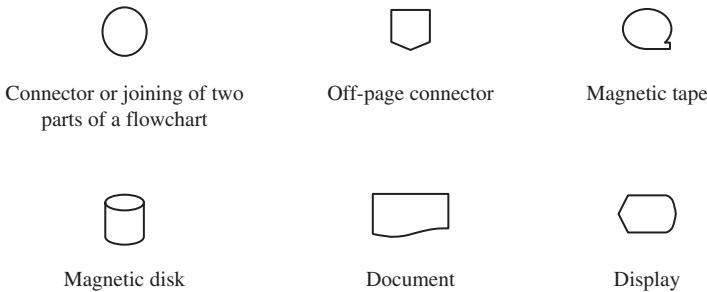
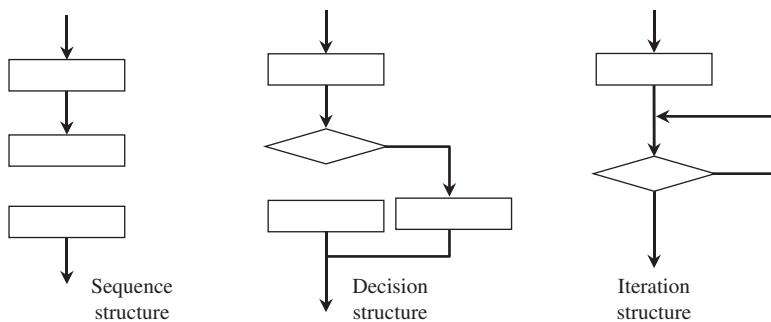
Input/Output: These are represented as parallelograms. Examples: Get X from user; display X.

Branching or decision: These are represented as rhombuses. These typically contain a condition statement (question) that has a Yes/No or True/False test. This symbol has two arrows coming out of it, one corresponding to Yes or True and the other to No or False.

In addition to the basic symbols shown above, there are many more symbols. Some of those additional symbols are shown in Figure 6.8.

A computer program normally consists of three types of structures. These are: (1) sequence structure, (2) decision structure and (3) iteration structure. These three types of structures are the building blocks of a computer program. These can be shown by using basic flowchart symbols as shown in Figure 6.9.

To illustrate the use of a flowchart, let us look at the processing logic to compute the factorial value of any natural number.

**Figure 6.8** Some additional flowchart symbols**Figure 6.9** The building blocks of a program

If N is a natural number and F is factorial of N, then F is defined as under.

$$\begin{array}{ll} F = 1 \times 2 \times 3 \times \dots \times N & \text{If } N \neq 0 \\ F = 1 & \text{If } N = 0 \end{array}$$

The logic of the program to determine factorial 'N' is described as under.

Step 1: Read value of 'N' whose factorial is to be determined.

Step 2: Take two variables 'M' and 'F' and set their values to 0 and 1, respectively.

Step 3: Increment the value of M by 1 ($M = M + 1$). Multiply F with M and let this be the new value of F (i.e. $F = F \times M$).

Step 4: Check if the value of M has reached equal to or greater than the value of N.

If not go to step 3, else go to step 5.

Step 5: The value of F is the factorial of N. Print F.

A flowchart based on the above processing logic to compute the factorial of a number is shown in Figure 6.10.

A program flowchart is a diagrammatic representation to depict the processing logic (algorithm) of a program. The program flowchart is easy to understand. It is a useful means to communicate the logic of a system to all concerned. So, it facilitates software designers to get the processing logic cross-checked by the users. The program flowchart helps in effective analysis of a problem for arriving at the solution. Once the flowchart is drawn, it becomes easy to write the program in any high-level language. It also serves as good documentation that is needed for various purposes.

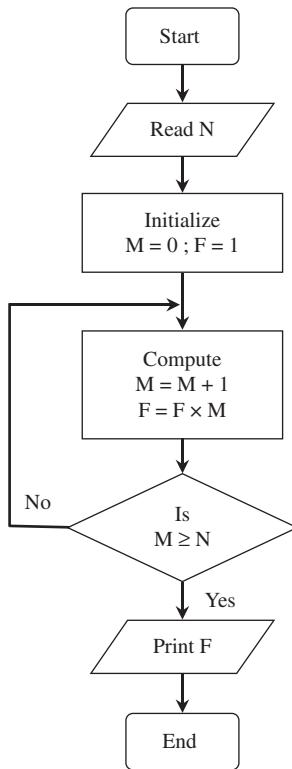
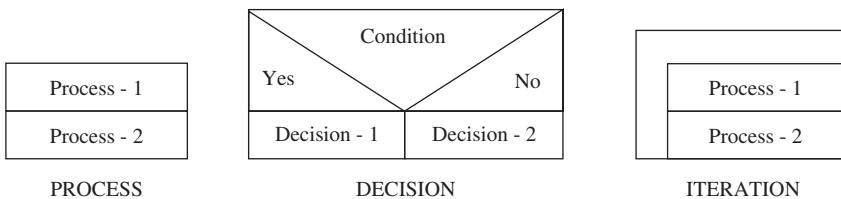


Figure 6.10 Flowchart for computing factorial of a number

6.6.2 Structured Flowchart

The structured flowchart, also called the Nassi-Shneidermann chart (N-S chart), was developed by I. Nassi and B. Shneidermann in the early 1970s. It is a graphic tool to describe the logic of computer programs. N-S charts are concise and much more compact than program flowcharts. The use structured statements, sequence, decisions and repetition constructs to depict the logic of a program.

There are three basic elements used in developing a structured flowchart (N-S chart), namely, process, decision and iteration.



Processes or steps in the program are represented by rectangular rows. A name or brief description, written inside the rectangular box, states the purpose of the process. The rectangular rows show the

sequence in which the instructions are to be executed. A rectangle divided into halves with angular lines is used for depicting selection or decision. The iteration symbol represents a loop. The condition is specified by the word Do While or Until.

Structured flowcharts do not use arrows to show flow of control. Each structured flowchart is generally shown on a single sheet of paper. To illustrate the N-S chart, the processing steps to compute factorial of a natural number are shown in Figure 6.11.

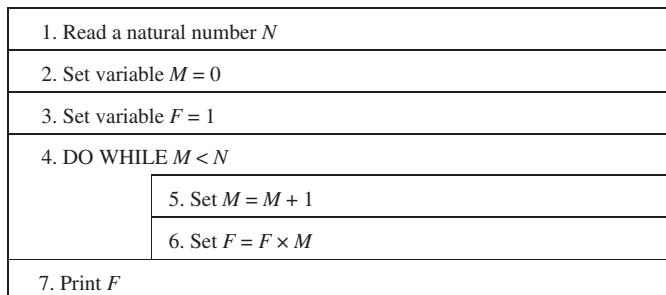


Figure 6.11 N-S chart for computing factorial of a natural number

In this example, instructions bearing numbers 1–3 are executed in sequence. Instruction number 4 is a ‘Do While’ statement for iteration, indicating that the subsequent instructions from 5 to 6 are to be repeated as long as the condition mentioned in instruction 4 is satisfied. Instruction number 7 prints the value of F , which is same as the factorial of number N .

N-S charts are very popular among system analysts and programmers because they are simple, compact and yet suitable for specifying the complex logic of computer programs.

6.6.3 Pseudocode

The objective of the method ‘Pseudocode’ (PCODE) is to describe the functionality of a module or of a procedure in a structured way. A structured program uses three types of control statements. These are:

- Sequence – sequence of statements for execution, in order (one after the other).
- Selection – used to select from a statement for execution based on certain specified conditions. It uses syntax:

if <condition> then <statement> else <statement>

- Iteration – repetition of a group of statements based on a specified condition. It uses syntax:

do <statements> until <condition> or
 do <statements> while <condition>

Pseudocode makes use of programming control structures (i.e. sequence, selection and iteration) to describe the internal logic of program modules. It may also include information about the data required (for example, variable names and types) and output to be produced (for example, report formats).

In pseudocode, the logic of operation is described with predefined English words that are mostly adopted from a structured programming language. It is called pseudocode because it is almost a program

code. Pseudocode is produced by writing a sequence of *rules for solving a problem in a finite number of steps* to meet the program requirement. *Pseudocode* is also known as *Structured English*. For illustration purposes, the pseudocode for a publisher's discount policy is shown in Table 6.1. Pseudocode is written by referring to Software Requirement Specification (SRS), DFD, Data Dictionary and Structure Chart. It is written in English language using multilevel indentation. Structures are intended to reflect the logical hierarchy. The names of data items and of processes should conform to those in the data dictionary. Sentences should also be clear, concise and precise in wording and meaning.

Table 6.1 Pseudocode for a discount policy

| Stakeholders |
|--|
| COMPUTE-DISCOUNT Sum number of copies for each book title IF (order is from book store) THEN: IF (order size is 5 or more) THEN: discount is 25% ELSE: discount is nil IF (order is from libraries or individuals) THEN: IF (order size is 20 or more) THEN: discount is 15% ELSE: IF (order size is 5 or more) THEN: discount is 10% ELSE: discount is nil |

There are many reasons for using pseudocode:

1. Pseudocode is generic in nature and is not oriented towards any particular programming language. So, pseudocode can be easily converted to program code in any programming language.
2. Writing down what needs to be done before starting the actual coding reduces the total software development time.
3. Pseudocode is like English. It is easier to read and understand than program code. So, even a person who is not a computer expert can understand pseudocode to verify if the procedure (method) given in pseudocode is correct or not. It is useful at times to check the pseudocode with the person who asked for the program to be developed.
4. Structured English can be used as comments in the program code and helps in documentation.

However, pseudocode has some limitations:

1. Pseudocode takes time to produce. If a program is not very difficult, an experienced programmer may not need pseudocode to write the program.
2. Every programming language has some special feature. But, pseudocode being language independent, it does not permit the special features of a language to be exploited.

There are more reasons for using pseudocode than against. It is definitely worth investing time to understand what it is and how to produce it.

6.7 HIPO DOCUMENTATION

In addition to Structure Charts and MSpec, HIPO documentation technique is also quite popular and is used for communicating system specifications. HIPO stands for Hierarchy plus Input Process Output. It consists of two types of diagrams:

1. Visual Table of Contents (VTOC)
2. Input Process Output (IPO)

VTOC shows the arrangement of all the modules in a hierarchical structure. Thus, it shows the overall content of the system. The detail of each module is shown by the IPO chart. The IPO chart shows various processes performed by a module. For each process, it also depicts various inputs that are used and the outputs that are generated.

To illustrate this technique, VTOC of the marketing information system of a typical organization is shown in Figure 6.12.

The VTOC diagram breaks a system into components. The name of the system appears at the top of the VTOC; the names of the major functions of the system appear on the second level. Each major module is further decomposed into smaller sub-modules to show increasing levels of detail. Modules are given suitable names and levelled. This facilitates the proper linking of lower level modules to their higher modules. The module 'Demand Forecasting' is numbered 1.21. This level number 1.21 indicates that it is a sub-module of the 'Inventory Control' module having level number 1.2.

The IPO chart of the 'Order Processing' module is shown in Figure 6.13. Module number and name, a brief description of module, etc. are written on top of the IPO chart. The chart consists of three columns. Various input data that are used in the module are listed in the first column. Processes that are done on these inputs to generate output are shown in the second column. The outputs of the module are shown in the third column.

The HIPO system forms a useful technique for system documentation. HIPO allows the analyst to represent the system graphically. HIPO charts can be drawn or modified rapidly. HIPO charts facilitate estimation of the time required to program the module and thus it helps in efficient scheduling and work-assignments.

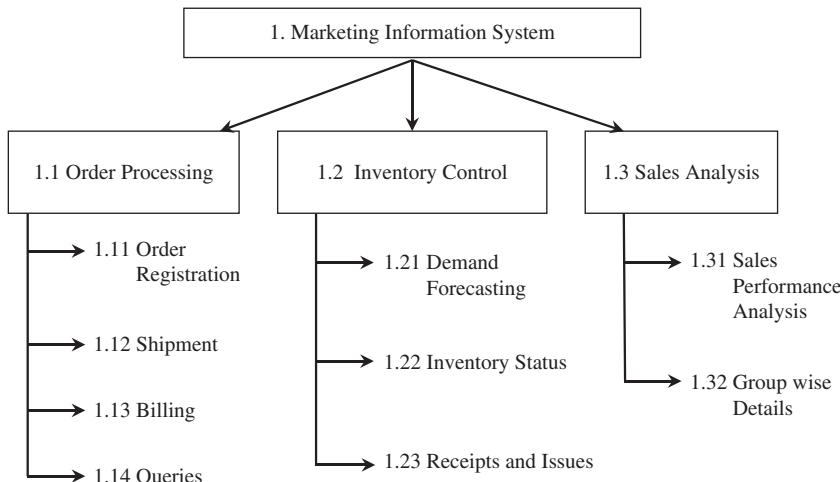


Figure 6.12 VTOC of marketing information system

| Module No. 1.1 Name: Order Processing Description: Processes for execution of orders | | Date Prepared by: |
|--|---------|---|
| INPUT | PROCESS | OUTPUT |
| Order from customer | | 1.1 Order Registration Allot unique registration number Update order master file |
| Shipment detail | | 1.12. Shipment Link customer order Compute outstanding order quantity Prepare dispatch note Dispatch note Outstanding order report |
| Dispatch note Price list of items Cheques from customers | | 1.13 Billing Prepare invoice Link invoices with cheques Invoices Bills receivable reports |
| Query options | | 1.14 Queries Process queries Answer to queries |

Figure 6.13 The IPO chart detail

SUMMARY

Structured design is a disciplined approach to software design. The analysis models form the basis for design. There are four components to software design: (1) architectural design, (2) detailed design, (3) data design and (4) interface design.

Structured design emphasizes on the hierarchical view of the system. The modules are accessed from top to bottom. Modularity is an important aspect of design.

The extent to which modules are interdependent is called coupling. The modules can be coupled by different types of coupling, such as data coupling, stamp coupling, control coupling, common coupling and content coupling. The extent to which the instructions of a module contribute to performing a single unified task is called cohesion. There are different types of cohesion such as functional cohesion, sequential cohesion, communicational cohesion and logical cohesion. So, a good design should have a high level of cohesion and minimum coupling.

Software architecture is represented by a structure chart to graphically depict the modular structure of software. Program modules communicate with each other through passing of messages or control parameters, called flags. The structure chart is derived from DFD. The modules on a structure chart are identified from processes appearing on the DFDs. The identification of modules is done by transaction analysis and transform analysis of DFDs.

Data design is done based on ER diagram and Data Dictionary. Data design may be done either as a file-based system or as a DBMS. The DBMS approach is more suitable for a system that involves large volume of data. DBMS acts as an interface between database and program modules. So, DBMS-based software require additional DBMS software. Oracle, DB2, Sybase, Ingres are some widely used DBMS software.

The detailed algorithm of each module is given by MSpec. It is given in graphical form by '*Program Flowchart*' and Structured Flowchart. The Structured Flowchart, also called the N-S chart, is much more concise than the program flowcharts. It uses structured statements, sequences, decisions and repetition constructs to depict the logic of the program.

The detailed algorithm of each module is also given in textual form by *Pseudocode*. It uses programming control structures (i.e. sequence, selection and iteration) to describe the internal logic of program modules.

The HIPO documentation technique is also quite popular and is used for communicating system specifications. It consists of two types of diagrams: (1) VTOC and (2) IPO.

EXERCISES

1. Write the principles of structured design.
2. Write the features of a well-designed system.
3. What are the different types of coupling and what is their significance to system design?
4. What is the purpose of a structure chart? Is it of any use to the user?
5. What are the graphical components of a structure chart?
6. Explain how a structure chart is related with DFD.
7. What do you mean by a transaction-centred system?
8. Distinguish between a transaction-centred system and a transform-centred system.
9. Explain how a data file-based system is different from a DBMS-based system.
10. Draw a HIPO for a payroll information system.

OBJECT-ORIENTED CONCEPTS AND PRINCIPLES

This chapter describes the basics of the Object-oriented Approach. It covers the following topics:

- *Objects*
- *Classes*
- *Messages*
- *Inheritance*
- *Abstraction*
- *Encapsulation*
- *Polymorphism*

Also, the various relationships between objects are discussed in this chapter. Finally, the object-oriented modeling techniques by Booch, Rumbaugh and Jacobson that led to the establishment of Unified Modeling Language (UML) are discussed.

The object technology has made a rapid progress since the 1980s. Object-oriented (OO) software development gathered momentum in the the 1990s and today most of the software developments are done through this approach. In this chapter, we will discuss about the fundamentals of this technology.

7.1 KEY CONCEPTS

Object technology is not only object-oriented programming (OOP), but it also includes OO analysis, design and implementation. The term object was first used in Simula language. Simula is a programming language meant for computer simulation. Here, the objects were used to simulate some aspects of reality. Some of the shortcomings of the procedural approach are removed in 'OOP'. In OOP, data are treated as critical and are not allowed to flow freely. In this approach the data are bound to the functions that operate on them. Data can only be accessed through the functions they are bound with. Thus, data

are protected from accidental modification from outside functions. OOP facilitates decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. OOP also facilitates code reusability, which is its other major advantage.

The key concepts in the object technology are focused on *objects*. In an OO terminology, anything that exists in the real world/system is an object. Therefore, identification of objects of the system is important for software development. Throughout the book we will use software development as a synonym of systems development.

7.1.1 Object

An object is anything that exists in the real world, such as a person, a place or a thing. It can be any noun or noun phrase, either physical or conceptual.

Let us consider an organization system for our analysis. Some examples of objects of the organization are given below:

Table 7.1 Examples of objects

| Object name | Description of the object |
|---------------|--|
| Alok Mishra | An employee of the organization. A physical entity |
| Marketing | A department of the organization. A conceptual entity |
| John Mathew | Head of the marketing department and also an employee of the organization |
| Kalpana Gupta | An employee of the marketing department and also an employee of the organization |

The objects Alok Mishra, John Mathew, Kalpana Gupta have similarity. They can be categorized into a group (class) such as 'Employee'.

As discussed earlier, an object contains data and functions as its integral parts. In the object terminology, the data integral to an object are called *attributes* of the object and the functions are called *methods*. This is shown in Figure 7.1. In the example stated above, the objects can have different attributes such as employee-id, name, grade, designation, address, basic pay etc. These attributes distinguish one object from another. The methods that operate on the data (attribute) of the object are also the integral parts of the object. An object has a unique identity and refers to a single entity from a group of entities.

7.1.2 Class

A class is a group of similar objects. It is a blueprint that is used to create objects. A class is a set of objects that share a common structure and common behavior. A single object is simply an *instance* of a class.

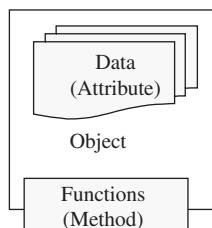


Figure 7.1 Structure of an object

Each object in a single class has the same data format and responds to the same instruction (with the exception of *polymorphism* to be discussed later in this chapter). In the object terminology, the instruction is called a *message*.

For example, Alok Mishra, John Mathew, Kalpana Gupta and other employees of organizations can be grouped together into a class of objects, say ‘Employee’. Then Alok Mishra, John Mathew and Kalpana Gupta are different instances of the same class ‘Employee’.

We can specify different methods that will operate on objects of this class. For example, we can have different methods, say method to update the attribute of an employee, method to get (calculate) salary of employee etc. These methods are named as `updateEmployee()` and `getSalary()`, respectively. Note that in the OO technology, it is a convention to name a method as combination of ‘verb’ and ‘noun’. The structure of class ‘Employee’ is shown in Figure 7.2.

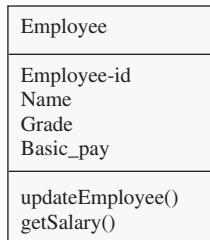


Figure 7.2 Structure of a class employee

The name of the class is shown at the top. The attributes are listed underneath. The methods are listed below that. This is the usual (standard) way of specifying a class. Strictly speaking, this is a class diagram. This will be explained in some detail in Chapter 8.

We can have another example of class ‘Shape’ as shown in Figure 7.3. Any shape has its perimeter and area as attributes. It can have method say `getArea()`, to find the area of that shape.

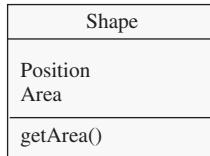


Figure 7.3 Structure of a class shape

The classes are generic in nature and the objects are specific instances of a class. For example, triangle can be a specific object of the class ‘Shape’. This can be defined in the OO language (e.g. Java or C#) as:

shape triangle;

Similarly object John Mathew can be defined as:

employee johnmathew;

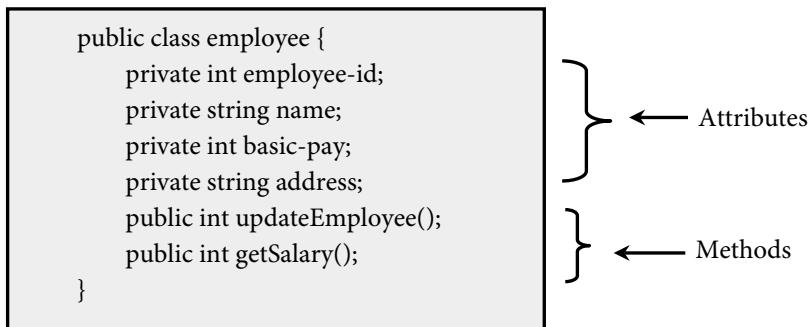
In Java and C#, ‘int’ is a general class for integer. Similarly ‘float’ is a general class for decimal (floating point) number. Hence, if we want to define variable ‘i’ as integer and ‘j’ as decimal number, then ‘i’ and ‘j’ are objects of classes ‘int’ and ‘float’, respectively.

```
int i;
float j;
```

The classes and objects that belong to respective classes are as follows:

| Class | Object |
|----------|------------|
| Int | I |
| Float | J |
| Employee | Johnmathew |
| Shape | Triangle |

The classes ‘int’ and ‘float’ are predefined classes in Java and C. However, ‘Employee’ as a class needs to be declared exclusively



For another example, consider software for a car racing game. Here, the class ‘Car’ represents a collection of cars. It will have attributes and methods as shown in Figure 7.4. A particular instance of car can be a car having color white, make X, model Y, specified size and cost. The car can have methods such as goCar() to start, accelerateCar() to accelerate and so on.

| Car |
|-----------------|
| Make |
| Model |
| Colour |
| Size |
| Cost |
| goCar() |
| accelerateCar() |
| turnLeft() |
| turnRight() |
| stopCar() |

Figure 7.4 Structure of class car

Class Hierarchy: In an OO system, classes are organized into a hierarchy. At the top of the class hierarchy are the most general classes and at the bottom are the most specific classes. A class that is higher in hierarchy to the class below it is called the superclass; and class which is below is called the subclass.

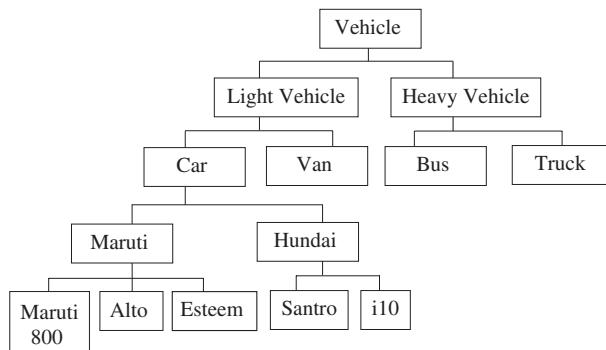


Figure 7.5 Class hierarchy of the class vehicle

Superclass is also referred to as base class or parent-class and subclass as child class. A *subclass* inherits all attributes and methods of its *superclasses*. Further, the subclass can have some additional attributes and methods specific to that class. This is also called the *derived class*. The derived class may be created from the superclass or *base class*. A class may simultaneously be the subclass to some class and superclass of some others.

For example, let us examine hierarchy structure of class vehicle shown in Figure 7.5. The ‘Car’ is a class of vehicle. It can have a superclass say ‘Light Vehicle’ and subclasses say Maruti, Hyundai, Ford, etc. The class ‘Car’ will possess all properties and behavior of ‘Light vehicle’, which is a superclass of ‘Car’. Similarly, Maruti can have different models of cars such as Maruti800, Alto, Esteem, etc. These subclasses of Maruti will possess all behaviors of Maruti plus some additional behavior specific to their own. This eliminates duplicated codes. This is a major advantage of the OO language in comparison to procedural languages. A class can share and reuse the behaviors of its superclasses.

7.1.3 Message

In object technology, the object behavior is described by methods or functions. However, to invoke those methods, one object has to pass an instruction or message to another object. Sometimes external objects can also pass such messages. However, the receiver object on which the method operates has to respond to such messages.

For example, to compute the salary of an employee, a message such as ‘getSalary’ may be sent to the object ‘Employee’. However, if that object does not have that method, then it cannot understand the message. Therefore, every object has to declare the list of methods it contains and supports. This enables other objects to check the list and send messages. It may sometimes happen that the method is not present in the class to which the object belongs. In that case the *superclass* is searched and the process continued till the required method is found. In this way, a message sent to invoke a method (say getSalary or updateEmployee) produces results.

7.1.4 Inheritance

Inheritance is the process by which objects can acquire the properties of objects of other classes. When two classes have a parent-child relationship, the child class (subclass) inherits the properties of the parent class (superclass). In OOP, *inheritance* provides reusability, like, adding additional features to an

existing class without modifying it. This is achieved by deriving a new class from the existing one. The new class will have combined features of both the classes. Hence, inheritance supports programming by extension as opposed to programming by reinvention.

For example, the class 'Car' contains the general attributes of cars. Maruti class inherits all the general attributes of cars and adds attributes specific to that of Maruti. Similarly, all the subclasses of the class Maruti will inherit those of Maruti and add their own specific attributes.

For example, if Maruti uses the showSpeed() method of the general car, then in response to a message to show the speed, it has to search the methods of its superclass and act accordingly. This may be seen in Figure 7.6. However, it may so happen that some class of car, say Hyundai, does not use the general method of showSpeed(), but some other method (say instrumentation-controlled method). In that case, only the method of class Hyundai can be updated and there will be no conflict with methods of other classes. The general method showSpeed will continue to exist and used by other subclasses.

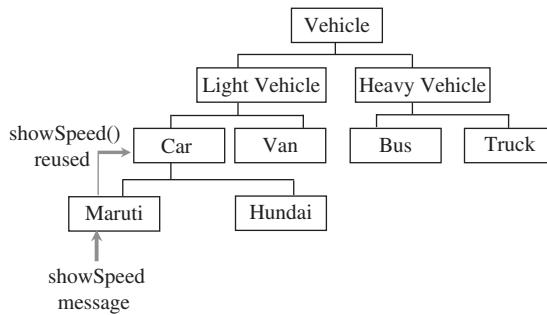


Figure 7.6 Reusability of attributes and methods

As discussed earlier in this section, Maruti is one of the subclasses of the class 'Car'. However, Maruti also manufactures vans. Hence, it can also be the subclass of the class 'Van'. This is known as '*multiple inheritance*' and sometimes it can cause some kind of problem. Hence, it is safer to inherit from the most appropriate class and then add the object of another class as attribute.

7.1.5 Abstraction

The process of picking up common features of objects is known as abstraction. The main purpose of abstraction is to consider those aspects that are essential and to remove the unimportant aspects. It is a mechanism for reducing complexity of the software. Abstraction allows the analyst to consider only the essential aspects of objects. This is very important for a high-level analysis model. Under this, the analyst only tries to find out what is to be done and subsequently extend the model by adding more details.

Abstraction can be defined as the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide precise conceptual boundaries, relative to the perspective of the viewer. Abstraction arises from recognition of similarities between objects, situations and processes in the real world. The principle of abstraction focuses on similarities and ignores the differences in the initial stages. Important points about abstraction are as under:

- It simplifies the understanding of any complex system.
- It hides irrelevant information.
- It simplifies by comparing to something similar in the real world.

Abstraction is an important concept of object technology. A class hierarchy can be viewed as an abstraction level. Each superclass can be considered as an abstraction of its subclasses. For example, the basic electronic components can be at the lowest level of abstraction, whereas the logic gates can be at the next higher level. The digital circuits, which are built using the logic-gates, will be at the next higher level and the CPU built using digital circuits will be at the highest level of abstraction. Similarly, in the programming language front, the high-level language is at the highest level of abstraction. Levels of abstraction from highest to lowest are shown below.

| Level | Hardware Front | Software Front |
|-------|-----------------------------|-------------------------|
| 1 | Central Processing Unit | High-Level Language |
| 2 | Digital Circuits | Assembly-Level Language |
| 3 | Logic Gates | Machine-Level Language |
| 4 | Basic Electronic Components | Hardware Level |

Also, an object itself can be treated as a data abstraction entity, because it speaks about the private data items of the objects and the methods used for manipulating these data items.

7.1.6 Encapsulation

Abstraction and encapsulation are complementary concepts. Abstraction helps people to think about what they are doing, whereas encapsulation allows program changes to be reliably made with limited effort. Abstraction focuses on the observable behavior of an object, whereas encapsulation focuses on the implementation that gives rise to this behavior.

Encapsulate is derived as ‘En’ + ‘Capsulate’ that means ‘in a capsule’. Here, data and procedures are tied together to prevent them from external misuse. Encapsulation means containing or packaging data within an object. Encapsulation also means data hiding. Data hiding is the process of hiding all aspects of an object that do not contribute to its essential characteristics. For example, a car’s dashboard hides the complexity and internal workings of its engine.

To understand encapsulation, let us consider the object ‘Employee’. The attributes of employees say ‘salary’ is kept hidden inside the object and may be made accessible only through the method meant for the purpose. The method resides within the object. For example, if `getSalary()` is a method of the object ‘Employee’ to get the salary of an employee, then the salary of an employee can be obtained by no other way but by this method. Other objects can also send messages to the object ‘Employee’ and get the salary of an employee by the `getSalary()` method. Other objects need not be concerned with the attributes and internal structure of the object. They pass on the message based on the list of methods published by the object.

An object has a border consisting of methods, which allow other objects to interact with it by sending messages. This is shown in Figure 7.7. The figure shows that attributes are hidden inside the object by a method.

7.1.7 Polymorphism

Polymorphism means different forms (poly means different and morph means form). *Polymorphism* means the ability to take more than one form. In the context of object technology, polymorphism means the same operation behaving differently on different classes. Two features of an object that achieves polymorphism are method overloading and method overriding.

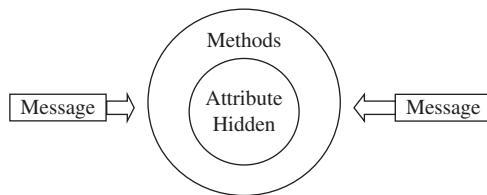


Figure 7.7 Encapsulation of an object

The practice of using the same method name to denote several operations is called ‘*Method Overloading*’. Some OO languages allow overloading of both functions and operators like +, -, etc. For example, a function ‘Append’ may be overloaded to accept different types of data. One ‘Append’ function adds an integer value to a string, and another ‘Append’ function adds a float value.

Method Overriding refers to the practice of providing different implementations of a method in the derived class. A base class ‘Shape’ has function ‘Draw’, which draws the shape on screen. The derived classes ‘Line’, ‘Rectangle’, ‘Circle’, etc. implement their own ‘Draw’ methods, which draw respective shapes on the screen. These features can be implemented respectively in two ways:

- (1) *Static Binding* and (2) *Dynamic Binding*

Suppose the class *circle* has two definitions for the *create* method. These definitions work based on the parameters of the *create* method. These definitions of the *create()* method may be seen as given below:

```
class Circle{
    private float x, y, radius;
    private int fillType;
    public create(float x, float y, float radius); //First definition//
    public create(float x, float y, float radius, int fillType); //Second definition//
}
```

The first creates a circle based on the center point of coordinates x and y, whereas the second fills the circle with the fillType. These methods are implemented at the time of compilation, based on data types (signature) of the method. This is termed as static binding. In static binding, the same message can result in different actions when received by different objects. This occurs when multiple methods with the same operation name exist.

The process of determining the appropriate function that should be invoked during the execution of a program is termed as Dynamic Binding. This occurs when polymorphic calls are issued. Unlike static binding, which occurs during compilation time, dynamic binding occurs during runtime.

An example to illustrate method overriding that can be implemented in structured programming is given as follows:

```
if (shape==Line) then draw_line();
else if (shape==Rectangle) then draw_rectangle();
else if (shape==Circle) then draw_circle();
```

This can be implemented using *draw()* method under OOP methodologies.

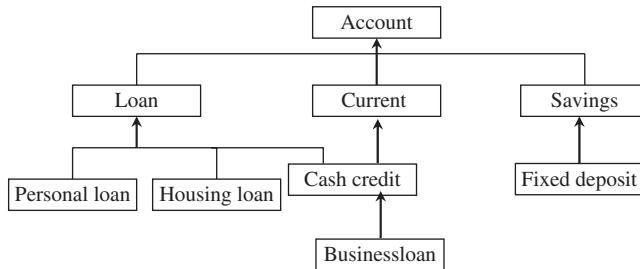


Figure 7.8 Hierarchy of a class account

7.2 RELATIONSHIPS

Just as Entity Relationship Diagram (ER Diagram) helps in the design of a database in traditional methodologies, the identification of relationships greatly helps in the design of objects. Let us consider the example of the banking system to explain the concept of relationships. The ‘Account’ is one of the main classes in the banking system. There can be three types of accounts such as loan account, current account and savings account. Hence, three types of classes ‘Loan’, ‘Current’ and ‘Savings’ can be derived from the class ‘Account’. The hierarchy of class account is shown in Figure 7.8.

7.2.1 Is-A Relationship

This relationship specifies the features of *inheritance*. Inheritance has already been discussed in an earlier section. Here, a class can inherit the properties or features of another class.

For example, there is ‘Is-A’ relationship between class ‘Loan’ and class ‘Account’ (Loan Is-A Account). The relationship is shown by a relationship diagram in Figure 7.9.

A triangular head of arrow (as per the UML notations) represents inheritance. These diagrams can even represent multilevel inheritance. As Cash Credit is derived from the class ‘Loan’, class ‘CashCredit’ is a ‘Loan’. The entire hierarchy of multilevel inheritance is shown in Figure 7.10.

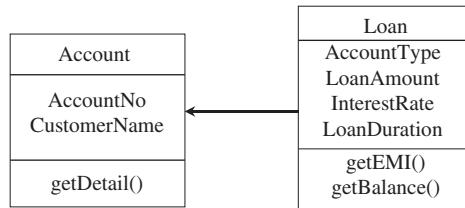
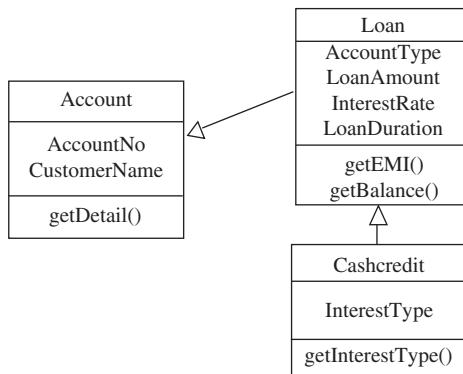
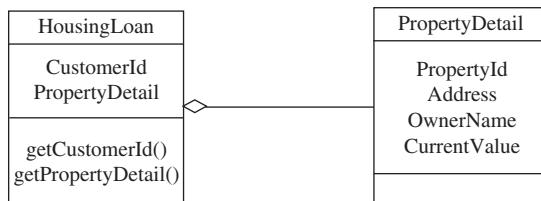


Figure 7.9 Is-A relationship between Loan and Account

7.2.2 Has-A Relationship

This relationship is termed as *Aggregation*. Here, a class contains another class as its member.

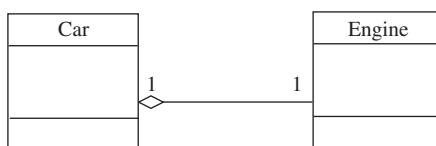
For example, the class ‘HousingLoan’ has ‘PropertyDetails’ as a member variable. The ‘Has-A’ relationship is represented by a diamond headed line in UML. This relationship is shown in Figure 7.11. The diamond sign in the diagram represents relationship ‘HousingLoan Has-A PropertyDetails’. In other words it means class ‘PropertyDetails’ is a part of class ‘HousingLoan’.

**Figure 7.10** Is-A relationship in multi-level inheritance**Figure 7.11** Has-A relationship

Aggregation can be of different types based on the multiplicity of relationships. This is specified by the term '*Cardinality*'. The Cardinality of a relationship specifies how many instances of one class may relate to a single instance of an associated class. The notations for the various relationships are given below:

| Notation | Meaning |
|----------|-----------------------------|
| 1 | One only |
| * | Many (more than one always) |
| 0..1 | Zero or One |
| 0..* | Zero or Many |
| 1..* | One or Many |

A car can have one and only one engine. Therefore, the aggregation between the classes 'Car' and 'Engine' may be seen in Figure 7.12.

**Figure 7.12** One-to-one relationship between classes

A customer can have zero or more loan accounts. Therefore, the aggregation between the classes ‘customer’ and ‘loan account’ may be seen in Figure 7.13.

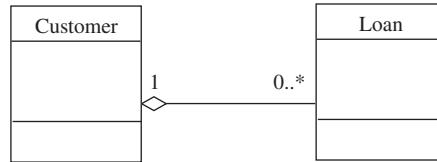


Figure 7.13 One-to-many relationship between classes

7.2.3 Uses-A Relationship

This relationship represents Association. Association is the relationships between objects and classes. Here, objects interact with other objects. It may include creation of another type of object or method invocation (Message passing) on an existing object. For example, LoanAdvisor creates a Loan object for a new loan and LoanAdvisor invokes a method on Loan object. By default, association is bi-directional and is denoted by a simple line as shown in Figure 7.14.

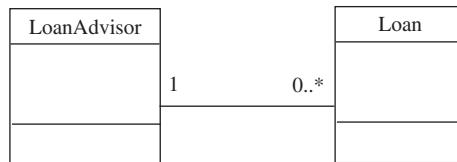


Figure 7.14 Uses-A relationship between classes

It may be noted that sometimes association can also be uni-directional. In that case it is represented by open arrow (\rightarrow). These are explained in UML diagrams.

7.3 SOME MORE CONCEPTS

Besides the key concepts discussed in previous sections, there are some more concepts, which may be useful in the OO systems. These are briefly described below:

7.3.1 Object Identifier

Whenever an object is created, it is uniquely identified in the system by an *object identifier* (OID) or sometimes as *unique identifier* (UID). An object’s name, location or key may be changed, but the OID remains the same. Even if the object is removed, the OID is never deleted. OID is never changed or deleted even if the object’s name, its location or key changes of the object itself is deleted.

7.3.2 Object References

Like other programming languages, the object system also maintains a reference known as object references. Normally, these are maintained through UIDs. For example, a customer Suman has a loan of Rs. 50,000/- . Hence, the instance of customer will have reference to the instance of loan (i.e. object Suman **has-a** object Loan). Similarly, the instance of loan will also refer to the object Suman (i.e. object Loan **belongs-to** object Suman). This is shown in Figure 7.15.

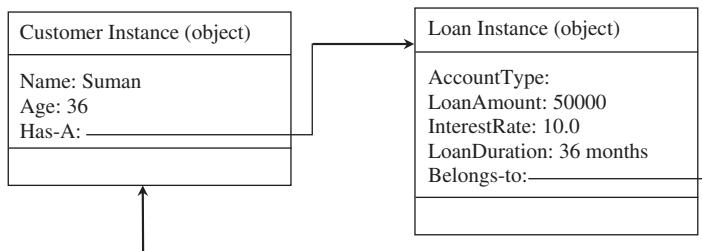


Figure 7.15 Object references among objects

7.3.3 Object Persistence

Normally, an object exists for the duration of the process in which it is created. However, objects can be made to persist for a longer duration than their usual lifetime. This is known as object persistence. The object can persist beyond the application session. A file or database provides support for objects to persist for a longer duration.

7.3.4 Metaclasses

Metaclass is the class of all the classes of a system. This provides services to the application program by returning a set of all methods, instances and parents for review or change. Generally, the compiler uses the metaclass. Therefore, all objects are instances of a class and all classes are instances of metaclass. The metaclass can be an instance of itself.

7.4 MODELING TECHNIQUES

The concept of *modeling* is quite old. However, in the context of OO software development, there are many methodologies developed since the 1980s. The three notables among them are:

1. The initial model for the OO design given by Booch
 2. The ‘Object Modeling Technique (OMT)’ developed by Rumbaugh and his team in General Electric company
 3. The ‘use-case’ diagram introduced by Jacobson

7.4.1 Booch OO Design Model

Bloch's OO design model is widely used in the OO methodology. It uses six types of diagrams to depict the underlying design as given below:

1. Class Diagrams
 2. Object Diagrams
 3. State Transition Diagrams
 4. Module Diagrams
 5. Process Diagrams
 6. Interaction Diagrams

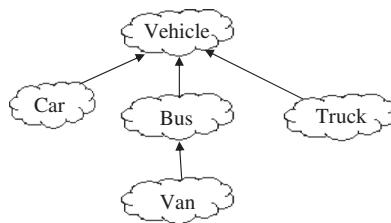


Figure 7.16 Inheritance relationship between classes under Booch notation

Booch is the first person to show OO concepts through notations and diagrams. The inheritance example of vehicle as seen in Figure 7.6 may be examined. The classes ‘Car’, ‘Truck’ and ‘Bus’ can inherit from the class ‘Vehicle’, whereas class ‘Van’ can inherit from ‘Bus’ as shown in Figure 7.16.

Booch used cloud symbols to represent classes. Booch is often criticized for his large set of symbols. Details of Booch notations are not given in this book. The other pioneers of the OO methodology used their own sets of notations. Hence, to avoid confusion, there was a feeling of the need to device a standard set of notations. Accordingly, Booch, Rumbaugh, Jacobson and others devised a standard notation called ‘Unified Modelling Language (UML)’. These UML notations are generally considered a standard in the OO methodology

7.4.2 Rumbaugh's Object Modeling Technique

Rumbaugh's Object Modeling Technique (OMT) is a complete OO development method. The OMT methodology consists of building a model of an application domain and then subsequently adding details to it during the design of a system. The various stages of the OMT methodology are given below:

1. Analysis: In this stage, a concise and precise abstraction of what the desired system must do in terms of attributes and operations is visible to the user.
2. System Design: In this stage, the overall architecture and high-level decisions about the system are made.
3. Object Design: This is done in accordance with the overall system design. However, here the focus is on data structure and algorithms needed to implement each class.
4. Implementation: The object classes and relationships developed during object design are finally translated into a particular programming language, database or hardware implementation.

The OMT methodology uses three kinds of models to describe any system. These are:

- (i) Object model
- (ii) Dynamic model
- (iii) Functional model

The object model describes the static structure of the objects in the system and their relationships. It represents the data aspects of the system. For example, Figure 7.17 displays associations between classes ‘Account’ and ‘Transaction’. The relationship is shown by a filled in circle head that represents one-to-many relationship as per Rumbaugh notations. However, details of Rumbaugh notations are not given in this book because UML notations are now accepted as standard in the OO methodology.

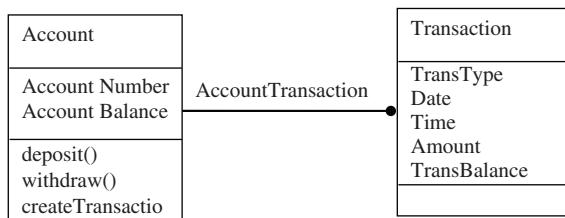


Figure 7.17 Association relationship between classes under Rumbaugh notation

The dynamic model describes aspects of a system concerned with time and sequencing of operations. It consists of various states and events. Each state receives one or more events. The next state depends on the current state as well as the events. For example, in a banking transaction first an account is selected by entering an account number. Then, in the next state account type is selected. Hence, a system can have a number of different states and it passes through from one state to another. The dynamic model represents the temporal, behavioral and control aspects of a system.

The functional model represents the transformational functional aspects of the system. It is represented with the traditional data flow diagrams (DFDs). The primary symbols used here are process, data flow, data store and external entities.

The banking transaction may be depicted through a *Functional* diagram shown in Figure 7.18. The rectangular boxes represent the external entities, the oval shapes represent the processes, the open-ended boxes represent the data stores and the arrows represent the data flows. The exception output from the process is shown by the dotted line. The functional model describes the functions invoked by operations in the object model and the actions in the dynamic model. The functions operate on the data values specified by the object model.

7.4.3 Jacobson's Model

Jacobson introduced the use-case diagram in 1986. Use-case is an effective tool for representing the requirements of a system. It depicts interactions between users and the system. An interaction between a user and the system initiates some action due to which a series of events may take place. A use-case should describe one main flow of events. Use-cases are scenarios for understanding the user requirements.

For illustration purposes, a simple use-case diagram is shown in Figure 7.19. The diagram depicts a transaction of a customer with a banking system.

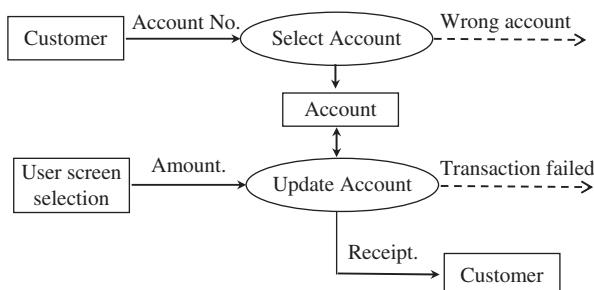


Figure 7.18 Functional diagram for a banking transaction

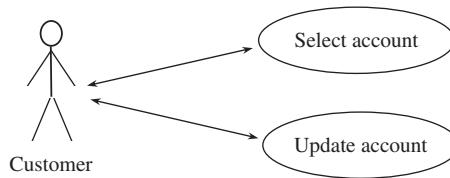


Figure 7.19 Use case for a banking transaction

The customer is the external entity and is referred to as '*actor*' in use-case terminology. The two processes 'select account' and 'update account' are part of the use-case.

A use-case may be used or initiated by another use-case. Such use-cases may not have any actor and are referred to as '*abstract use-case*'. For example, one abstract use-case print can be used by update account to print receipts as shown in Figure 7.20.

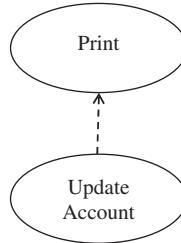


Figure 7.20 Abstract use case print

A technique to extract abstract use-cases is to identify *abstract actors*. An abstract actor typically describes a *role* that should be played by the system. When different actors play similar roles they may inherit a common abstract actor. For example, in Figure 7.19 the banking clerk can also withdraw money. Since the system provides every customer a receipt of the transaction, the abstract actor can be termed for this purpose as receipt receiver, which may be seen in Figure 7.21.

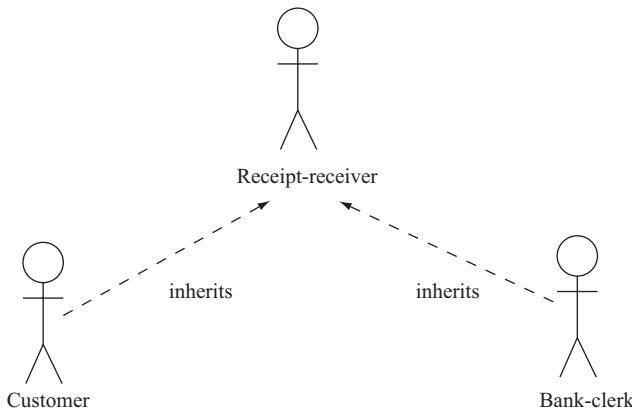


Figure 7.21 Abstract actor receipt receiver

The use-case model mainly employs two types of relationships, namely (i) *Uses* and (ii) *Extends*. The ‘*Uses relationship*’ depicts the reuse of behavior of a use-case by other use-cases. The ‘*Extends relationship*’ extends the functionality of the original use-case. It is similar to derived class. The use-case contains the following description of a system:

1. How and when the use-case begins and ends
2. The interaction between the use-case and its actors, including when the interaction occurs and what is interchanged
3. How and when the use-case will need data stored in the system or will store data in the system
4. Exceptions to the flow of events
5. How and when the concepts of problem domain are handled

Ivar Jacobson had also founded a software company in Sweden by the name Objectory Systems in 1987. In 1991, this company became a subsidiary of Ericsson and was renamed as ‘Objectory AB’. Jacobson along with others in Objectory AB developed a complete set of tools required for OO software engineering. This was called object factory for software development and is often referred to as *ObjectOry* or ‘*Objectory*’. Objectory AB was acquired by Rational Software Corporation in 1995.

The Objectory is a set of models used in OO software development. It is built around several models as given below:

1. Use-case model
2. Requirement model
3. Domain object model
4. Analysis model
5. Design model
6. Implementation model
7. Test model

The Requirement model specifies all the functionality of the system. The objects of the real world are mapped into the Domain object model. The Analysis model is the basis of the system’s structure. It presents how the implementation should be carried out. The Design model tries to adapt the actual implementation environment. The Implementation model does the actual implementation of the system through source code.

The Requirement and Analysis models are used for analysing the system, whereas the Design and Implementation models are used for the construction of the system. The Test model aims to verify the developed system. It consists of test plans, specifications and reports. However, all the models are built around the Use-case model. The Use-case model is part of the Requirement model and is also the basis for the OO analysis, design, implementation and testing as shown in Figure 7.22.

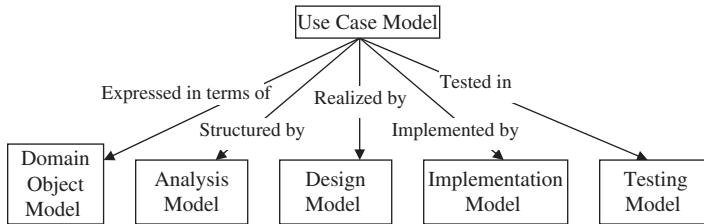


Figure 7.22 Models of objectory showing use-case as the core model

7.5 THE UNIFIED APPROACH TO MODELING

The Unified Approach (UA) is the methodology for software development, based on methodologies by Booch, James Rumbaugh and Jacobson. UA has the following features:

- Just like Jacobson's model, the UA is also a Use-case driven software development methodology. In UA modeling, all the models are built around the use-case model.
- UA utilizes a standard set of tools for modeling.
- The OO analysis is done by utilizing use-cases and object modeling.
- It favors repositories of reusable classes and emphasizes on maximum reuse.
- UA follows a layered approach to software development.
- UA is suitable for different development lifecycles such as Incremental development and prototyping.
- UA favors continuous testing throughout the development lifecycle.

The standard set of tools used in UA is called the Unified Modeling Language (UML). UML does not specify a methodology of what steps to follow to develop an application; that would be the task of UA. These are dealt with in subsequent chapters on OO Analysis and OO Design.

7.6 UNIFIED MODELING LANGUAGE

Booch, Rumbaugh and Jacobson assisted by others at Rational Software Corporation collaborated to combine the best features of their OO methods into a unified method. Thus, a Unified Modeling Language (UML) was developed jointly by them in 1997.

UML is a set of tools that was accepted by the Object Management Group (OMG) as standard for modeling in OOSD. OMG is a consortium originally aimed at setting standards for distributed OO systems. It was founded in 1989. Presently, over 800 companies, including Hewlett-Packard, IBM, Sun Microsystems, Apple Computer, American Airlines and Data General are members of OMG.

UML comprises nine types of diagram. The UML diagrams can capture the five views of a system. These are listed in Table 7.1.

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very

different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

Structural View: It defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

Behavioral View: It captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Table 7.2 UML diagram for different views of a system

| Views of a System | UML Diagram |
|------------------------------|---|
| 1. User Model View | 1. Use-case Diagram |
| 2. Structural Model View | 2. Class Diagram 3. Object Diagram |
| 3. Behavioral Model View | 4. State Chart Diagram 5. Activity Diagrams 6. Sequence Diagram 7. Collaboration Diagram |
| 4. Implementation Model View | 8. Component Diagram |
| 5. Environmental Model View | 9. Deployment Diagrams |

Implementation View: It captures the important components of the system and their dependencies.

Environmental View: It models how the different components are implemented on different pieces of hardware.

The UML analysis model focuses on User model and Structural model views, whereas UML design model addresses the Behavioral model, Implementation model and Environmental model views of the system. UML is now accepted as a standard modeling tool in object-oriented analysis and design (OOAD). These are dealt with in greater detail in subsequent chapters.

SUMMARY

OO software development gathered momentum in the 90s and today most of the software developments are done through this approach. Objects are the basis of OO software development. Data and behavior of a real world object are the building blocks of the OO approaches. Today most of the big and complex systems are mainly data oriented and thus require OO approach.

An object is anything that exists in the real world, such as a person, a place or a thing. It can be any noun or noun phrase, either physical or conceptual. A class is a group of similar objects. It is a blueprint that is used to create objects. A class is a set of objects that share a common structure and common behavior. A single object is simply an *instance* of a class. In object technology, object behavior is described

by methods or functions. However, to invoke those methods, one object has to pass an instruction or message to another object.

Inheritance, abstraction, encapsulation and polymorphism are some important concepts of OO methodology. There are three relations such as is-a, has-a and uses-a relation between various objects.

OO software development is based on OOAD. UML is a standard documentation tool used in OOAD methodology. However, UML is based on many methodologies developed since the 1980s. The initial model for the OO design given by Booch, the 'OMT' developed by Rumbaugh and the 'use-case' diagram introduced by Jacobson are the three basic building blocks of UML.

EXERCISES

1. Distinguish between object and class.
2. What do you mean by *metaclass*?
3. Identify objects in a banking system.
4. Define Inheritance, Abstraction, Encapsulation and Polymorphism.
5. With the help of a suitable example, explain how the inheritance feature of the OO paradigm helps in code reuse.
6. What is the difference between method overloading and method overriding? Explain with suitable example.
7. List out the differences between an OO language like C++ and a procedural language like C.
8. Give an account of various relationships in the OO paradigm.
9. Explain how Uses relationship and Extends relationship are used in use-case diagram.
10. List the various stages of Rumbaugh's OMT methodology.
11. Name the three kinds of models used to describe any system in OMT methodology.
12. List various models around which the Objectory is built. Discuss how the use-case model is related to other models of Objectory.
13. List the features of UA for the development of software.
14. Write various views of a system and the corresponding diagram used for depicting those views in UML.
15. Give an account of the best features of Booch, Rumbaugh and Jacobson methodologies.

This page is intentionally left blank.

OBJECT-ORIENTED ANALYSIS

This chapter describes the basics of Object-oriented Analysis methodology. It covers the following topics:

- *Analysis using Use-cases*
- *Activity Diagrams*
- *Interaction Diagrams*
- *State Chart Diagrams*
- *Class Classification Approaches*
- *Identification of Relationships, Attributes and Methods*

Also, the various diagrams such as use-case diagram, class diagram and object diagram are drawn. A detailed case study is presented at the end of the chapter.

The Object-oriented (OO) methodology consists of Object-oriented Analysis (OOA) and Object-oriented Design (OOD). In OOA, object-modeling techniques are applied to analyze the functional requirements of a system. In OOD, the analysis models are elaborated to produce implementation specifications. OOA focuses on '*what the system does*', whereas OOD focuses on '*how the system does it*'. This chapter describes the tools and techniques of OOA.

8.1 USE-CASE MODELING

Participation and involvement of users are very important to correctly determine the functional requirements of a system. The key functions are generally explained by the users, which are then observed and concluded by the system developers. Since people from different backgrounds are involved in determining the functional requirements, it is important that the functional requirements be described in a manner that can be easily understood by all concerned. The following terms are used in the context of users' view of a system.

- *User Stories*—written by the customers describing things that the system needs to do for them
- *Scenario*—an example of how the system is used in terms of a series of interactions between the user and the system
- *Use-case*—an abstraction that describes a set of scenarios

Use-case is an effective tool to capture, represent and analyze the functional requirements of a system. Use-cases provide the external view of what the system does, and feed into OOA and design. The use-case model specifies the behavior of a system.

A system gets input from some external entities. It provides output to some external entities. Hence, the system interacts with various external entities. In use-case terminology, any entity that interacts with the system is termed as actor. An actor is usually a role played by a human user. However, it can also be any entity such as a third-party organization or another computer system. A single user may play different roles at different times, and thus be represented as different actors. An actor in a use-case diagram is analogous to an external entity in a data flow diagram.

The use-case approach requires the analyst to determine all the potential actors involved in a system. Actors are external to the system and make use of it. An actor makes use of a system in different ways. Each use-case describes a sequence of transactions between one or more actors and the system. A use-case may involve a number of actors, and an individual actor may also make use of several use-cases.

The use-case is graphically depicted by a use-case diagram. In the use-case diagram, the use-cases are represented by ovals (ellipse) and actors are represented by stick-men. Let us consider interactions between a banking system and its customers. A customer withdraws money from the bank. He can also deposit money in the bank. Hence, we can represent:

- ‘Customer’ as an ‘Actor’
- ‘Withdraw money’ as a ‘Use-case’
- ‘Deposit money’ as another ‘Use-case’

This is depicted by the use-case diagram in Figure 8.1.

Suppose in the banking system, the money can be withdrawn by the customer through the Automatic Teller Machine (ATM), but money is deposited through the cashier, then the cashier can be also considered as an actor. Hence, a use-case may have multiple actors as shown in Figure 8.2.

And the use-case model is built according to how the system is used. More use-cases can be added based on different purposes for which the actors interact with the system. For example, the customers also check their account balance and they can also borrow a loan from the bank. Hence, use-cases for these purposes can also be added to the use-case diagram.

One use-case may also interact with another use-case. For example, use-case ‘Withdraw money’ would use use-case ‘Check account balance’ for authorizing payment.

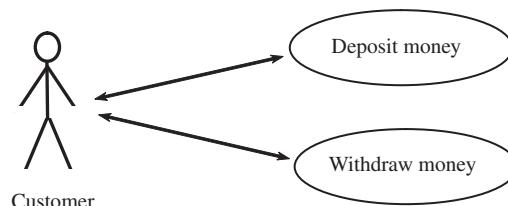


Figure 8.1 A simple use-case diagram

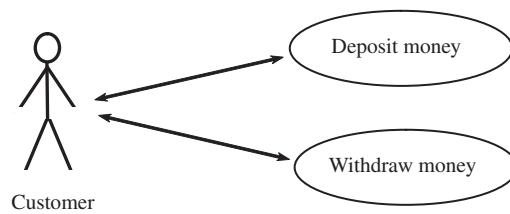


Figure 8.2 A use-case diagram having two actors

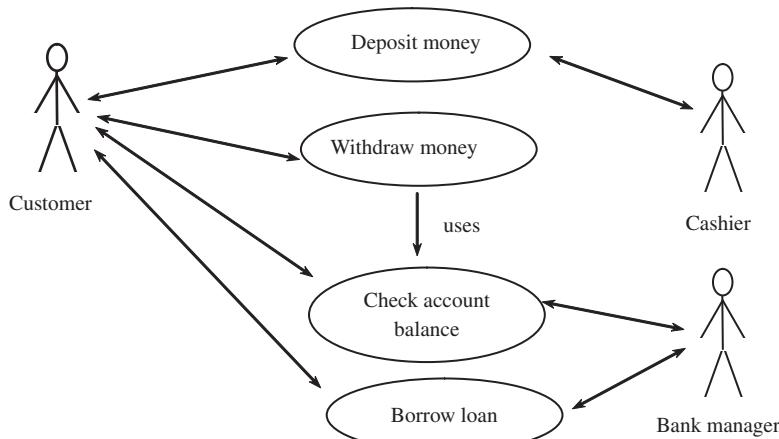


Figure 8.3 Use-case diagram of a banking system

For sanction of loans, the bank procedure may require the approval of the bank manager. The bank manager may need to check the bank transaction of the customer before he sanctions the loan. Hence, the bank manager can also be considered as an actor.

In this way various actors and use-cases may be added and interlinked to depict the overall (high level) users' view of the system. It depicts various users of a system and what service the system provides them. The overall use-case diagram for the complete bank system is depicted in Figure 8.3.

The use-case diagram is an effective tool for depicting a process. A process consists of a number of activities. The use-cases should be named properly so that the function performed by the use-cases can be known from their names. The names should preferably be in the form of a verb followed by a noun (e.g. deposit money, withdraw money).

Use-case diagram is not intended to give a detailed account of how the services are provided. It merely lists what the system does. By keeping out the details, the requirements of a system are depicted in a concise manner. Use-case diagrams are the starting points, which can be elaborated at a later stage for detail design of the system.

Use-case analysis provides a method to represent the ways in which a user uses a system. Representation of system requirements through the use-case diagram facilitates better understanding of a process. It facilitates users and system analysts to work together for determining the system's requirements.

8.1.1 Development of Use-Case

Development of use-cases is usually done in an iterative manner. Hence, there is no need to do it perfectly in the first instance. The use-case can always be refined at a later stage. For illustration purposes, let us consider development of use-case for the online trading (buying/selling) process through the website. The typical steps for development of the use-case model may be as follows:

Identification of Actors: There can a number of actors for the online e-commerce site, such as buyers, sellers, suppliers, wholesalers, auditors, customer service etc. Out of these, some actors may be more important than the others based on the importance of their roles. In initial stage, it is not necessary to identify all the possible actors. The use-case model can be refined in subsequent stages and more actors can be added later. The important actors should be identified first. For example, buyer and seller are two most important actors of an online e-commerce site.

Identification of Actors' Behavior: After identifying the actors, the next step is to outline the activities of each actor. The activities performed by an actor are termed as behavior. In our example, the possible behavior of each actor, i.e. buyers and sellers, is listed in Table 8.1.

In the initial stage, it is not necessary to capture all the possible behavior of identified actors. The use-case model can always be refined later for inclusion of behavior.

Relationship among Use-cases: There may be some behavior that is common to two or more actors. For example 'create account' and 'search item from item-list' are common behavior of actor_buyer and actor_seller. Instead of having all these duplications, a more general actor can be identified, who has the common behavior. Hence, the two actors, e.g. buyer and seller, can 'inherit' this behavior from the new actor. The actors and their behavior listed in Table 8.1 can be drawn taking a generic actor into account. This is shown in Figure 8.4.

The actor buyer and actor seller have an extend relationship with the generic actor. There are other types of relationships as well such as (1) Association relationship, (2) Include relationship, (3) Extend relationship and (4) Generalization relationship.

Relationship aspects can be considered in Use-case modeling.

Use-case Index: After producing the initial list of use-case actors and their behavior, it is useful to create the use-case index. The use-case index lists all the use-cases and their attributes such as actors, scope, complexity, status and priority.

Table 8.1 Actors and their behavior

| Actor | Behavior |
|--------|-------------------------------|
| Buyer | Create account |
| | Search item from item-list |
| | Place price bid for item |
| | Place purchase order for item |
| Seller | Create account |
| | Search item from item-list |
| | Create auction for an item |
| | Ship the item to buyer |

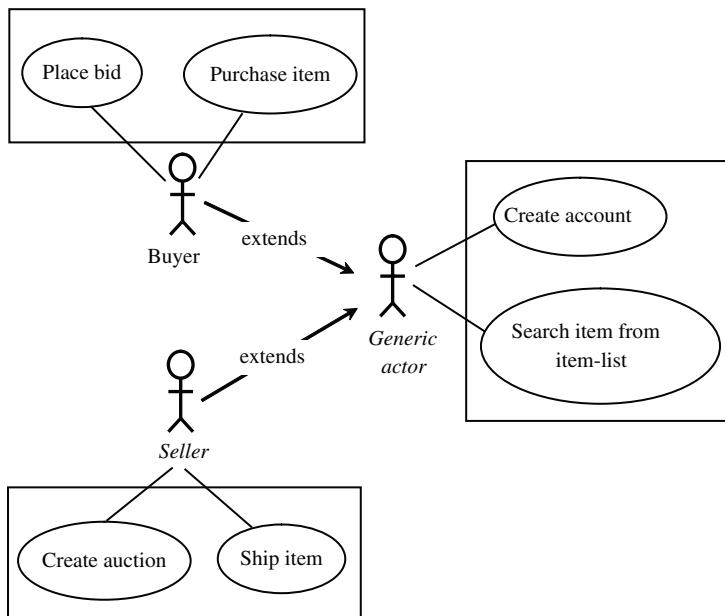


Figure 8.4 Buyers and sellers extending to generic actor

The use-case document contains the detail of a use-case. It lists the key components of the use-case as given below:

- (i) Use-case Number: ID to represent the use-case
- (ii) Use-case Name: The name of the use-case. The name should be short and convey the purpose of the use-case.
- (iii) Application: The system or application that the use-case pertains to.
- (iv) Use-case Description: Elaboration on purpose of the use-case. For example, the description of the use-case 'place bid' may be as under:
After the buyer of the e-commerce website has identified an item for purchase, he places a bid for the item with the intent of winning the auction.
- (v) Primary Actor: The main actor that the use-case represents.
- (vi) Precondition: Preconditions that must be met before the use-case can start.
- (vii) Trigger: Event that triggers the use-case.
- (viii) Basic Flow: The basic flow of a use-case represents the most important course of events or what happens most of the time. It is sometimes referred to as the 'Happy Day Scenario' because it is what occurs when everything goes well when there is no error or exception. Once the basic flow is understood, it becomes much easier to comprehend the exception cases. The exceptions are handled in the 'Alternate Flows' section.
- (ix) Alternate Flows: These are significant alternatives and exceptions. The alternate flows provide the following information:
 - An error flow corresponding to a basic flow
 - An additional flow (not necessarily an error) that could happen

A few examples of alternate flows are as follows:

- While a customer is placing an order, the credit card is not accepted.
- While a credit card is accepted for making payment, the user session is timed out.

For illustration purposes, the use-case elements of use-case ‘place Bid’ are shown in Table 8.2.

It is much easier to learn a system through use-cases than a narrative requirements description. The use-case is not only a high-level description, a walk through a living scenario but also a formal requirement document.

Table 8.2 A Use-case Document

| General Information | | |
|----------------------------|--|---|
| Use-case number | 1 | |
| Use-case name | Place bid | |
| Goal in Context | Buyer issues an order directly through the website, expects the goods to be shipped and to be billed | |
| Scope and Level | Website, Summary | |
| Preconditions | Buyer is already registered having a valid login ID | |
| Success Condition | Buyer gets the goods, and the website gets the payment | |
| Primary Actor | Buyer | |
| Secondary Actor | Credit card company, bank, shipping service | |
| Trigger | Receipt of purchase request | |
| Description | Step | Action |
| | 1 | Buyer makes a purchase request |
| | 2 | Website captures details of buyer and goods |
| | 3 | Website creates order |
| | 4 | Buyer pays with credit card |
| | 5 | Website registers order for shipment |
| Extensions | Step | Branching Action |
| | 3a | One or more ordered items are out of stock: 3a1. Renegotiate order |
| | 4a | Buyer pays directly with credit card: 4a1. Credit card has expired |
| | 5a | Buyer returns goods due to poor quality: 7a. Handle returned goods |
| Subvariations | Step | Branching Action |
| | 4 | Buyer may pay by transfer through bank |
| Related Information | | |
| Priority | Top | |
| Performance | 5 minutes per order | |

| | |
|-----------------------|--|
| Frequency | 200 orders per day |
| Open Issues | What if the website has part of the order? What if the credit card is stolen? |
| Due Date | release 1.0 |
| Super-ordinates | Manage customer relationship |
| Subordinates | Create order |
| Any other information | |

8.1.2 Use-case Realization

The purpose of use-case is to:

- identify the classes that perform a use-case's flow of events
- distribute the use-case behavior to those classes, using use-case realizations
- identify the responsibilities, attributes and associations of the classes

Some of the main points of this approach are as follows:

Box 8.1: Highlights of use-case analysis

- Modelling with use-cases is recommended aid in finding the objects of a system
- Technique is used in the unified approach
- Use-cases are employed to model the scenarios in the system and specify what external actors interact with the scenarios
- Once the system has been described in terms of its scenarios, then examine the textual description or steps of each scenario to determine what objects are needed for the scenario to occur
- The process of creating sequence or collaboration diagram is a systematic way to think about how a use case(scenario) can take place; and by doing so, it forces you to think about objects involved in your application

A use-case realization describes how a particular use-case is realized within the design model in terms of collaborating objects. For each use-case realization, we can have the following points:

- Find classes from use-case behavior
- Distribute use-case behavior to classes

A use-case realization in the design model can be traced to a use-case in the use-case model. A realization relationship can be drawn from the use-case realization to the use-case it realizes. A use-case realization can be presented using a set of diagrams:

- (i) **Interaction diagrams** are used for describing how the use-case is realized in terms of collaborating objects. They comprise (1) Sequence and/or (2) Collaboration Diagrams. They model the detailed collaborations of the use-case realization.
- (ii) **Class diagrams** can be used to describe the classes that participate in the realization of the use-case, as well as their supporting relationships. These diagrams model the context of the use-case realization.

Use-cases are employed to model the scenarios of the system. The scenarios are described through a sequence of steps. Developing scenarios also requires us to think about class methods. We will discuss about the class methods in a subsequent section. The use-case analysis is a unified approach in order to find classes. It is the process that involves examining the use-cases to identify objects and classes.

- Scenarios are detailed and shown graphically in interaction diagrams.
- Classes are grouped into packages.
- Class diagrams are created.

8.2 ACTIVITY DIAGRAM AND STATE DIAGRAM

Activity Diagrams describe the workflow behavior of a system. They describe the state of activities by showing the sequence of activities performed. They can describe both the sequential and concurrent groups of activities.

Activity Diagrams are constructed by using the following symbols:

- Rounded rectangles represent activities.
- Diamonds represent decisions.
- Bars represent the start (split) or end (join) of concurrent activities.
- Black circles represent the start (initial state) of the workflow.
- Encircled black circles represent the end (final state).

In Activity Diagrams, the nodes represent the execution of an activity and the edges represent the transition on the completion of one set of activities to the commencement of a new set of activities. Arrows run from the start towards the end and represent the sequence in which the activities are performed.

For illustration purposes, the Activity Diagram of a buying process in a super market with customers is shown in Figure 8.5. The activities will consist of the customers collecting goods and paying through the cash-card. These are the two activities done by the customer (actor-1). To enable a customer to do these activities, many intermediate activities are performed by a counter-clerk (actor-2).

These two sets of activities are shown by separating the activities by a solid line. The activities shown at the left-hand side are done by the customer (actor-1) and the activities shown at the right-hand side are done by the counter-clerk (actor-2).

The Activity Diagram can be regarded as a form of flowchart. A typical flowchart technique lacks constructs for expressing concurrency. This drawback is overcome by the join and split symbols in the Activity Diagram. Activity Diagrams capture high-level aspects of activity.

State Diagrams depict the occurrences of events and corresponding states of an object. In State Diagrams, the nodes represent the states and the arrows represent occurrences of events. The concurrent states are represented by synchronization bars. Synchronization bars can be put in both Activity Diagrams and State Diagrams. A combination of many states is called as composite state. Composite states can be of two categories: (1) Sequential composite states and (2) Concurrent composite states.

Sequential composite states and Concurrent composite states are shown in Figures 8.6 and Figure 8.7, respectively.

In State Diagrams, a filled circle represents the start state. A filled circle within another normal circle represents the end state. Figure 8.7 has two synchronization bars one after the start state and another before the end state.

Let us consider another example of the check out process in an e-commerce site. The process is illustrated by drawing a State Diagram as shown in Figure 8.8.

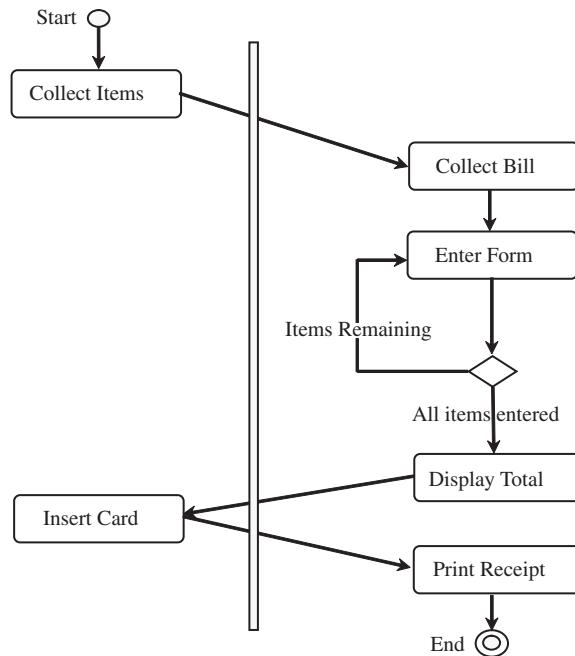


Figure 8.5 Activity diagram for a cash counter in a supermarket

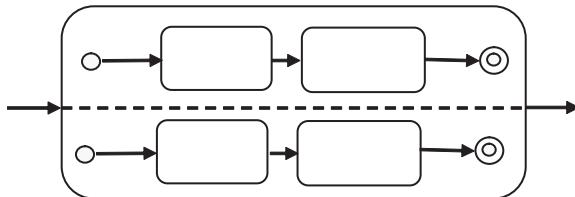


Figure 8.6 Sequential composite states

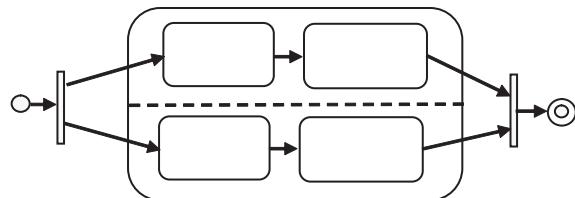


Figure 8.7 Concurrent composite states

The check-out process in an e-commerce site can have two concurrent states, one to compute the total cost and another to process the credit card by fetching the card number and address. After the second synchronization bar, the card is verified and if not approved another card is fetched. After the card is approved, the shipping order is placed.

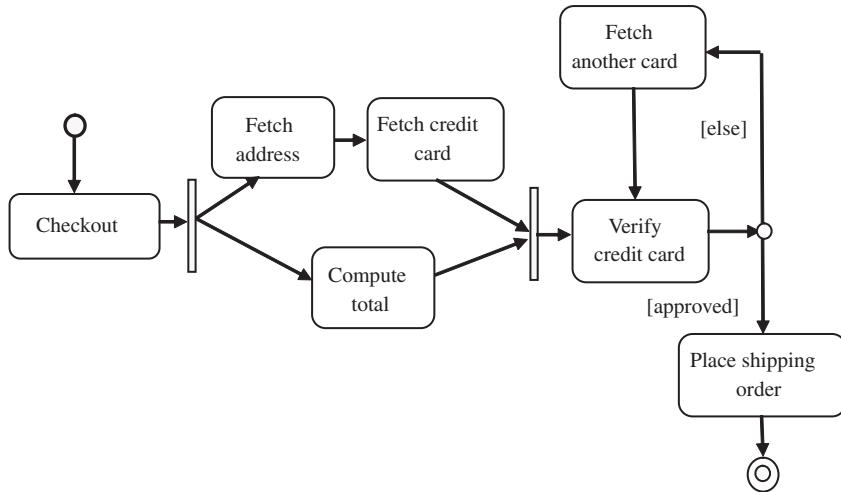


Figure 8.8 State diagram for checkout

8.3 INTERACTION DIAGRAMS

Interaction Diagrams are models that describe how a group of objects collaborate in some behavior. They come in two forms:

1. Sequence Diagram
2. Collaboration Diagram

8.3.1 Sequence Diagram

A Sequence Diagram shows a set of messages arranged in time sequence. Each classifier's role is shown as a lifeline, i.e. the vertical line that represents the role over time through the entire interaction.

Messages are shown as arrows between lifelines. A Sequence Diagram can show a scenario, i.e. an individual history of a transaction. Its basic use is to show the behavior sequence of a use-case.

Let us consider that the process of buying products through a website is described through use-case. Once a user places an order through the shopping cart, first he gets a bill. Based on the bill, he makes the payment. The `get_bill` use-case is initiated by the customer on his computer. A shopping cart is created by the user at the first instance. The shopping cart sends a request message to get the total bill amount from the database.

The sequence of activities of a process is effectively depicted by Sequence Diagram as shown in Figure 8.9.

8.3.2 Collaboration Diagram

A Collaboration Diagram models the objects and links that are meaningful to an interaction. A classifier's role describes an object whereas an association role describes a link within collaboration. A Collaboration Diagram for searching a database through a browser is shown in Figure 8.10.

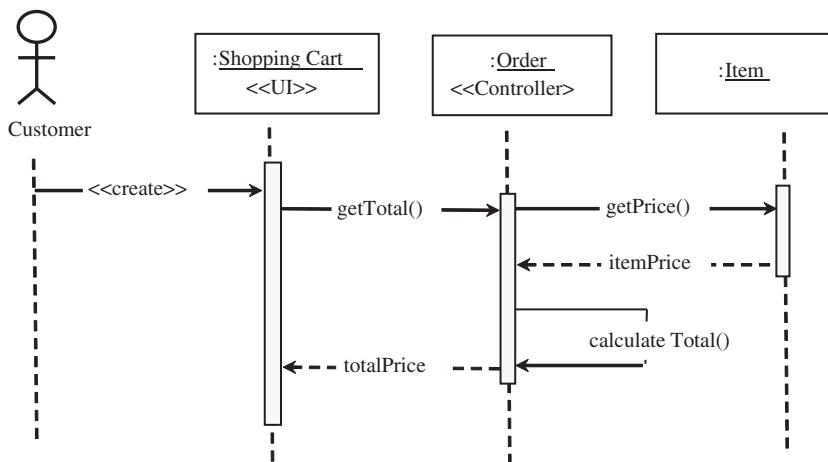


Figure 8.9 Sequence diagram of ordering process through website

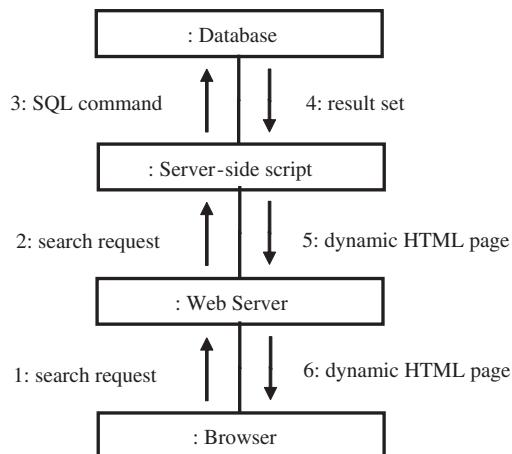


Figure 8.10 Collaboration diagram for making a search

In this diagram, the messages are shown as arrangements attached to the relationship lines connecting classifier roles. The sequence of messages is indicated by sequence numbers preceded to message descriptions.

8.4 TYPES OF CLASSES

The OO analysis starts by identifying the classes from use-cases. The identification of classes can be done by considering three different perspectives of the system. These three perspectives are:

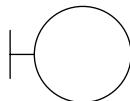
- the boundary between the system and its actors
- the information the system uses
- the control logic of the system

Identification of classes means that the classes should be identified, named and described briefly.

In the OO paradigm, the objects are classified into three types, viz., interface or boundary objects, entity objects and control objects. The corresponding classes are interface classes representing interface objects, entity classes representing entity objects and control classes representing control objects.

8.4.1 Interface Class

The interface classes contain all functionalities specified in the use-case descriptions of the system environment. The actors communicate with the system through objects belonging to the interface class. These are intermediates that facilitate interaction between the system and the actors. The task of an interface object is to translate the actor's input to the system into events in the system. Interface objects can, in other words, describe bi-directional communication between the system and its users. Interface classes are represented by the symbol shown below.



One interface class should be identified per actor/use-case pair. For illustration purposes, consider online registration to a course by a student through a University website. The student can do online registration in two ways. He can look into the course catalog and select the course for registration. Alternatively, he can open the registration form and enter the course for registration.

Hence, the system can have two interface classes, say 'RegisterForCoursesForm' and 'CourseCatalogSystem'. The system is depicted through 'Class Classification Diagram' as shown in Figure 8.11.

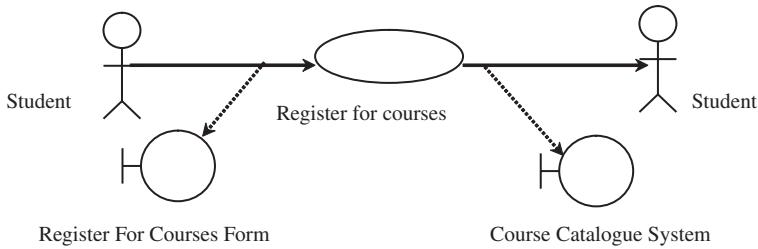
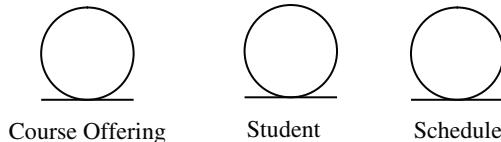


Figure 8.11 Class classification diagram using interface classes

8.4.2 Entity Class

Similar to interface classes, entity classes are also identified from use-cases. Entity classes are represented by the symbols as shown below:



Some of the main features of an entity class are given below:

- It represents the key elements that the system manages or what the system consists of. Entity objects are identified as 'nouns'. The elements may be physical or conceptual.
- Its main responsibilities are to store and manage data in the system.
- Entity objects (instances of entity classes) are used to hold data about some phenomenon, such as an event, or some attributes of real-entity (e.g. a person).

Most entity classes are identified early at the beginning of analysis. They are often identified in the problem domain.

However, sometimes it is difficult to decide whether a certain piece of data should be modeled as an entity object or as an attribute. To be able to decide, we must see how the data will be used. Information that is handled separately should be modeled as an entity object, whereas information that is strongly coupled to some other information and never used by itself should be made an attribute of an entity object. In other words, it can be decided based on how the information is handled by the use-cases. Therefore, different systems may form different kinds of entity objects and attributes for same set of information. For example, let us model a car and its owner of a system. There can be various possibilities for modeling this system. In one model, we could have two entity objects, 'Car' and 'Owner'. In another model, we could have just one entity object say 'Car' and 'Owner' as an attribute of car. The same system can also be modeled where 'Owner' is considered as an entity object 'Person' and 'Car' is an attribute. The modeling is generally done based on the use-cases. In the first case, two use-cases could use owner and car independently of each other. In the second case, we use car in association with its owner. In the third case, we only use person in association with a car.

8.4.3 Control Class

The flow of a use-case can normally be divided either into interface class or entity class. However, in some complex use-cases, certain behavior cannot be placed in either class. Such behavior is placed in control classes. These classes can be found directly from the use-cases. As each use-case normally involves interface classes and entity classes, the behaviors that are left out are placed in control classes. Control classes are represented by the symbol shown below.



Control class:

- It models the control behavior specific to one or more use-cases.
- It handles the tasks and the control flows of a system. It represents the dynamics of the system.
- One control class per use-case

Use of control class for the application 'registration of courses' is depicted through the Class Classification Diagram, as shown in Figure 8.12.

The control class is responsible for controlling the process described in the associated use-case.

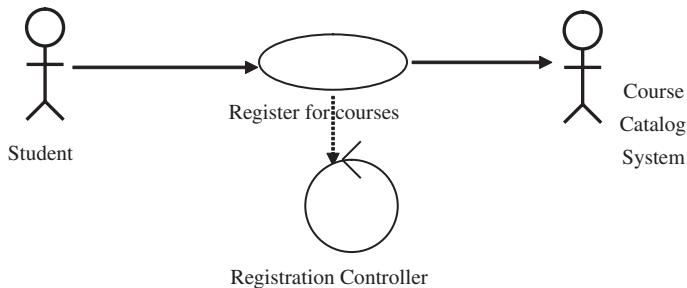


Figure 8.12 Class classification diagram using control classes

8.5 CLASS CLASSIFICATION APPROACHES

Identification of classes is an important step of the OO analysis. The identification of classes is highly subjective and requires ingenuity and experience. There is no such thing as the perfect class structure or the right set of objects. It best comes about through an incremental and iterative process. Seven class classification approaches are discussed in this section. These are:

1. Noun phrase approach
2. Classical approach
3. Function point approach
4. Domain analysis approach
5. Structural approach
6. CRC approach
7. Use-case-driven approach

8.5.1 Noun Phrase Approach

Nouns are considered as the foundation of the OO paradigm. The nouns appearing in the textual description of use-cases/requirements are the probable sources for identification of classes. Hence, the nouns and noun phrases are picked up from the textual description of use-cases. These nouns are probable candidates to form the classes of the system. Similarly, the methods of classes can be identified from the verbs. The methods are discussed later. Some of the classes may be irrelevant, which should be removed. Also, redundant classes, adjective classes and attribute classes should not be considered. The steps for the noun phrase approach are depicted as follows:

Relevant class: Each class must have a purpose and should be clearly defined and necessary. A statement of purpose for each candidate class is formulated.

Irrelevant class: Some classes may fall outside the scope of the problem and do not have a statement of purpose in line with the objective of the system.

Fuzzy class: If the class cannot be determined as either relevant or irrelevant, then it is considered as fuzzy. Here, the analyst uses his ingenuity to decide if it is to be considered as a class or not.

Redundant class: Two classes that express the same information are not kept.

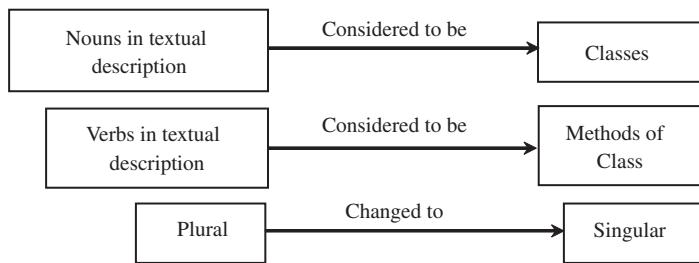


Figure 8.13 Key steps in noun phrase approach

Adjectives class: An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. If the use of the adjective signals that the behavior of the object is different, then a new class is made.

Attribute class: Objects that are used only as values should be defined or restated as attributes and not as a class.

The key steps of the noun phrase approach are shown in Figure 8.13.

Example:

Let us consider an example of a library. The study of the library manual can provide certain rules as follows:

'The library contains books and journals. It may have several copies of a given book. Some of the books are for short-term loan only. All other books may be borrowed by any library member for 20 days. Members of the library can borrow up to four items at a time. However, members of staff may borrow up to eight items at a time. Only members of staff can borrow journals. The system must keep a track of the books and journals being borrowed and returned by enforcing the above rules.'

The underlined portions represent noun phrases. The following noun phrases can be discarded:

1. Library, because it falls outside the scope of our development
2. Short-term loan, because the loan is an event
3. Member of the library, which is redundant with 'library member'
4. Day, because it is a measure not a thing
5. Item, because it is just an entity without any purpose
6. Time, because it is outside the scope of the system
7. System, because it is just another description of the requirements
8. Rule, also another description of the requirements

Thus, out of 13 noun phrases selected earlier, only five are left out.

1. Book
2. Journal
3. Copy of the book
4. Library member
5. Member of staff

Hence, the above five noun phrases may be identified as classes of the library system.

8.5.2 Classical Approach

A number of classical approaches for categorization of classes have been suggested by various researchers. The classes and objects usually come from one of the sources such as tangible things, roles, events, interactions etc. Some sources for identification of classes are given below.

| Actor | Behavior |
|---------------|--|
| People | People who carry out some functions |
| Places | Areas set aside for people or things |
| Things | Physical objects or devices with which the application interacts |
| Organizations | Formally organized collections of people, resources with facilities and capabilities of having a defined mission |
| Concepts | Principles or ideas |
| Events | Things that happen at a given date and time |
| Other systems | External systems with which the application interacts |
| Roles played | Different roles the users play while interacting with the system |

Some of the disadvantages of the common class pattern approach are as follows:

- It is used just for guidance.
- It is loosely bound to user requirements.
- It creates possible naming misinterpretations.

8.5.3 Function Point Approach

Analysis of system behavior can be an effective approach for identification of classes. The meaningful behaviors of the system are studied to identify the entities ‘who initiate the behaviors and who participate in the behaviors’.

Entities that play significant roles in system behavior are recognized as objects of the system and are assigned responsibilities for these roles.

The above concept of behavior is closely related to the idea of function points. A function point is a quantitative measure of size of software in terms of equivalent functions it performs. A business function represents some kind of output, inquiry, input, file or interface. The information system perspective of this definition shows that the idea of a function point generalizes to all kinds of systems. A function point is any relevant outwardly visible and testable behavior of the system.

8.5.4 Domain Analysis Approach

The idea of domain analysis was first suggested as a means to identify the objects, operations and relationships. The following steps may be followed in domain analysis:

- Construct a generic model of the domain in consultation with domain experts.
- Examine existing systems within the domain and represent this understanding in a common format.
- Identify similarities and differences between the systems in consultation with domain experts.
- Refine the generic model to accommodate existing systems.

Overall domain analysis is nothing but using the domain knowledge to find out classes. The potential users who are using the system since long may be considered as domain experts.

The noun phrase approach, classical approach and function point approach are generally applied to the development of a single application. However, sometimes it becomes necessary to identify the classes and objects that are common to all applications within a given domain, e.g. inventory management, railway reservation system. Domain analysis can also help in pointing out the key abstractions in the midst of design. This can prove useful in other related systems as well.

8.5.5 Structural Approach

In this approach, Data Flow Diagram (DFD) is the starting point for identification of classes. The candidate classes may be derived from the Data flows and Control flows. Hence, this approach uses structured analysis as a front-end to OOA and design. This technique is appealing because majority of analysts are skilled in structured analysis. A number of Computer Aided Software Engineering (CASE) tools exist, which support the automation of these methods. However, from the OO perspective the use of structured analysis as a front-end to OOA and design is sometimes discouraged. However, for some organizations, it is the only pragmatic alternative.

8.5.6 CRC Card Approach

Class Responsibility Collaboration (CRC) card was developed by Cunningham, Wilkerson and Beck. It was presented as a way of teaching the basic concepts of OO development. It is a technique used for identifying the attributes and methods of classes. It is developed based on the idea that an object accomplishes certain responsibilities on its own or with the assistance from other objects.

By identifying an object's responsibilities and collaborators, we can identify its attributes and methods. CRC cards are 4" x 6" index cards, on which all information of an object is written. The analyst writes the name of the class in pencil at the top of the card, its responsibilities on the left-hand side and its collaborators on the right-hand side of the card. This is shown in Figure 8.14.

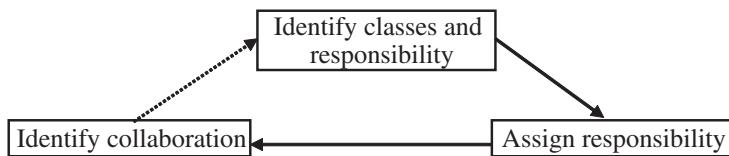
The CRC process comprises the following activities:

1. Identify classes' responsibilities (and identify classes).
2. Assign responsibilities.
3. Identify collaborators.

Classes are identified and grouped by common attributes. This helps to identify the super classes and also the class structure. Next, the responsibilities are assigned among the classes. The idea in locating collaborators is to identify how classes interact with each other. The CRC process is shown in Figure 8.15.

| | |
|------------------|---------------|
| Class name | |
| Responsibilities | Collaborators |
| Attributes | |
| Operators | |
| | |

Figure 8.14 The CRC index card

**Figure 8.15** CRC process

For illustration purposes, let us consider the example of a library. The librarian can have many responsibilities as listed in Figure 8.16.

| Class Librarian | |
|---------------------|----------------|
| Responsibilities | Collaborators |
| Check in book | Book |
| Check out book | Book, Borrower |
| Search for book | Book |
| Know all books | |
| Search for borrower | Borrower |
| Know all borrowers | |

Figure 8.16 CRC index card of librarian class

For discharging some of these responsibilities, the librarian may have some collaborators. In this case, book and borrower are two collaborators. Therefore, we have to make CRC cards for classes book and borrower. The class_book and class_borrower can also have responsibilities. The CRC card for book is shown in Figure 8.17.

| Class: Book | |
|-----------------------|---------------|
| Responsibilities | Collaborators |
| Knows whether on loan | |
| Knows due date | |
| Knows its title | |
| Knows its author(s) | |
| Knows its regd. code | |
| Knows if late | Date |
| Check out | |

Figure 8.17 CRC index card of book class

8.5.7 Use-case-driven Approach

It is better to have a mixed approach for identifying classes. A hybrid approach may be followed to identify classes as shown below:

- Initial classes through Domain knowledge
- Classical approach to guide

- Noun phrase approach to add more classes
- Use-case approach to verify
- CRC approach to brainstorm

8.6 RELATIONSHIP, ATTRIBUTES AND METHOD IDENTIFICATION

In order to have a complete OO analysis, the identified classes should have relationships among themselves and their attributes and methods should be properly identified.

8.6.1 Relationships

The various relationships have been already discussed in Chapter 7, Section 7.2.

- All objects stand in relationship to others on whom they rely for services and control.
- The relationship among objects is based on the assumptions each makes about the other objects, including what operations can be performed and what behavior results.

The three types of relationships among objects are as follows:

- Association
- Super-Sub structure (also known as generalization hierarchy)
- Aggregation and a-part-of structure

Association is the side effect of discovering classes. Some attributes may be associations. However, a 'Dry-run' of use-cases is required to discover more associations. Ternary associations are generally avoided. In order to name associations small letters are recommended and underscore is used to separate words, e.g. emp_task.

There are two types of associations:

- Association represented by a line
- Directional association represented by an arrow

Sometimes the associations can also be declared as classes. For example, in a student-course relationship, the association relationship may be declared as an independent class. Hence, the class 'Takes' is used for the association relationship between class 'Student' and class 'Course' as shown in Figure 8.18.

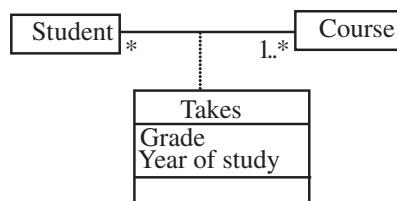


Figure 8.18 Association relation takes is a class

Aggregation is a stronger form of association, in which an object from one class is part-of or is-contained-in an object of another class. There can be four different types of semantics for aggregation possible as shown below:

1. ExclusiveOwns (e.g. Book has Chapter)
 - Existence-dependency
 - Transitivity
 - Asymmetry
 - Fixed property
2. Owns (e.g. Car has Tire)
 - No fixed property
3. Has (e.g. Division has Department)
 - No existence dependency
 - No fixed property
4. Member (e.g. Meeting has Chairperson)
 - No special properties except membership

Aggregation is discovered in parallel with discovery of associations. The litmus test phrases are as follows:

- 'has'
- 'is-part-of'

It can relate more than two classes. The two types of aggregation that UML supports are:

1. Aggregation
 - By-reference semantics
 - Hollow diamond
 - Corresponds to Has and Member aggregations
2. Composition
 - By-value semantics
 - Solid diamond
 - Corresponds to ExclusiveOwns and Owns aggregations

Generalization is another form of association, in which the relationship between super-class and sub-class is established.

8.6.2 Attributes

After finding out the relationships among classes, the next step is to find out attributes and methods of different classes. Identifying attributes and methods is like finding classes. It is an iterative approach.

Attributes are things an object must remember such as color, make, size and cost. Identifying attributes of a system's classes starts with understanding the system's responsibilities. We saw that a system's responsibilities can be identified by developing use-cases and the desired characteristics of the applica-

tions, such as determining what information users need from the system. The following questions help to identify responsibilities of classes and to decide:

- what information about an object should we keep track of
- what services must a class provide

Answering the first question helps to identify the attributes and the second question to identify the methods of a class.

The guidelines for identifying attributes and methods of classes in the problem domain can be made by analyzing use-cases. The other Unified Modelling Language (UML) diagrams such as UML activity and State Diagrams assist in this process by helping in better understanding of class responsibilities. Attributes can be derived from scenario testing by analyzing the use-cases and Sequence/Collaboration Diagram, activity and State Diagrams. The main goal here is to understand the class and its responsibilities. Responsibility is a key issue. The following questions may be asked about an object:

- How is the object going to be used?
- How is the object described in the context of the system's responsibility?
- How is the object going to collaborate with other classes?
- What information does the object need to know?
- What state information does the object need store in its memory?
- What states can the object be in?

The following guidelines for defining attributes can be used to identify attributes of classes:

- Attributes usually correspond to nouns followed by prepositional phrases such as cost of the item.
- The classes should be kept simple. Only the attributes that are really required to define the state of an object should be retained.
- Attributes are less likely to be fully described in the problem statement. Thus, it must be found out from the knowledge of the application domain and the real world. However, only those attributes that are necessary to the design must be included. For example, in a banking system, the initial thought process may be that the customer name, age, weight and address may be important to the class customer in a personal system, but it is not within the scope of the system as there is no scenario in the banking system that needs to keep track of the age and weight of the customer.
- Derived attributes may be omitted. Derived attributes should be expressed as a method.

Attributes usually correspond to nouns followed by progressive phrases such as *cost of* the car. However, derived attributes should be omitted. Age should not be used as it can be derived from date of birth. Thus, derived attributes should be expressed as methods.

8.6.3 Methods

Objects not only describe abstract data but also must provide some services. Methods and messages are the drivers of OO systems. In an OO environment, every piece of data or object is surrounded by a rich set of routines called methods. These methods do everything from printing the object to initializing its variables. Methods usually correspond to queries about attributes and sometimes association of objects. They are responsible for managing the value of the attribute such as query, updating, reading and writing; e.g. for the class car, cost and model can be attributes, getCost(), setCost() can be methods.

CASE STUDY—I: THE ATM SYSTEM OF A BANK

Let us make a case study on an ATM of a bank. The use-case diagram for the ATM is shown in Figure 8.19.

The consolidated requirements of the ATM taken from all the use-cases are given in Box 8.1. Let us start an analysis to identify noun/noun phrases. These are underlined in the textual description of requirements.

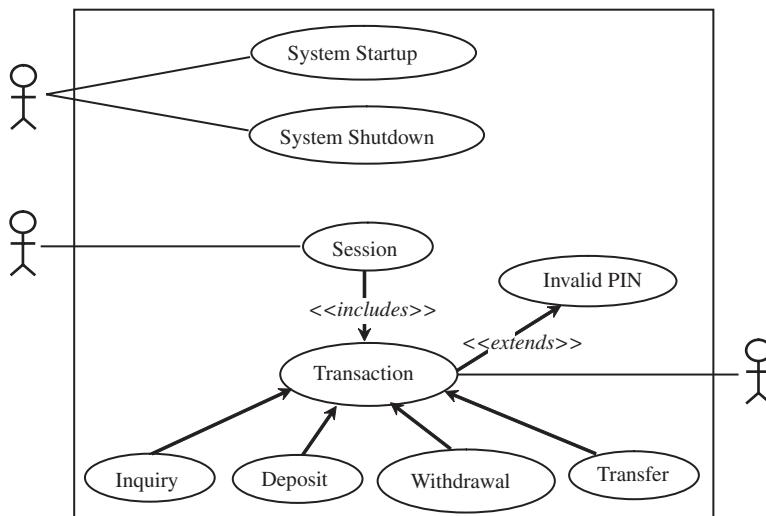


Figure 8.19 Use case of ATM

Box 8.2: Consolidated Requirements of an ATM

The system startup is made when the operator turns the operator switch to the 'on' position. The operator will enter the amount of money currently in the cash dispenser based on the question raised by the ATM system and a connection to the bank's server will be established. Then the customers can begin their transactions.

System shutdown is made when the operator makes sure that no customer is using the machine and then turns the operator switch to the 'off' position. The connection to the bank's server will also be cut off. Then the operator is free to remove deposited envelopes, replenish cash and papers etc.

A session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. The customer is asked to enter the Personal Identification Number (PIN) and is then allowed to perform one or more transactions, choosing from a menu of possible types of transactions in each case. The customer can cancel the transaction by pressing the Cancel key.

A transaction use-case is started within a session when the customer chooses a transaction type from an option menu. The customer is asked to furnish appropriate details (e.g. account(s) involved, amount). All messages to the bank and responses are recorded in the ATM's history. The Transaction is an abstract generalization. Each specific concrete type of transaction implements certain operations in the appropriate way as specified. The flow of events given here describes the behavior common to all types of transactions. The flows of events for the individual types of transactions, e.g. withdrawal, deposit, transfer, inquiry give the features that are specific to that type of transaction. The customer will get a receipt at the end of each transaction.

On going through the use-cases the following parts of the system are considered for identification of classes:

1. The ATM is considered as a control class that manages either the interface or boundary objects.
2. The following components of the ATM are considered as Interface class:
 - a. Operator panel
 - b. Network connection to the bank
 - c. Customer console, consisting of a display and keyboard
 - d. Card reader
 - e. Cash dispenser
 - f. Envelope acceptor
 - g. Receipt printer
3. Control classes corresponding to use-cases. It may be noted that the class ATM can handle the Startup and Shutdown use-cases. Thus, these do not give rise to separate classes here. Some of the other controller classes are:
 - a. Session
 - b. Transaction
4. The following components of the ATM may be considered as Entity class:
 - a. Information encoded on the ATM card inserted by a customer
 - b. The history of transactions maintained by the machine.

Based on the above underlined classes, the class Classification Diagram of the system is drawn and is shown in Figure 8.20.

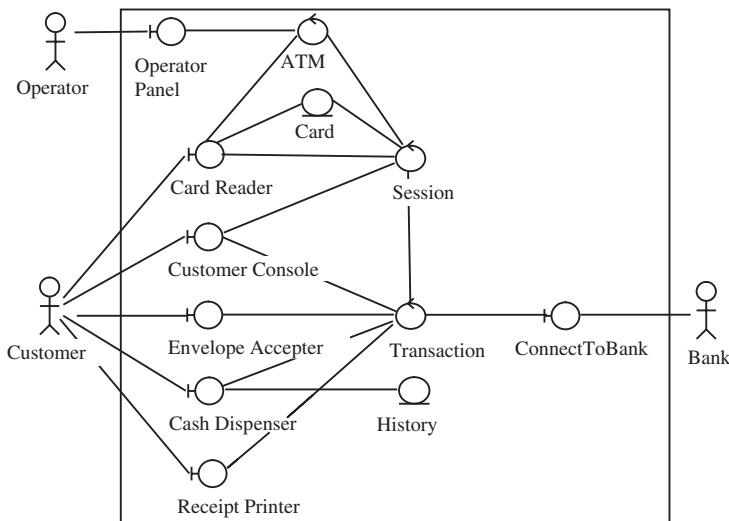


Figure 8.20 Class classification diagram

Noun Phrase Approach

From the requirements, noun phrases are underlined and the following classes are identified. Once a noun is underlined, it is not underlined again in its subsequent occurrences. The list of noun phrases is given in Table 8.3.

Table 8.3 List of Noun Phrases

| Noun phrases | Classes | Comments |
|-----------------------------|--------------------|--------------------------------------|
| System startup | | Irrelevant class with respect to ATM |
| Operator | | Actor, not a class |
| Operator switch | OperatorPanel | |
| On position | | Attribute of ATM |
| Amount | | A value, not a class |
| Money | Money | |
| Cash dispenser | CashDispenser | |
| ATM system | | Redundant of ATM |
| Connection to Bank's server | NetworkToBank | Changed nomenclature |
| Customers | | Actor, may not be a class |
| Transactions | Transaction | Singular of Transactions |
| System shutdown | | Irrelevant class with respect to ATM |
| Off position | | Attribute of ATM |
| Envelopes | | Irrelevant class |
| Cash | | Redundant of Money |
| Papers | | Irrelevant class |
| Session | Session | |
| ATM Card | | Redundant of Card |
| Card Reader Slot | CardReader | |
| Machine | | Redundant of ATM |
| ATM | ATM | |
| Card | Card | |
| PIN | | Attribute of ATM |
| Cancel key | | Attribute of Operator panel |
| Transaction type | | Attribute of Transaction |
| Details | AccountInformation | |
| Accounts | | Redundant of Account information |
| Messages | Message | |
| Responses | Status | Better nomenclature |
| History | History | |
| Withdrawal | Withdrawal | |
| Deposit | Deposit | |
| Transfer | Transfer | |
| Inquiry | Inquiry | |
| Receipt | Receipt | |

From the above class identifications, it may be seen that the envelope accepter and the receipt printer are not entered in the list of classes. One can write requirements and include them in the list of classes.

Classical Approach

Using the classical approach, the following candidate classes and objects can be identified:

Table 8.4 List of Classes as per Classical Approach

| Sources of class | Class |
|-------------------|--------------------|
| Things/Devices | ATM machine |
| Roles | Customer |
| Events | Request |
| Interactions | Session |
| People | Customer |
| Places | ATM room |
| Organizations | Bank |
| Structure | Withdraw |
| Concepts | Connection to bank |
| Events remembered | History |

Function Point Approach

Function point approach is another way to identify classes and objects derived from system functions. In this approach, we need to find out all the business functions and convert them into classes. Some of the classes in this example are given below:

Table 8.5 List of Classes as per Function Point Approach

| Sources of class | Class |
|--------------------|----------------------------|
| Input | Account information |
| Output | Money, Receipt |
| Inquiry | Inquiry |
| File | History |
| External interface | Network connection to bank |

CRC Cards

A CRC card is a technique to identify classes and determine their attributes and methods. Using this technique, the responsibilities of class_ATM are determined as follows:

- (i) Startup when switch is turned on
- (ii) Shutdown when switch is turned off
- (iii) Start a new session when card is inserted by customer
- (iv) Provide access to various components of ATM for sessions and transactions

For the above responsibilities, the OperatorPanel class is collaborated to start up and shutdown the ATM. In order to start the new session, the CardReader, CustomerConsole classes are to be collaborated. The Session class and the ConnectToBank classes are also collaborated to start the transactions. Hence, the collaborators of the ATM class are identified as follows:

- (i) OperatorPanel
- (ii) CardReader
- (iii) CustomerConsole
- (iv) Session
- (v) ConnectToBank

This may be shown in the CRC card as given in Figure 8.21.

| ATM class | |
|---|----------------------------|
| Responsibilities | Collaborators |
| Start up when switch is turned on | Operator Panel |
| Shutdown when switch is turned off | Card Reader |
| Start a new session when card is inserted by customer | Custmer Console Session |
| Provide access to component parts for sessions and transactions | Connect To Bank |
| | |

Figure 8.21 CRC card for ATM class

Final Use-Case Analysis

Based on analysis using various approaches, the final list of classes is given in Table 8.6.

Table 8.6 List of Classes of the ATM System

| Entity Class | Interface Class | Control Class |
|--------------------|-----------------|---------------|
| AccountInformation | CardReader | ATM |
| Card | CashDispenser | Session |
| History | CustomerConsole | Transaction |
| Message | ConnectToBank | Inquiry |
| Money | OperatorPanel | Deposit |
| Receipt | | Withdrawal |
| Session | | Transfer |
| Status | | |

Relationship

For the ATM system, Inquiry, Deposit, Withdrawal and Transfer are the sub-classes of the super-class Transaction. This is shown in Figure 8.22.

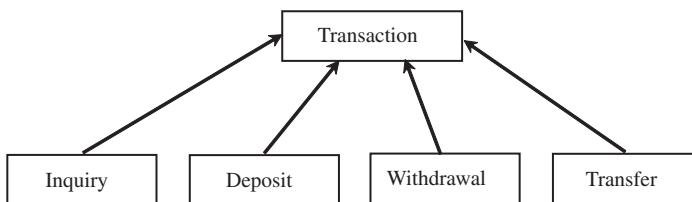


Figure 8.22 Generalization relation

The class hierarchy consists of the abstract class `Transaction` and four subclasses, namely, `Inquiry`, `Deposit`, `Withdrawal` and `Transfer`. The class `Transaction` has a ‘virtual constructor’ called `makeTransaction()`, which prompts the customer to choose a transaction type and then constructs and returns an object of the appropriate subclass. `Transaction` class is responsible for carrying out the `Transaction` use-case and the `Invalid PIN` extension. For these responsibilities it uses methods `getTransactionType()` and `completeTransaction()`, respectively, which are implemented by the appropriate subclass.

Attributes

Attributes of the `ATM` class in the `ATM` example may be defined as follows:

- `bankAddress`: Internet address of the bank
- `bankName`: Name of the bank owning this `ATM`
- `cardInserted`: Becomes true when the card reader informs the `ATM` that a card has been inserted; the `ATM` will make this false when it has tried to read the card
- `cardReader`: The `ATM`'s card reader
- `cashDispenser`: The `ATM`'s cash dispenser
- `customerConsole`: The `ATM`'s customer console
- `envelopeAcceptor`: The `ATM`'s envelope acceptor
- `id`: Unique ID for this `ATM`
- `IDLE_STATE`: The `ATM` is on, but idle.
- `history`: The `ATM`'s history
- `networkToBank`: The `ATM`'s network connection to the bank
- `OFF_STATE`: The `ATM` is off
- `operatorPanel`: The `ATM`'s operator panel
- `place`: Physical location of this `ATM`
- `receiptPrinter`: The `ATM`'s receipt printer
- `SERVING_CUSTOMER_STATE`: The `ATM` is servicing a customer
- `state`: The current state of the `ATM` – one of the possible values as given here: `switchOn`: Becomes true when the operator panel informs the `ATM` that the switch has been turned on: becomes false when the operator panel informs the `ATM` that the switch has been turned off.

It may be noted that besides the adjectives and adverbs like `id`, `place`, `on`, `off` etc., all the interface classes are part-of `ATM` and hence are included as attributes, which will further have links to the description of those classes.

Then in order to find the attributes and methods, let us first check the relationships among various classes. One such relationship between `ATM`, `Session` and `Transaction` is shown in Figure 8.23.

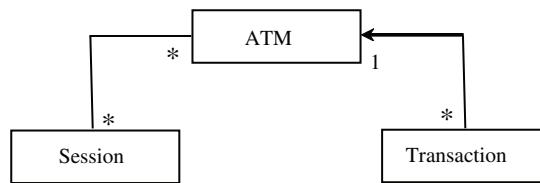


Figure 8.23 Association relation in ATM class

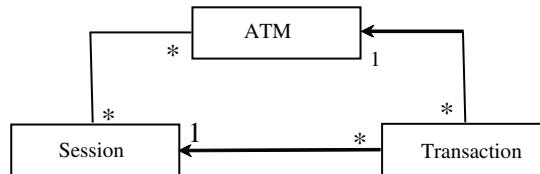


Figure 8.24 Aggregation relation between Session and Transaction

Aggregation is a stronger form of association, in which an object from one class is part-of or is contained-in an object of another class. Let us consider the same ATM example. A Transaction can be part-of the Session, which has been represented by an aggregation relation in Figure 8.24. Also, there can be many transactions in one session.

Operations (methods or behaviors) in the OO system usually correspond to queries about attributes of the objects. We need a set of operations that can maintain or change values; for example, an operation like `setBankName` will initially set the `bankName` attribute to the name of the bank.

The methods of the ATM class are determined as follows. Italic letters define the methods as shown below:

Table 8.7 List of Methods of ATM System

| Method | Description |
|---------------------|--|
| Run | To execute the main program of the system |
| switchOn | To send a signal to the ATM that the switch has been turned ON |
| switchOff | To send a signal to the ATM that the switch has been turned OFF |
| getOperatorPanel | To access the operator panel of the ATM |
| cardInserted | To inform the ATM that a card has been inserted into the card reader |
| getCardReader | To read the ATM card inserted into the ATM |
| getCustomerConsole | To access customer console of the ATM |
| getToBank | It returns the network connection to bank |
| getID | Returns unique id of the ATM |
| getPlace | Returns physical location of the ATM |
| getCashDispenser | To dispense cash by the ATM |
| getEnvelopeAcceptor | To acceptor envelope by the ATM |
| getReceiptPrinter | To print the receipt of a transaction |
| getHistory | To access the history component of the ATM |
| performStartup | Perform system startup when the switch is turned on |
| performShutdown | Perform system shutdown when the switch is turned off |

Identification of classes, responsibilities of classes and methods are the outcome of the OO analysis. These are further elaborated in OO design for development of software.

CASE STUDY—II: THE MILK DISPENSER

Let us consider the use-case diagram of a milk dispenser as shown in Figure 8.25. A merchant installs a milk dispenser system that provides milk to customers on payment basis. A customer requests for milk by inserting a predefined coin into the dispenser. The dispenser processes the request and dispenses one unit measure (say 500 ml) of milk to the customer. Therefore, the merchant has two use cases, e.g. 'add milk' and 'remove coin'. Similarly, the customer has two use cases: 'request milk' and 'receive milk'.

Class diagrams for all the classes are drawn and connected using relationships. The class diagram shows the class structure, contents and the static relationships among the classes used to model the system. Various relationships such as associations and inheritance are shown by lines connecting the related nodes. Each node in a class diagram is labeled with its class name. The class node also contains the attributes and methods. An association line among merchant and coin box indicates the linkage between these two classes. This association is labeled with a label 'own'. The relationship between 'milk' and 'dispenser' is 'part whole' relationship, called aggregations in UML. It is represented by a diamond at one end of the link. The relationship between 'valid coin' and 'coin' is 'inheritance' relationship, called generalization in UML. It is shown by a triangular arrow pointing from 'coin' to 'valid coin'

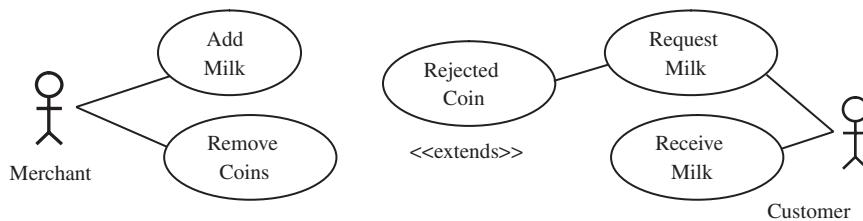


Figure 8.25 Use case diagram for milk dispenser

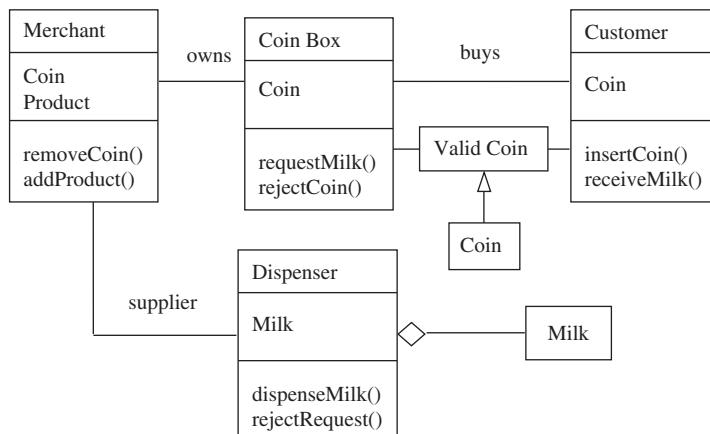


Figure 8.26 Class diagram of milk dispenser

A Sequence Diagram can be drawn showing the system behavior of use-cases through necessary class interactions. The customer as an actor can insert a coin that can be shown as insertCoin() method in Figure 8.27. It shows the exchange of a message from the actor to the object for performing the desired task.

The actors and objects are arranged horizontally across the top of the diagram. As the vertical dimension represents time, a vertical line called a lifeline is attached to each actor or object. The lifeline becomes an activation box to show the live activation period of the object or actor. The message is represented by an arrow labeled with a message, e.g. requestProduct().

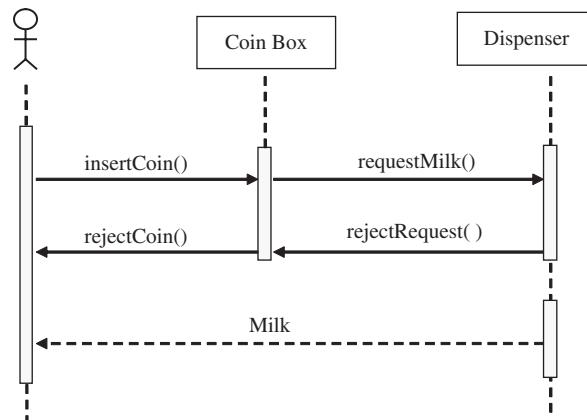


Figure 8.27 Sequence diagram

The Collaboration Diagram as shown in Figures 8.28 and 8.29 shows the message-passing structure of the system. There are two Collaboration Diagrams for the above two scenarios.

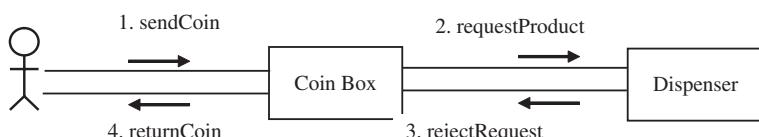


Figure 8.28 Collaboration diagram for return of coin

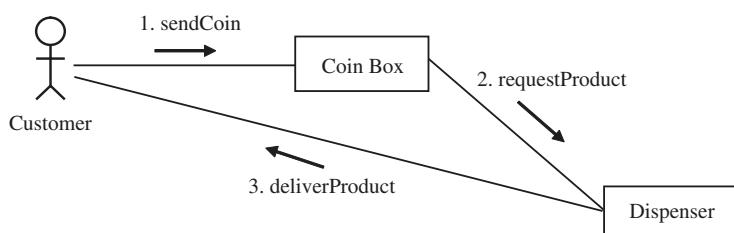


Figure 8.29 Collaboration diagram for delivery of product

The State Diagram for the system is shown in Figure 8.30. It is a directed graph whose nodes are labeled with state names. The nodes in a State Diagram are drawn as rectangles with rounded corners, e.g. wait for coin, check coin etc.

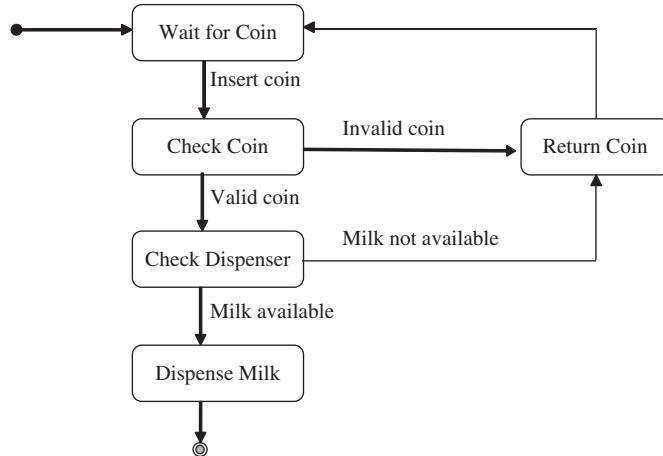


Figure 8.30 State diagram

The Activity Diagram for the system is shown in Figure 8.31.

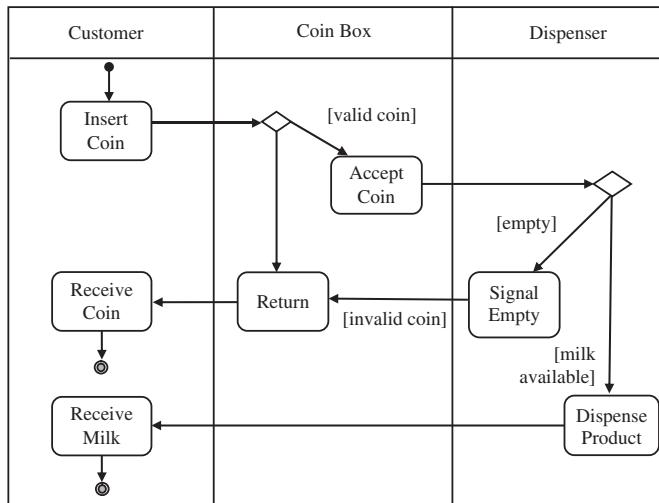


Figure 8.31 Activity diagram

This shows the workflow of the system components. This diagram is similar to the State Diagram except that transitions are triggered by internal events. Decision points are represented using unlabeled diamonds.

Note on Case Studies

The two case studies given above are meant to illustrate the procedure and use of UML tools in OO analysis. The identification of the classes, their relationships, attributes and methods is based on understanding of the system, which may vary from person to person. Usually it is a group effort. The various UML diagrams help to understand the system better for arriving at a common understanding. The readers may go through the case studies and examine the pros and cons of different aspects. These may also be discussed in a group.

SUMMARY

The OO methodology consists of OOA and OOD. In OOA, object-modeling techniques are applied to analyze functional requirements for a system.

Use-cases provide the external view of what the system does, and feed into OOA and design. The use-case model specifies the behavior of a system.

A system interacts with various external entities. Any entity that interacts with the system is termed as actor. A use-case is graphically depicted by the use-case diagram. In a use-case diagram, use-cases are represented by ovals (ellipse) and actors are represented by stick-men.

Use-case analysis provides a method to represent the ways in which a user uses a system. Representation of system requirements through use-case diagram facilitates better understanding of a process. It facilitates users and system analysts to work together for determining the system's requirements.

The use-case index lists all the use-cases and their attributes such as actors, scope, complexity, status and priority. The use-case document contains the details of a use-case. A use-case realization can be presented using Interaction Diagrams and Class Diagrams.

An Activity Diagram describes the workflow behavior of a system. In State Diagrams, the nodes represent states of an object in a class and edges represent occurrences of events. Synchronization bars can be put in both Activity Diagram and State Diagram. The synchronization bars represent concurrent states.

Interaction Diagrams are models that describe how a group of objects collaborate in some behavior. Interaction Diagrams come in two forms: (1) Sequence Diagram and (2) Collaboration Diagram.

A Sequence Diagram shows a set of messages arranged in time sequence. The basic use of the Sequence Diagram is to show the behavior sequence of a use-case. A Collaboration Diagram models the objects and the links that are meaningful to an interaction.

The OO analysis starts by identifying the classes from use-cases. The classes are classified into three types, viz., interface or boundary class, entity class and control class. The task of an interface object is to translate the actor's input into events of the system. The entity class consists of objects that constitute the elements of a system. The elements may be physical or conceptual. The entity objects are identified as 'nouns'. The main responsibilities of entity classes are to store and manage data in the system. Control class models the control behavior specific to one or more use-cases. It handles the tasks and the control flows of a system. It represents the dynamics of the system.

Identification of classes is an important step of the OO analysis. There are seven major class classification approaches, namely, (1) Noun phrase approach, (2) Classical approach, (3) Function point approach, (4) Domain analysis approach, (5) Structural approach, (6) CRC approach and (7) Use-case-driven approach.

After identification of classes, the relationship among them is determined. At the end, the attributes and the methods associated with each class are determined to complete the OO analysis process.

EXERCISES

1. Mention the activities involved in OOA.
2. Describe the process for identification and development of use-case.
3. Explain how use-cases help in determining the requirements in the OO approach with the help of an example.
4. Give a template for the use-case document. Explain a use-case by using the same template.
5. Can Requirement Engineering (RE) be done in the OO approach? Is it a practical approach? Explain.
6. What is the OO analysis? List various activities done in the OO analysis.
7. Draw an Activity Diagram that shows the process for starting a car, driving the car, stopping the car and switching off the ignition key.
8. Draw an Activity Diagram for sending an SMS through a mobile phone.
9. Write the application of a State Diagram.
10. Draw a Sequence Diagram for the process to check into a hotel.
11. Draw the Collaboration Diagram for the process to check into a hotel.
12. Discuss the purpose of boundary class.
13. Differentiate between entity class and control class with the help of a suitable example.
14. List various approaches for finding classes. Explain noun phrase approach with the help of an example.
15. Distinguish between relevant, irrelevant and fuzzy classes.
16. Determine the classes for a typical library system by using noun phrase and draw Class Classification Diagram.
17. In a particular academic institution, a student can register in five courses. A teacher can teach a maximum of three courses. A course can be taught by one or two teachers. Identify the classes and draw the Class Classification Diagram for the same.
18. Determine the classes for the billing system of a super market system by using the Classical approach.
19. What are CRC cards? Explain it in the context of a super market system.
20. Give a procedure for finding attributes and methods. Apply that procedure for finding attributes and methods in the context of a super market system.

This page is intentionally left blank.

OBJECT-ORIENTED DESIGN

This chapter describes the basics of object-oriented design (OOD) methodology. It covers the following topics:

- *System Architecture*
- *System Context Diagrams*
- *Component Diagrams*
- *Deployment Diagrams*
- *Patterns*
- *Frameworks*

The various design issues are discussed, mainly from the architectural point of view.

In the object-oriented analysis (OOA), object-modeling techniques are applied to analyze functional requirements for a system. In OOD, the analysis models are elaborated to produce implementation specifications. OOA focuses on '*What the system does*', whereas OOD focuses on '*How the system does it*'.

OOD is concerned with developing an object-oriented (OO) model of a software system that implements the identified requirements. Here, all objects are related to the solution. The steps of the OOD process are given below:

- Step 1: Identify the subsystems and their relationships.
- Step 2: Import the classes and methods from OOA and refine them.
- Step 3: Apply design axioms and refine class diagrams.
- Step 4: Develop views and access layer prototypes based on nonfunctional requirements (NFRs).
- Step 5: Test and continue refinement.
- Step 6: Prepare deployment diagrams and component diagram.
- Step 7: Check reusability in terms of patterns and frameworks.

In this chapter, we will discuss about tools and techniques of OOD.

9.1 SYSTEM CONTEXT AND ARCHITECTURAL DESIGN

To design any software, it is first necessary to understand the environment in which the software will be used. The interaction of a system with its environment is already discussed in OOA. Once the interactions between the software system and its environment have been defined, the knowledge can be used as a basis for designing the system architecture.

9.1.1 Defining System Boundary

A system boundary defines the scope of a system. In a use-case diagram, the processes of a system are shown inside a rectangle that represents the system boundary. The system boundary defines the problem statement. The actors of the system are normally shown outside the system. For large complex systems, each of the modules may be the system boundary. The use-case realization is already discussed while explaining the state diagrams and interaction diagrams, in Chapter 8. The actual use-case realization comes when the requirements are properly implemented.

An e-store system for online buying is considered here for explanation purposes. The interaction between the e-store system and the customer can be depicted by following use-cases.

1. Make Registration.
2. Search and Buy Items in Shopping Cart.
3. Make Payment by Customer.

On the other hand, the interaction between the e-store and the seller will be having only one important use-case, namely, shipping.

9.1.2 Identification of Subsystems

The e-store system may contain three subsystems as given below:

1. Registration: For customers to register with the e-store
2. Shopping Cart: To enable customer to search and select items for purchase
3. Payment: To enable customers to make payment or send payment details

Each subsystem will have many classes. For example, the payment subsystem will contain class Pay. The class Pay may be divided into two sub-classes, (1) Pay by Credit card and (2) Pay by Cheque as shown in Figure 9.1.

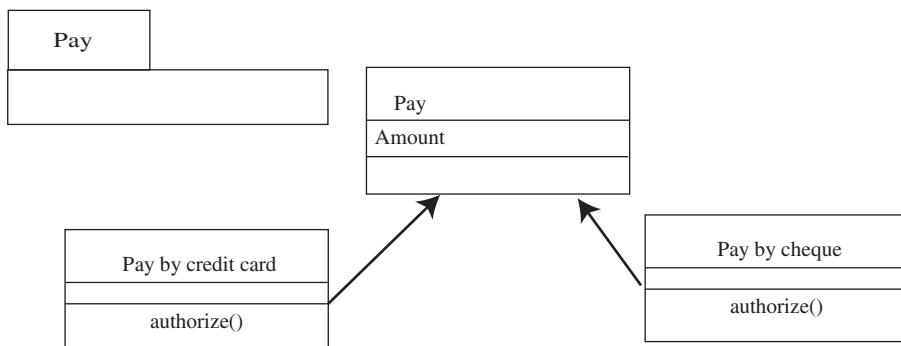


Figure 9.1 Payment subsystem of an e-store

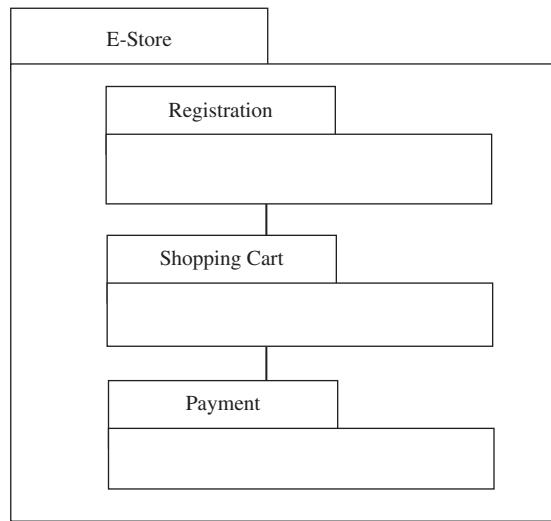


Figure 9.2 E-Store architecture

The subsystems of an e-store are shown in Figure 9.2. Finding out the subsystems is otherwise known as layering. Layering is the common way to design the structure of software. It is a way of organizing the software design into groups of classes that fulfil a common purpose.

The purpose of layering is to distribute the functionality of software among classes, so that coupling among classes is minimized. Each layer should be loosely coupled to the layers beneath it. One of the common layered architecture is the three-layered architecture consisting of presentation layer, business logic layer and data access layer.

9.1.3 Prioritization of Non-functional Requirements

Non-functional Requirements (NFR) such as performance, usability, authenticity, scalability, availability etc. need to be converted to design elements. The security aspects such as authenticity, confidentiality and data integrity are also important NFRs. It may not always be possible to incorporate all NFRs. Therefore, NFRs are assigned a priority as given below:

| | |
|--------------|---|
| Must have | Requirements that are fundamental to the system (mandatory) |
| Should have | Requirements that may be important |
| Could have | Requirements that are truly optional |
| Want to have | Requirements that can wait for later releases of the system |

This way of prioritizing is popularly known as MoSCoW.

9.1.4 Design Framework

The design of OO Software Systems depends on transformation in domain relationship as shown in Figure 9.3.

The design framework should ensure that the modules are correctly defined and located in the right place in the right order. To explain the concept, the V-model for a typical design framework is shown in Figure 9.4.

In this framework, the first step is to build the software hierarchy by following a top-down approach. The next step is to generate the full design matrix to define modules. The final step is to build the OO model with a bottom-up approach.

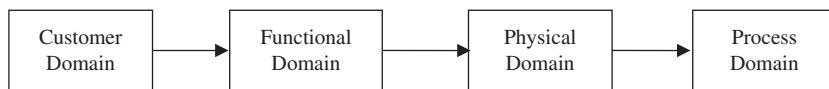


Figure 9.3 Concepts of domain

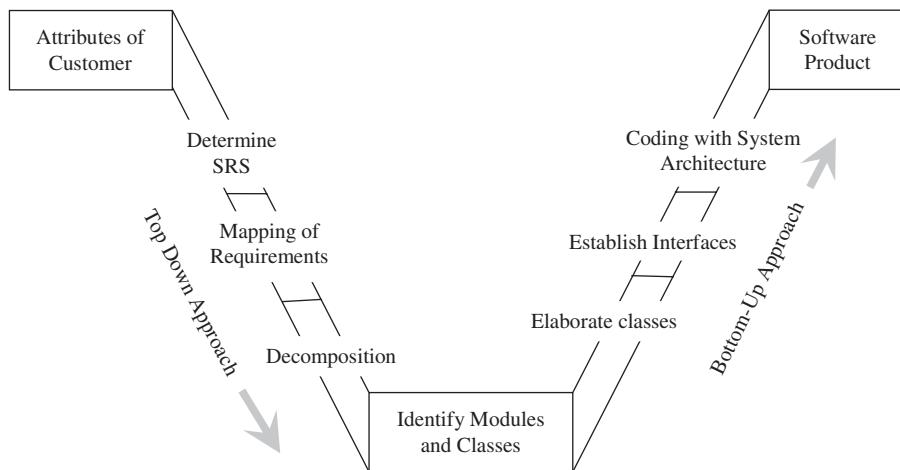


Figure 9.4 The design process of OO software systems

9.1.5 Design Axioms

An axiom is a fundamental truth that is always observed to be valid. Axioms cannot be proven or derived but they cannot be invalidated by counterexamples or exceptions. There are two design axioms applied to OOD. Axiom 1 deals with relationships between system components and Axiom 2 deals with the complexity of design.

Axiom 1: The independence axiom. Maintain the independence of components. As the software development process moves from requirement/use-case stage to a system component design stage, each component must satisfy the requirement of independence, without affecting other requirements.

Axiom 2: The information axiom. Minimize the information content of the design. It is concerned with simplicity. It relies on the premise that the best designs usually involve the least complex code but not necessarily the fewest number of classes or methods. Minimizing complexity should be the goal, because it produces maintainable and enhanced application. In an OO system, the best way to minimize complexity is to use inheritance. This is achieved by using the built-in classes and reusing the classes to the maximum extent possible.

A corollary is a proposition that follows from an axiom or another proposition that has been proven. A corollary is shown to be valid if its referent axioms and deductive steps are valid. The design rules or corollaries derived from design axioms are stated below:

Corollary 1: Uncoupled design with less information content

Corollary 2: Single purpose

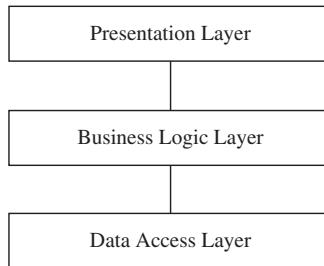


Figure 9.5 Three-layer architecture

Corollary 3: Large number of simple classes

Corollary 4: Strong mapping

Corollary 5: Standardization

Corollary 6: Design with inheritance

9.1.6 Access Layer Prototypes

The common layered architecture is the three-layered architecture consisting of presentation layer, business logic layer and data access layer. This is shown in Figure 9.5.

The presentation layer mostly discusses the user interfaces. The business logic layer describes the business processes and data access layer describes the data access through communication channels. For example, in the e-store system that has been already discussed the different layers can be as follows:

- The interfaces for user-registration and user-transaction can be part of the presentation layer.
- The customer-registration, product-purchase and payment-processing can be part of the business logic layer.
- The customer, product and transaction can be part of the data access layer.

9.2 PRINCIPLES OF CLASS DESIGN

It has been already discussed that the layers should be loosely coupled in the layered architecture phase. The same method is also applied in the design of classes. The classes and methods identified under OOA are specified in detail in OOD. If class 'student' is identified in OOA, then its attributes and methods are specified in detail in OOD. The design principles are stated below:

Completeness: It refers to the degree of providing users of a class the services they expect.

Primitiveness: It refers to creation of methods that should be designed to offer a single primitive, atomic and unique service.

Cohesiveness: It is a measure of the diversity of the class's features. A high cohesive class represents single abstraction. For example, instead of single class CustomerAccount for customers and accounts, it is preferable to have two classes, e.g. class Customer and class Account.

Coupling: It is a measure of the interconnectedness. The loosely coupled classes provide greater interdependence among classes. Hence, it is desirable that the classes should be loosely coupled. Of course, some coupling is required so that objects can interact.

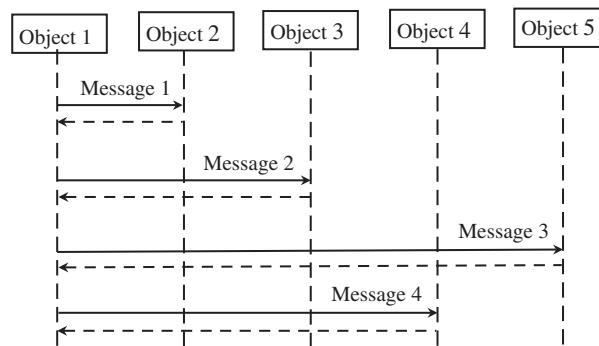


Figure 9.6 Coupling between objects — The finger effect

In the coupling of objects shown in Figure 9.6, object1 gives a message to all other objects and receives responses just like messages from a menu to different entities. In this form of coupling, the objects are tightly coupled with each other. This is called the finger effect.

It is always better to have a stair effect as shown in Figure 9.7. In fact, the finger effect is indicative of high coupling and the stair effect is indicative of low coupling.

OOD has three types of coupling, namely, (1) Interaction Coupling, (2) Identity Coupling and (3) Inheritance Coupling.

Interaction Coupling: It is a measure of the number of message types an object must send to another object and the number of parameters passed with those messages. This should be reduced in order to increase object reusability.

Identity Coupling: It refers to the level of connectivity in a design, i.e. in case of a reference from one object to another, the object knows the identity of the other object and hence exhibits identity coupling. It can be reduced by removing unnecessary associations from the class diagram and implementing associations in one direction only, in case bidirectional associations are unnecessary.

Inheritance Coupling: It refers to the coupling between superclasses and subclasses. A subclass is coupled to its superclass in terms of attributes and methods. High inheritance coupling is desirable. A client should try to refer always to the most general class or superclass, not to the subclasses.

The classes should be loosely coupled, highly cohesive, primitive and complete.

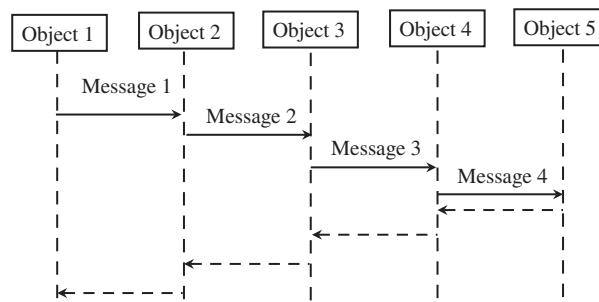


Figure 9.7 Coupling between objects — The stair effect

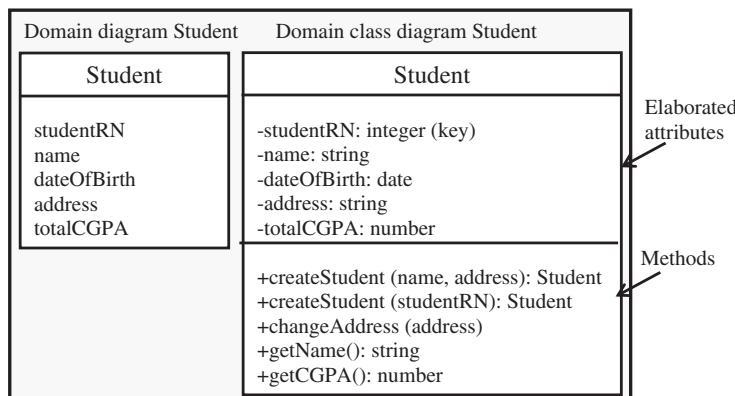


Figure 9.8 Examples of class diagrams

9.3 TYPES OF DESIGN CLASSES

In OOD the analysis classes are converted to design classes. For example, in case of student class, the attributes and methods can be written into the class diagram as shown in Figure 9.8.

The completion of a design class diagram has to be followed by a testing. In OO technology, the unit testing will be mainly made on classes, particularly on the methods. Class testing is followed by integration testing and system testing. More details about testing are discussed in Chapter 12.

In OO methodology, testing may be done in five levels. The five levels of testing are:

- | | |
|-------------------------------|---------------------|
| 1. Method | Unit testing |
| 2. Message quiescence | Integration testing |
| 3. Event quiescence | Integration testing |
| 4. Thread testing | System testing |
| 5. Thread interaction testing | System testing |

A method is programmed in an imperative language. It performs a single cohesive function. Therefore, method testing in OO methodology corresponds to the unit testing of the functional approach. In this testing both functional and structural techniques are applicable. Message and event quiescence generally correspond to integration testing. Thread testing and thread interaction testing correspond to system-level testing. In this, the high-level functional requirements that are represented by a thread are tested.

9.4 COMPONENT DIAGRAM AND DEPLOYMENT DIAGRAM

A component diagram shows the overall system architecture. A deployment diagram shows the physical components of a new system. However, both the diagrams are part of the physical view of the system. In fact, there are two physical views, such as implementation view and deployment view. The implementation view models the components in a system, i.e. the software units from which the application is constructed, as well as dependency among components. The other view, i.e. the deployment view, represents the arrangement of run-time component instances on node instances. A node is a run-time resource such as a computer, device or memory.

9.4.1 Component Diagram

The implementation view is displayed on the component diagram. The component diagram depicts how components are joined together to form larger components and/or software systems.

The component is a nearly independent part of a system, which fulfils a clear function in the context of a well-defined architecture. A component may be a source code component, a run-time component or an executable component. A component is a physical building block of the system. It is represented as a rectangle with tabs. A small circle attached to the component represents an interface. The dependency between components is shown by a dashed arrow. Figure 9.9 shows a component diagram for the Students Database Subsystem.

Component diagrams can be used to illustrate the structure of complex systems. The example above illustrates what a student database subsystem for an examination system might look like. It is possible to envisage that each of the components depicted in the above diagram will, in turn, have other component diagrams illustrating their internal structure.

9.4.2 Deployment Diagram

A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and how software elements and artifacts are mapped onto those nodes.

Artifact: An artifact is a product of the software development process. That may include process models (use-case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals etc. An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword.

Node: A Node is either a hardware or a software element. It is shown as a three-dimensional box. A node can contain other elements, such as components or artifacts.

Node Instance: A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon.

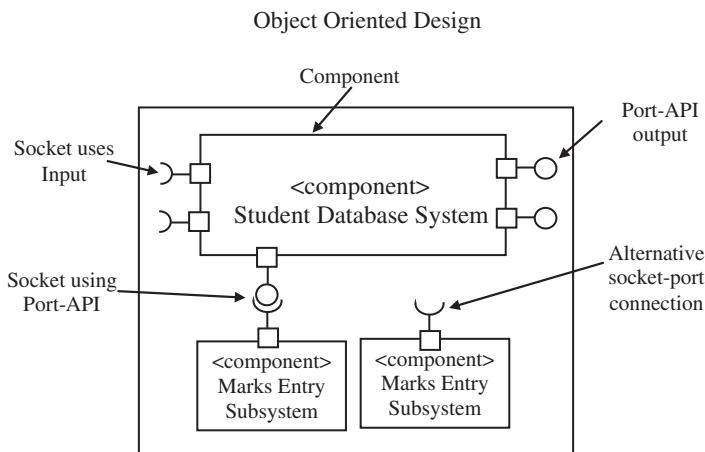


Figure 9.9 Component diagram of students database subsystem

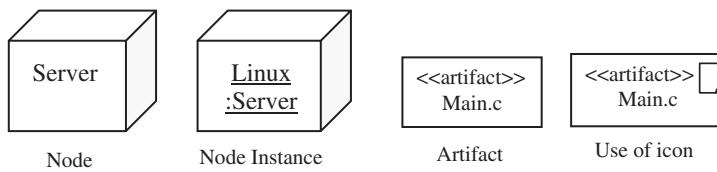


Figure 9.10 Symbols used in deployment diagram

Node Stereotypes: A number of standard stereotypes are provided for nodes, namely, «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc».

Association: In the context of a deployment diagram, an association represents a communication path between nodes.

Optionally, appropriate icons can also be displayed on symbols to identify the type of nodes/artifacts. This helps in better understanding of the deployment diagram. The symbols for node, node instance, artifact and use of icons are shown in Figure 9.10.

The deployment view is displayed on the deployment diagram. It represents the arrangement of runtime component instances on node instances such as any device or memory. The three-level architecture of an Internet system is depicted through a deployment diagram as shown in Figure 9.11.

The browser in the client side can connect to the server in order to access data from the database system. Artifacts can be components after they are compiled into executables. Therefore, the browser and server can be represented as individual nodes and linked as shown in the figure. In fact, there are two artifacts in the server, one the internet server and the other the application server. Also, the database component is linked with the server to complete the three-layer architecture.

The deployment diagram for part of an embedded system is shown in Figure 9.12. The diagram depicts an executable artifact as being contained by the motherboard node.

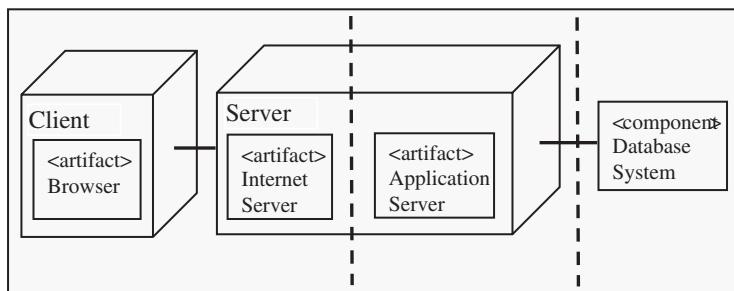


Figure 9.11 Deployment diagram of an internet system

9.5 PATTERNS

Patterns and frameworks are just some reuse techniques. *Reusability* is the likelihood that a segment of source code can be used again to add new functionalities with slight or no modification. Reusable modules and classes reduce implementation time, increase the likelihood that prior testing and use has eliminated bugs and localizes code modifications when a change in implementation is required.

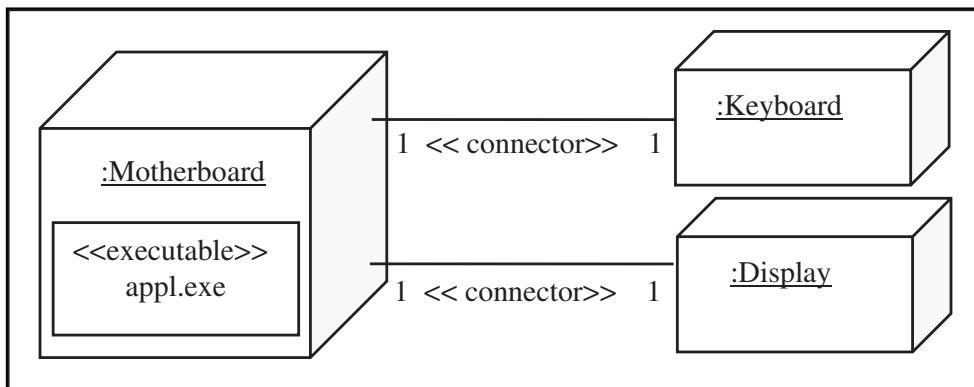


Figure 9.12 Deployment diagram of embedded system

The concept of pattern originated from the book titled ‘Design Patterns: Elements of Reusable Object-Oriented Software’ by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (frequently referred to as the *Gang of Four* or just *GoF*) in 1995. The concept became very popular with the wide acceptance of this book.

A *pattern* may be defined as instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.

The goal of patterns within the software community is to create a body of knowledge to help software developers resolve recurring problems encountered during software development. Patterns help to create a shared language for communicating insight and experiences about problems and their solutions. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design. The patterns created can guide others to learn from it and make use of it in similar situations.

9.5.1 Types of Patterns

There are three types of patterns in terms of levels of abstraction and detail. These are given below.

- Architectural Patterns:** They express a fundamental structural organization or schema for software systems. They provide a set of predefined subsystems. They also specify the responsibilities of subsystems and include rules and guidelines for organizing the relationships between them. These patterns are high-level strategies that concern large-scale components and the global properties and mechanisms of a system. They have wide-sweeping implications, which affect the overall skeletal structure and organization of a software system.
- Design Patterns:** They provide a scheme for refining the subsystems or components of a software system, or the relationships between them. They describe the commonly recurring structure of communicating components that solves a general design problem within a particular context. These patterns are medium-scale tactics that flesh out some of the structure and behavior of entities and their relationships. They do *not* influence overall system structure, but instead define *micro-architectures* of subsystems and components.

3. **Idioms:** They are low-level patterns specific to a programming language. They describe how to implement particular aspects of components or the relationships between them using the features of the given language. These patterns are paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behavior.

Among these the design patterns provide a much wider recognition as they are described by means of software design constructs, for example, objects, classes, inheritance, aggregation and use-relationship.

9.5.2 Pattern Template

Every pattern must be expressed in the form of a rule/template, which establishes a relationship between a context, a system of forces that arises in that context and a configuration, which allows these forces to resolve themselves in that context. A standard template should contain the following:

Pattern: <<pattern name>>

1. Synopsis: Describe the pattern in one or two sentences. This should convey the essence of a solution provided by the pattern. This is what experienced programmers will usually read.
2. Context: The context describes the problem the pattern addresses. A good way to present the context is by a concrete example and then proceed to describe the general problem.
3. Forces: These are the motivation for the particular solution. They summarize the considerations that lead one to the general solution presented in the next section.
4. Solution: This section describes the general-purpose solution to the problem this pattern addresses.
5. Consequences: Any solution will have positive and negative implications. This section describes these implications for this pattern.
6. Implementation: This section describes any considerations you should be aware of when implementing the solution. It can also describe common variations to the pattern and should provide a sample solution code (can be a link to a zip file).
7. Related Patterns: These are an optional section. If there are other patterns that address the same or similar problems, identify them here.

9.5.3 Generative and Non-generative Patterns

Generative patterns not only describe a recurring problem, but can also tell us how to generate something and can be observed in the resulting system architectures they helped shape. These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules that define the patterns in the world. However, in one respect they are very different. The patterns in the world merely exist. But the same patterns in our minds are dynamic. Each pattern is a rule that describes what you have to do to generate the entity that it defines. Non-generative patterns describe recurring phenomena without necessarily saying how to reproduce them. They are static and passive. We should strive to document *generative* patterns because they not only show us the characteristics of good systems but they also teach us *how to build them*.

9.5.4 Antipatterns

They are ‘worst practice or lesson learned’. They mean a bad solution to the given problem. They may appear to be a good solution, but they backfire when applied. They are negative solutions that present more

problems than they address. They are a natural extension to design patterns. Yesterday's good solutions can become today's antipatterns if they fail badly. The notion of antipattern was proposed by Andrew Koenig in the November 1995 C++ Report. There are two notions of 'antipatterns':

1. those that describe a bad solution to a problem, which resulted in a bad situation
2. those that describe how to get out of a bad situation and how to proceed from there to a good solution

A comparison between pattern and antipattern may be depicted pictorially as shown in Figure 9.13.

While design patterns consist of problem and solution pairs, the antipatterns consist of solution and solution pairs.

The solutions that are well documented in the design pattern each define re-factored solutions and then define antipattern solutions. The iterative writing process of the antipattern can be as follows:

Iterative antipattern writing process

- Identify antipatterns
- Define refactored solution
- Define antipattern solution
- Define symptoms and consequences
- Review and elaborate

In the social environment there are many antipatterns such as criminals, terrorists, drug addicts, witches etc. These people are generated out of the same society. Similarly, we can have many examples of software antipatterns such as spaghetti code, stovepipe system, vendor lock-in, blob or poltergeist. All these software antipatterns are generated in the same iterative process as given above. However, like patterns antipatterns can have a template as may be seen below. Each of the antipatterns has to follow the template in order to develop a complete antipattern.

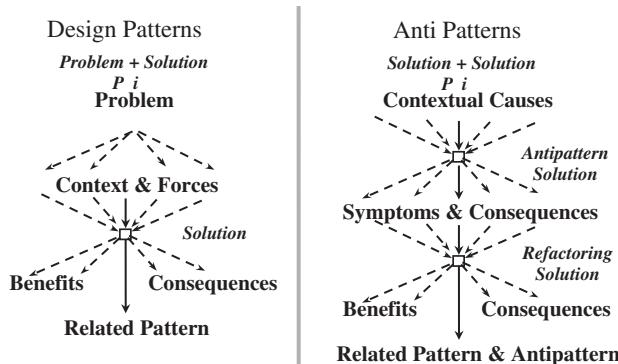


Figure 9.13 Design patterns vs. antipatterns

AntiPattern Template: <<antipattern name>>

1. Reference Model Keywords: This gives the keywords related to the antipattern.
2. Background: This gives the background of the problem in the form of context.
3. Anecdotal Evidence: This describes the evidence.
4. Antipattern Solution: This describes the general-purpose solution to the problem this antipattern addresses.
5. Symptom and Consequences: This describes the implications of this antipattern.
6. Typical Causes: This describes the reason.
7. Re-factored Solution: This separates the solution.
8. Variations: This gives related patterns and antipatterns.
9. Example: This provides the example.
10. Related Solutions: Related solutions are given here.

Spaghetti code is an undocumented code, which is extremely difficult to extend or modify. Similar to the unstructured code, the undocumented code is a liability. It results in an antipattern. The symptoms are the 50% maintenance spent on system rediscovery. In the OO spaghetti code, the symptoms are suspicious class or global variables. Many object methods do not have parameters. The OO advantage is lost. The re-factoring solution can contain either re-factor to generalize (i.e. create an abstract superclass) or re-factor to specialize (i.e. simplify conditionals by creating subclasses with matching invariants). The *blob* can be another antipattern having symptoms of a single class with many attributes and methods. Here the consequences have lost the OO advantage. It is too complex to reuse or test. The re-factoring consists of the following steps:

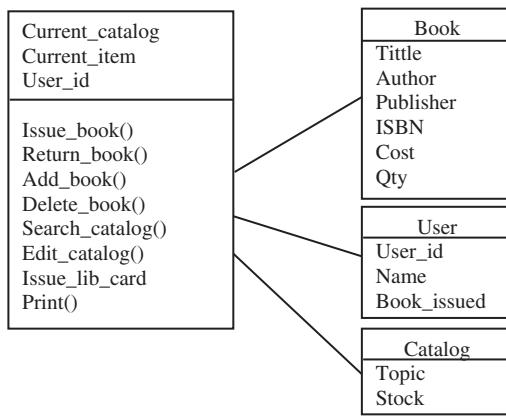
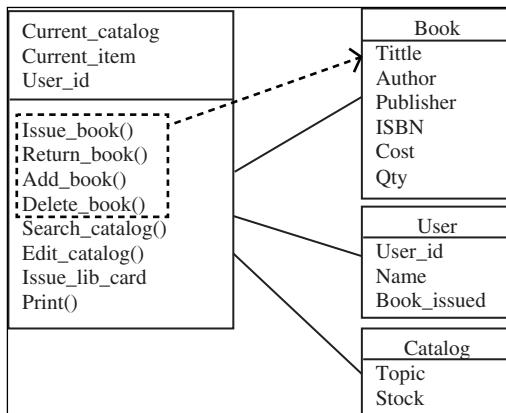
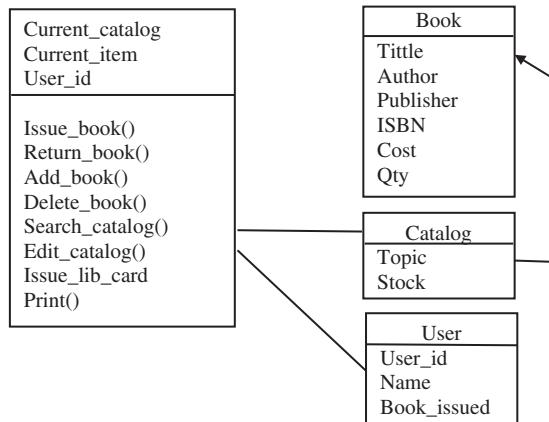
The Blob Refactoring

- Identify or categorize related attributes and methods according to contracts.
- Find natural homes for these contract-based collections of functionality and then shift them there.
- Remove all transient associations , replacing them as appropriate with type specifiers to attributes & methods arguments.

For example, let us consider the methods of a library class shown in Figure 9.14.

The indicated methods shown in Figure 9.15 are the related methods and belong to the item class. Also, the methods contain some related methods containing terms catalogue and can belong to the class catalogue. Hence, these methods can be shifted to the relevant classes. After that the coupling between library class and item class is removed and is placed between the catalogue class and item class, which may be seen in Figure 9.16.

Similarly, there can be many more antipatterns as suggested earlier. However, the development process of all these antipatterns follows the same iterative writing process.

**Figure 9.14** Library class**Figure 9.15** Refactored library class**Figure 9.16** Final blob refactored library class

9.6 FRAMEWORK

Frameworks are a way of delivering application development patterns to support best practice sharing during application development across different companies. For example, we can have Struts Framework, .NET Framework, Java Beans Activation Framework etc. A framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate. It is a reusable design for all or part of a software system. By definition, a framework is an OOD. It does not have to be implemented in an OO language, though it usually is. Large-scale reuse of OO libraries requires frameworks. The framework provides a context for the components in the library to be reused.

The difference between a framework and an ordinary programming library is that a framework employs an *inverted flow of control* between itself and its clients. When using a framework, one usually implements a few callback functions or specializes a few classes and then invokes a single method or procedure. At this point, the framework does the rest of the work by invoking any necessary client callbacks or methods at the appropriate time and place. Gamma et al. have given some of the major differences between design patterns and frameworks. The differences between frameworks and design patterns are listed below.

| Frameworks | Design Patterns |
|--|---|
| Frameworks can be written down in programming languages and reused directly. | Only examples of design patterns can be embodied in codes. Thus, design patterns are more abstract than frameworks. |
| A typical framework contains several design patterns. | Design patterns are smaller architectural elements than frameworks. |
| Frameworks are the physical realization of one or more design patterns. | Patterns are the instructions for how to implement those solutions. |
| Frameworks always have a particular application domain. They are more specialized than design patterns. | In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these would not dictate application architecture. |
| Frameworks are of physical nature. These are executable software used in either the design or the runtime phase. | Design patterns are of a logical nature, representing the knowledge of and experience gained with software. |

Descriptions of some common frameworks are given below.

9.6.1 Struts Framework

Model View Controller (MVC) architecture is another OOD architecture that can be used in various places. As discussed in Section 9.1, it represents model, view and controller. The *Model* represents the business or database code, the *View* represents the page design code and the *Controller* represents the navigational code. The main aim of the MVC architecture is to separate the business logic and application data from the presentation data to the user.

- **Model:** It contains the core of the application's functionality. It encapsulates the state of the application. Sometimes the only functionality it contains is state. It knows nothing about the view or controller.
- **View:** It provides the presentation of the model. It is the *look* of the application. It can access the model getters, but it has no knowledge of the setters. In addition, it knows nothing about the controller. The view should be notified when changes to the model occur.
- **Controller:** It reacts to the user input. It creates and sets the model.

However, in MVC1 architecture there is tight coupling between page and model as data access is usually done using custom tag or through java bean call. MVC2 architecture removes the page-centric property of MVC1 architecture by separating presentation, control logic and the presentation state. In MVC2 architecture there is only one controller that receives all requests for the application and is responsible for taking appropriate action in response to each request as shown in Figure 9.17.

Struts Framework is a standard for developing well-architected web applications. It has the following features:

- It is an open source.
- It is based on the MVC design paradigm, distinctly separating all three levels:
 - Model: application state
 - View: presentation of data (JSP, HTML)
 - Controller: routing of the application flow
- It implements the JSP Model 2 Architecture
- It stores application routing information and request mapping in a single core file, struts-config.xml.

The Struts MVC architecture is implemented in the following way:

- The model contains the business logic and interacts with the persistence storage to store, retrieve and manipulate data.
- The view is responsible for displaying the results back to the user. In Struts the view layer is implemented using JSP.
- The controller handles all requests from the user and selects the appropriate view to return. In Struts the controller's job is done by the ActionServlet.

This is known as Struts1. There is another approach known as Struts 2, which is based on the OpenSymphony Web Works Framework. Here, the model, view and controller are implemented by the action,

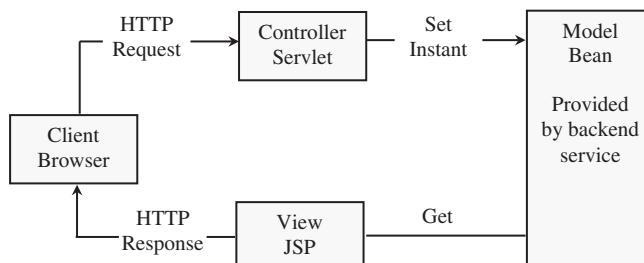


Figure 9.17 MVC model2

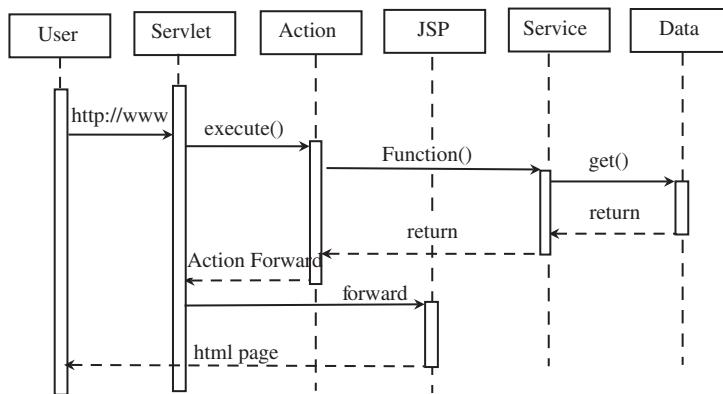


Figure 9.18 Sequence diagram to explain the working of Struts1

result and FilterDispatcher, respectively. However, we restrict ourselves to Struts1 in this book. We present a sequence diagram to explain the working of Struts1 as per Figure 9.18. The working of Struts is described in the following steps:

1. User clicks on a link in an HTML page.
2. Servlet controller receives the request, looks up mapping information in struts-config.xml and routes to an action.
3. Action makes a call to a Model layer service.
4. Service makes a call to the Data layer (database) and the requested data is returned.
5. Service returns to the action.
6. Action forwards to a View resource (JSP page).
7. Servlet looks up the mapping for the requested resource and forwards to the appropriate JSP page.
8. JSP file is invoked and sent to the browser as HTML.
9. User is presented with a new HTML page in a web browser.

9.6.2 .NET Framework

Microsoft .NET is software that connects information, people, systems and devices. It spans clients, servers and developer tools. It consists of all kinds of software, including Web-based applications, smart client applications and XML Web services. It also contains components to facilitate integration and sharing of data and functionality over a network through standard, platform-independent protocols such as XML (Extensible Markup Language), SOAP and HTTP. Developer tools such as Microsoft Visual Studio® .NET 2003 provide an integrated development environment (IDE) for maximizing developer productivity with the .NET Framework.

SUMMARY

In OOD, the analysis models are elaborated to produce implementation specifications. It involves the development of the software system model that implements the identified requirements.

A system boundary defines the problem statement and the scope of the system. The system may have many subsystems and each subsystem will have many classes. Finding out the subsystems is known as layering. Layering is the way of organizing the software design into groups of classes. The three-layered architecture is the common form of architecture. It consists of the presentation layer, business logic layer and data access layer. The ‘MVC’ architecture and ‘Boundary Control Entity (BCE)’ architecture are other layered architectures.

The design framework ensures that the modules are correctly defined and located in the right place in the right order. There are two design axioms applied to OOD. Axiom-1 prescribes that the independence of components should be maintained. Axiom-2 prescribes that the complexity of components should be minimized.

Completeness, primitiveness, cohesiveness and coupling are four important principles of design. OOD has three types of coupling, namely, (1) Interaction coupling, (2) Identity coupling and (3) Inheritance coupling. The classes should be loosely coupled, highly cohesive, primitive and complete.

In OOD, the analysis classes are converted to design classes. In class diagram, the classes are represented by using the symbols for objects corresponding to the types of classes. The interface class (also called boundary class) represents the boundary between the system and its actors. Entity class represents the information the system uses, control class represents the control logic of the system and data access class represents the retrieval and sending of data from and to the database. A comprehensive class diagram is drawn, which specifies the attributes and methods.

A component diagram shows the overall system architecture. It depicts the implementation view. A component may be a source code component, a run-time component or an executable component. A deployment diagram shows the physical components of a new system. It represents the arrangement of the run-time component.

A pattern is instructive information on solutions to a recurring problem learnt from past experience. There are three types of patterns: (1) Architectural Patterns, Design Patterns and Idioms. Every pattern must be expressed in the form of a rule called pattern template. Antipattern is the “worst practice or lesson learned”. Understanding antimatters provides the knowledge to avoid mistakes and to come out of it. While design patterns consist of problem and solution pairs, antipatterns consist of solution and solution pairs.

Frameworks are a way of delivering application development patterns to support best practices. A framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate. It is a reusable design for all or part of a software system. Struts Framework, .NET Framework, Java Beans Activation Framework are some important frameworks.

EXERCISES

1. Explain how OOD is different from OOA.
2. What is layered architecture? Explain how to draw a layered architecture for an Internet system.
3. Explain the axioms and corollary for software design.
4. Describe the prominent features of software designed by OO methodology?
5. Give an account of Axiomatic Design Process of OO Software Systems.

6. Describe the client server architecture of a multinational IT organization.
7. What are the different coupling techniques used in OOD? Explain through examples.
8. Find out the various design classes in the student course registration system. Also find out the analogy between analysis classes and design classes.
9. Define patterns, components and framework. Find a bilateral relationship between them.
10. Explain how antipatterns are helpful in software development.
11. Explain why .NET and Struts are called as frameworks.

This page is intentionally left blank.

USER INTERFACE DESIGN

Users like software that has an attractive and appealing user interface (UI). Hence, for the purpose of marketing the software and customer satisfaction, the UI is considered as an important component of any software. Different aspects of the UI such as types of UIs, their characteristics, design principles and procedures have been discussed in this chapter.

Objects are often judged by their looks. Good dress and grooming enhances the personality of a person. Colorful paints and packaging make a product more appealing and desirable to the customers. What is true for a person or physical product is also applicable to software. Software is also judged by its looks. User interface (UI) is the visual part of software. It consists of opening screens, input screens, output screens, response to users command displayed on the screen as dialogues or messages, warning messages etc. Users/customers usually make the first impression about the software from the looks of its UI.

The UI is also important because it determines how easily one can operate the software. It is a mechanism through which the users communicate with a program. It is the junction between users and a computer system. Even powerful software with a poorly designed UI is not liked by users.

10.1 TYPES OF USER INTERFACE

UIs can be classified into two major categories:

- Text-based Interface or Textual User Interface (TUI)
- Iconic User Interface or Graphical User Interface (GUI)

A text-based user interface or TUI is a mechanism for interacting with the computer by typing commands to perform specific tasks. In a text-based interface, the user types in a command and submits the text command by pressing the 'Enter' key. A command-line interpreter then receives, analyzes and executes the command to give the output. After this the system waits for the next command to be given by the user. The text-based interpreter usually runs in a text terminal.

Iconic interface or GUI presents the interface to the user in the form of visual models (i.e. icons or objects). In this type of interface, the user issues commands by performing actions on the visual representations of the objects; e.g. pull an icon representing a file into an icon, for opening the file. For this reason, GUI is sometimes called Direct Manipulation interface. The GUI uses a graphical form in which the icons are placed. The form is called 'Window'. A pointing device (e.g. mouse) is used to select an icon and give command.

Both TUI and GUI can be provided with menus and help tips to make it easier for users to operate the software.

10.2 CHARACTERISTICS OF USER INTERFACE

To design a User Interface (UI), it is important to identify the characteristics desired of a good quality UI. A few important characteristics of a quality UI are as follows:

Speed of learning: A good UI should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good UI should not require its users to memorize commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. The speed of learning characteristic of a UI can be determined by measuring the training time and practice that users require before they can effectively use the software.

Use of standard notations: Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar. This can be achieved if the interfaces of different applications are developed using some standard UI components. This, in fact, is the theme of the component-based UI. Examples of standard UI components are radio button, check box, text field, slider, progress bar etc.

Speed of use: Speed of use of a UI is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is sometimes referred to as productivity support of the interface. It indicates how fast the users can perform their intended tasks. The time and user effort necessary to initiate and execute different commands should be minimal. This can be achieved through careful design of the interface. For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow down the operating speed of users. The most frequently used commands should have the smallest length or be available at the top of the menu to minimize the mouse movements necessary to issue commands.

Speed of recall: Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This characteristic is very important for intermittent users.

Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures and intuitive command names.

Consistency: Once a user learns about a command, he should be able to use similar commands under different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. For example, in a word processor, 'Control-b' is the short-cut key to embolden the selected text. The same short-cut should be used on the other parts of the interface, for example, to embolden text in graphic objects also – circle, rectangle, polygon etc. The commands supported by a UI should be consistent. The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part

to another. Thus, consistency facilitates speed of learning, speed of recall and also helps in reduction of error rate.

Error prevention: A good UI should minimize the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by average users while using the interface. This monitoring can be automated by instrumenting the UI code with the monitoring code, which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users.

Moreover, errors can be prevented by asking the users to confirm any potentially destructive actions specified by them, for example, deleting a group of files.

Consistency of names, issue procedures and behavior of similar commands and the simplicity of the command issue procedures minimize error possibilities. Also, the interface should prevent the user from entering wrong values.

Attractiveness: A good UI should be attractive to use. An attractive UI catches user attention and fancy. In this respect, graphics-based UI's have a definite advantage over text-based interfaces.

Feedback: A good UI must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic. If required, the user should be periodically informed about the progress made in processing his command.

For example, if the user specifies a file copy/file download operation, a progress bar can be used to display the status. This will help the user to monitor the status of the action initiated.

Support for multiple skill levels: A good UI should support multiple skill levels of users. This is necessary because users with different levels of skill in using a computer system prefer different types of UIs. Experienced users are more concerned about how efficiently the commands can be issued, whereas novice users pay importance to usability aspects. Very cryptic and complex commands may discourage a novice user. New users will prefer that the sequence of commands should be step-wise and elaborate. However, this may make the command issue procedure very slow and therefore put off experienced users.

After a new user works on a system for certain periods of time, he becomes familiar with the operation. As the user becomes familiar with the system, his focus shifts from usability aspects to how fast the commands can be issued and executed. Experienced users look for options such as 'hot-keys', 'macros' etc. Thus, when the skill level of users improves, they look for commands to suit their skill levels.

Error recovery (undo facility): While issuing commands, even expert users can commit errors. Therefore, a good UI should allow a user to undo a mistake committed by him while using the interface. Users are put to inconvenience, if they cannot recover from the errors they commit while using the software.

User guidance and on-line help: Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with the appropriate guidance and help.

10.3 TEXTUAL USER INTERFACE

Textual User Interfaces (TUI) can be of two types:

1. Command Language-based UI
2. Menu-based UI

10.3.1 Command Language-based Interface

In a command language-based TUI, the user types in the command to operate the system. Hence, for this the users need a command language, which they can use to issue commands. Users are expected to frame the appropriate commands in the language and type them in appropriately whenever required.

A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands. Such a facility to compose commands dramatically reduces the number of command names. Thus, a command language-based interface can be made concise requiring minimal typing by the user. The earlier Disk Operating System (DOS) has command language-based UI.

For example, 'dir' is the command in DOS operating system to view files and folders (called directory in DOS). Hence, by typing this command at the command prompt followed by the 'enter' key one can view all the folders and files. However, if we want to search a file named 'student.dat', it may be preferable to give a command to list all files starting with 's'. Alternatively, we may prefer to list all files having the extension 'dat'. Hence, the same command with suitable suffix can be used to get different outputs as shown in Table 10.1.

Table 10.1 DOS Command to List Files and Folders

| | |
|--------------|--|
| C> dir | A simple command to list all files and folders of C drive |
| C> dir s*.* | A composed command to list all files and folders of C drive starting with letter 's' |
| C> dir *.dat | A composed command to list all files and folders of C drive having extension 'dat' |

In the above case

dir is the primitive command whereas

dir s.** and *dir *.dat* are composed commands

Composed commands are obtained by adding optional parameters to the primitive command. Thus, command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.

Disadvantages of command language-based interface: Command language-based interfaces have several drawbacks. Usually, they are difficult to learn and require the users to memorize the commands. Also, most users make errors while formulating commands in the command language and also while typing the commands.

Further, in a command language-based interface, all interactions with the system is through a keyboard and a user cannot take advantage of effective interaction devices such as a mouse. Obviously, for casual and inexperienced users, command language-based interfaces are not suitable.

Design Issues: Two major design issues in command language-based interface are:

1. To reduce the number of primitive commands that a user has to remember
2. To minimize the total typing required while issuing commands

To achieve these two objectives, the designer has to decide what mnemonics are to be used for the different commands.

The designer should try to develop meaningful mnemonics and yet be concise to minimize the amount of typing required. For example, the shortest mnemonic should be assigned to the most frequently used commands.

The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences. Letting a user define his preferred mnemonics for various commands is a useful feature, but it increases the complexity of UI development.

The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A command composition facility would require the syntax and semantics of the various command composition options to be clearly and unambiguously specified. The ability to combine commands is a powerful facility in the hands of experienced users. A good UI should have features to suit users of different skill levels. Hence, a command composition facility is usually made available in the case of command language-based interface.

10.3.2 Menu-based TUI

Menu-based interface permits the user to select the command (options) from a number of commands displayed on the screen. Hence, the user is not required to remember the commands and their syntax. This is an important advantage of menu-based interface over command language-based interface. However, menu-based interface also has drawbacks. It is not possible to compose commands in a menu-based interface. This is because of the fact that actions involving logical connectives (and, or etc.) are difficult to specify in a menu-based system. Also, if the number of choices is large, it is difficult to select from the menu. In fact, a major challenge in the design of a menu-based interface is to structure a large number of menu choices into manageable forms. A textual menu-based UI is shown in Figure 10.1.

Experienced users sometimes feel that working in a menu-based interface is slower than working in a command language-based interface. This is because an experienced user can type fast and can get speed advantage by composing different primitive commands to get the desired output. However, most users prefer menu-based interface to command language-based interface. A menu-based interface is based on recognition of the command names, rather than recollection. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections. The menu-based interface is also useful in GUI.

10.4 GRAPHICAL USER INTERFACE

Graphical User Interface (GUI) presents the interface to the user in the form of visual models (i.e. icons or objects). **Icon** is an informative picture/symbol displayed on the screen, which the user chooses to select an action. Icons can usually be used to represent software packages, documents and hardware devices. For this reason, GUI is sometimes called as ‘iconic interface’. It is a direct manipulation interface. In this type of interface, the user issues commands by selecting objects by a pointing device and performing

| |
|----------------------------|
| What you want to do? |
| 1. Check Account Balance |
| 2. Print Account Balance |
| 3. Withdraw money |
| 4. Transfer money |
| 5. Cancel or Exit |
| Enter Your choice (1-5): _ |

Figure 10.1 A textual menu-based interface

actions on the selected objects. The **pointer** is a symbol (e.g. arrow, line etc.), which is moved by a pointing device and can be used to select objects. For example, to delete a file, the icon representing the file is dragged into an icon representing a trash box. The icons are objects. Hence, properties can be specified for the icons. The program codes can also be added to the icons. The programs are invoked by clicking on the icons with a mouse.

Users like to use pointing devices such as joystick, trackball, light pen, touch screen and mouse to operate the computer system. Other advantages of iconic interfaces include the fact that the icons can be recognized by users very easily and that icons are language-independent. However, direct manipulation interfaces can be considered slow for experienced users. Also, it is difficult to give complex commands using a direct manipulation interface. For example, a file can be deleted by dragging the icon representing the file to a trash box icon. However, if it is required to delete all files of a directory whose name start with alphabet 's' then each file has to be selected and dragged individually to the trash box. However, in command-based interface the same task can be done very easily by issuing a simple command like '*delete s.**'.

Figure 10.2 shows an iconic interface. Here, the user is presented with a set of icons at the top of the frame for performing various activities. On clicking on any of the icons, either the user is prompted with a submenu or the desired activity is performed.

10.4.1 Menus in Graphical Interface

The 'Windows Icon Menu and Pointer (WIMP)' system is the most popular form of GUI. The pointer is used to select objects placed on the window. However, when the number of objects is too many, all these cannot be accommodated on a single window. Hence, one way is to group the objects into different categories and present these in a menu. This facilitates the user to select the required options from the menu. There are three types of menus:

1. The Scrolling Menu
2. The Walking Menu
3. The Hierarchical Menu

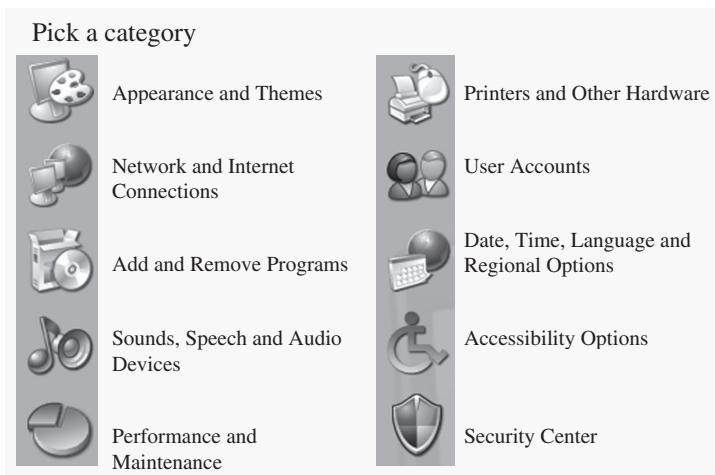


Figure 10.2 Example of an iconic interface

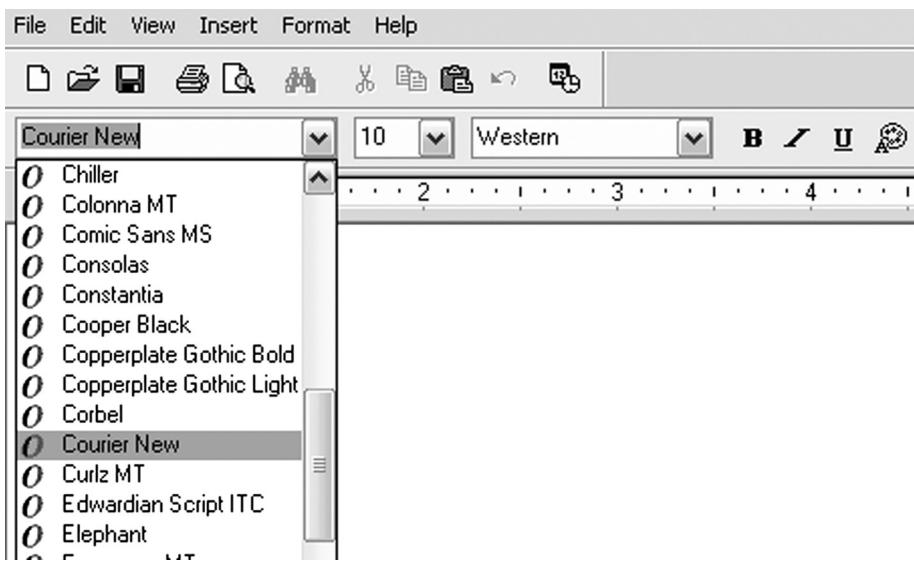


Figure 10.3 Scrolling menu

Scrolling Menu: When a full choice list cannot be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen.

However, in a scrolling menu, the user cannot see all the commands at any one time. Thus, it is important that the commands should be grouped together based on some similarity of purpose. This helps the user to locate the command that he needs. An example of a scrolling menu for word-processor software, which helps the user to select a particular type of font, is shown in Figure 10.3.

Here, the user knows that the command list contains only the font types that are arranged in some order and he can scroll up and down to find the font type he is looking for.

Walking Menu: Walking menu is very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a submenu. An example of a walking menu is shown in Figure 10.4. A walking menu can successfully be used to structure a large number of commands in a single screen.

Hierarchical Menu: Walking menu can accommodate a large number of commands in a single screen. However, due to limitation of screen size, there is a limit to the number of commands that can be accommodated.

In hierarchical menu, the menu items are organized in a hierarchical or tree-like structure. Selecting a menu item causes the current menu display to be replaced by an appropriate submenu. Thus, in this case, one can consider the menu and its various submenus to form a hierarchical structure. Hierarchical menu can be used to manage a large number of choices, but the users are likely to face navigational problems because they might lose track of where they are in the menu tree.

10.4.2 Mode-based Interface and Mode-less Interface

A mode is a state or collection of states of the system, which permits only some of the commands to be performed. Hence, in a mode-based interface, different sets of commands can be invoked depending on

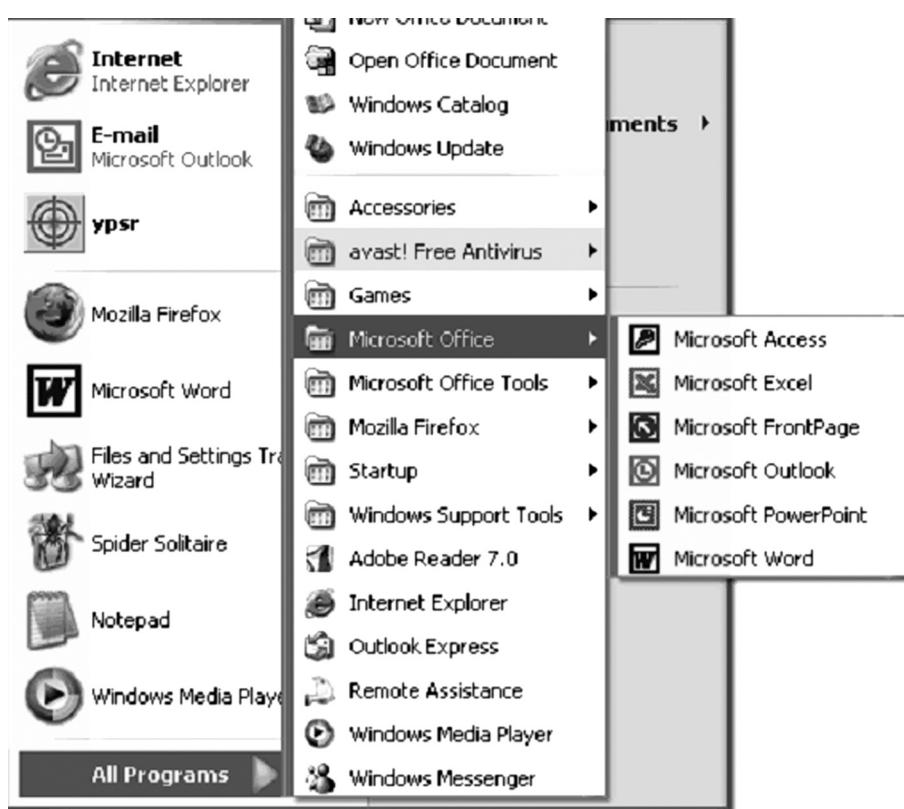


Figure 10.4 Example of walking menu

the mode of the system. The mode at any instant is determined by the sequence of commands already issued by the user.

On the other hand, in a mode-less interface, the same set of commands can be invoked at any time during the running of the software. Thus, a mode-less interface has only a single mode and all the commands are available all the time during the operation of the software.

A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

Figure 10.5 shows the mode-based interface of word processor software. When the user chooses the save option on file menu, another frame pops up, which allows the user to save the document as per his requirement. The mode-based interface is useful for the software product that has a large number of commands.

10.4.3 Window Management System

A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g. one window can be used for editing a program, another for drawing pictures etc.

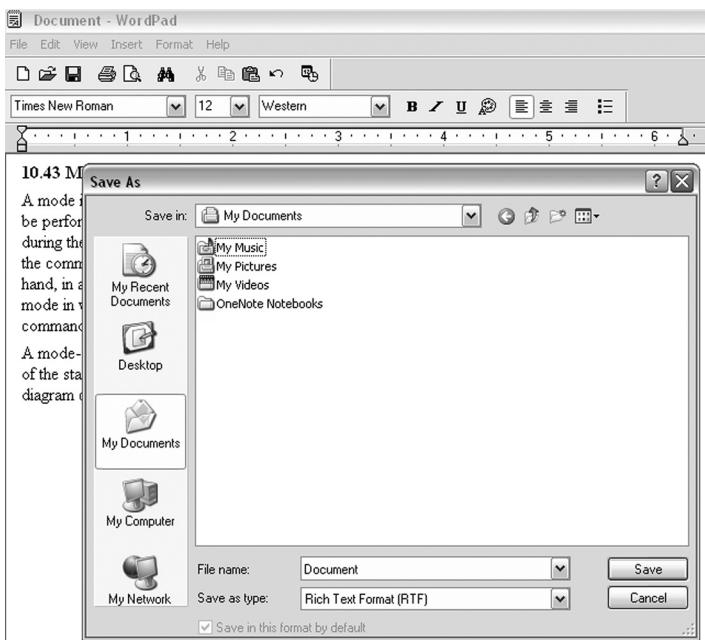


Figure 10.5 An example of mode-based interface

A window can be divided into two parts: client part and non-client part. The client area makes up the whole of the window, except for the borders and scroll bars. The client area is the area available to a client application for display. The non-client part of the window determines the look and feel of the window. The look and feel defines a basic behavior for all the windows, such as creating, moving, resizing etc.

GUI typically consists of a large number of windows. Therefore, it is necessary to have a systematic way to manage these windows. This is normally done through a window management system (WMS). It is primarily a resource manager. It keeps track of the screen area resources and allocates it to the different windows that seek to use the screen. From a broader perspective, a WMS can be considered as a UI management system (UIMS). It not only does resource management, but also provides the basic behavior of windows. It also provides several utilities for development of the UI.

Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, creating and manipulating icons etc. It determines how the windows should look and behave. It can be considered as a special type of client that uses the services (function calls) supported by the window system. The programmer can also directly invoke the services of the window system to develop the UI.

10.5 WIDGET-BASED GUI

Inputs (instructions and data) are fed to computer system through various controls available on the UI window. The user and computer communicate through the use of various standard controls called window objects or widgets. Development of the UI based on widgets is called component-based (or

widget-based) GUI development. Command Button, Option Button, Check Box, Text Box, List Box, Table and Grid etc. are some important widgets (window objects) used in GUI.

Command Button: When clicked with the mouse, the command button gets pushed and codes stored against the button are executed. This control is also called ‘Push Button’. Command buttons are shown in Figure 10.6a.



Figure 10.6a *Command buttons*

These are the primary ways through which the user can issue commands to the system within the dialog box. Command buttons should be properly labeled with meaningful and standard labels so that the users can know the major action of a particular button. Some standard labels used in command buttons are shown in Table 10.2.

Table 10.2 Standard Labels for Frequently used Actions

| Label | Action | Command Key |
|--------|-------------------------------------|---------------|
| OK | Makes changes and closes the window | Enter key |
| Cancel | Does not make changes | Escape key |
| Close | Closes the window | Alt + C |
| Reset | Resets to default settings | Alt + R |
| Apply | Makes changes | Alt + A |
| Help | Opens online help | Alt + H or F1 |

Buttons should be placed consistently either on the ‘top right’, ‘bottom right’ or ‘centred on the bottom’ of the window. Either a vertical or a horizontal design should be chosen for a particular window. The buttons should be positioned to match the design. A horizontal design should have buttons on the top right and a vertical design should have buttons on the bottom.

Option Buttons: If users have to pick one mutually exclusive choice from a list of options, option buttons are very convenient. They are sometimes called radio buttons. These are shown in Figure 10.6b. They are generally aligned vertically and labeled. They are used when the number of options is less, say not more than six.

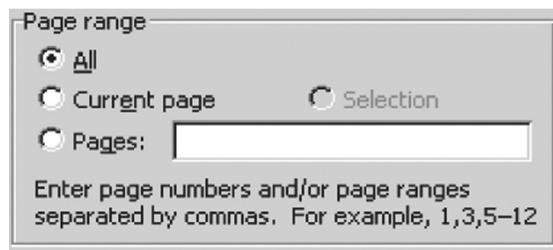


Figure 10.6b Option buttons

List Boxes: They are an alternative to long-option button lists. When there are more number of options, list boxes are preferred over option buttons. It is preferable that the list box shows at least three, but no more than eight, items at a time. The user can scroll through the list to choose an item. Use of a list box is already shown in scrolling menu in Figure 10.3. List boxes are also an alternative to data entry through keyboard. This is shown in Figure 10.6. Data entry list box ensures data integrity.

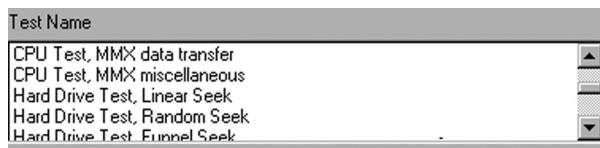


Figure 10.6c List box

Drop-down List Box: This is used if most users have a preference for one particular option from a list. That option is considered as the '*default option*' and is made to appear as the first item in the list, e.g. 'Times New Roman' in Figure 10.6d. List boxes hide all but the first option from the users. However, users have to go through an extra step to get to the rest of the list.

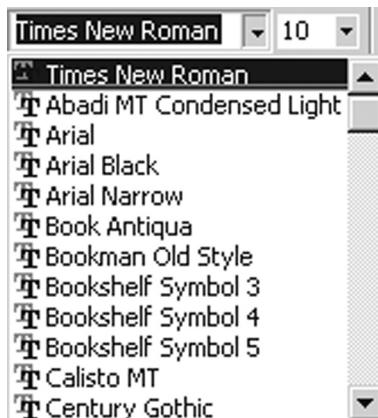


Figure 10.6d Drop-down list

Check Box: It replaces some data entry actions. It provides a quick way to make multiple choices. It can be used to choose one or more options. It can also be used for toggling purpose as shown in Figure 10.6e. Basically, on or off (yes or no) can be used here.

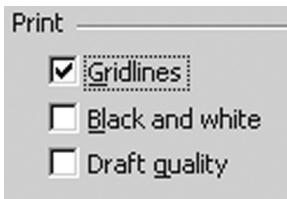


Figure 10.6e Check boxes

For check box ‘Gridline’ condition ‘on’ has been shown in the above figure as a tick mark. A clear descriptive label should be used so that users will understand each check box. Check boxes should be lined up vertically. Their number in a dialogue box should be limited to ten. They may be ordered properly based on criteria as given below:

- By frequency – most frequently used options at the top
- By task – if there is a usual order in which parts of a task are performed
- By logic – if there is a logical order, for instance a list of dates
- By alphabet – only use alphabetical order if the labels match the way your users think about the items

Text Boxes: These are the means for users to type in data. They should have a border to indicate the area where the users can enter or edit data, as shown in Figure 10.6. Each text box should be assigned a descriptive label.

Figure 10.6f Text box

Tables and Grid: Tables allow users to enter or view large amount of information at a time. If users need to compare data to make a selection, the data should be displayed in a tabular form (see Figure 10.7).

Grids are used for entering multiple data items at a time. Appropriate labels should be used for columns and rows that reflect the nature of data.

| Zone | 1st Qtr. | 2nd Qtr. | 3rd Qtr. | 4th Qtr. |
|-------|----------|----------|----------|----------|
| East | 20.4 | 27.4 | 90.0 | 29.4 |
| West | 30.6 | 38.6 | 34.6 | 31.0 |
| North | 45.9 | 46.9 | 45.0 | 43.9 |

Figure 10.7 Use tables for comparing data

Only the most frequently used GUI controls have been discussed in this chapter. There are many more controls available, which are beyond the scope of this book. Readers interested in knowing more details can refer to any standard text on GUI design. The summary of various widgets is described in Table 10.3.

Need for component-based GUI development: Presently, most of the UIs are component-based. Normally, any UI can be built from a few predefined components such as menus, dialog boxes, forms etc.

There are many advantages of using a widget-based interface. It permits the UIs for different applications to be built from the same basic components. Thus, it reduces the programmer's effort for development of the interface. Besides the standard components, the window system is a useful support available to the UI developers. The window system lets the programmer create and manipulate windows without having to write codes for basic window functions.

Users also have to learn only one type of standard control components (widgets). Users can extend their knowledge of standard components to other applications.

Table 10.3 Summary of Various Widgets

| Controls | Explanations | Features |
|------------------|---|---|
| Command Buttons | Users give commands such as OK, Cancel using this buttons | Label buttons consistently. Gray out unavailable buttons |
| Option Buttons | Users can pick one mutually exclusive choice from a list of options | Align option buttons vertically. Label option buttons descriptively. Limit option buttons to six or less. Choose an order. |
| List Boxes | These are alternative to long-option button lists | Show three to eight items at a time. Label each list box. Use drop-down list boxes to save space. |
| Check Boxes | Users can make multiple choices among various options. They can choose more than one option using this control. | Align check boxes vertically and label them descriptively. Limit check boxes to ten or less. Choose an order. |
| Text Boxes | Text boxes are the main way for users to type in data | Use a border to indicate data entry. Use box length to signify approximate data length. Align text boxes. Label all text boxes. |
| Tables and Grids | These allow users to enter or view large amount of information at a time | Use tables for comparisons among data. Use grids for multiple data entry. Label columns and rows. |

10.6 USER INTERFACE DESIGN

The characteristics of quality UI are already stated in the beginning of this chapter in Section 10.2. However, the users' satisfaction is the most important determinant of UI quality. Designing a quality UI is a creative process. However, certain tested and accepted principles serve as a useful guide for designing good quality UI. Most of these principles can be applied to both textual as well as GUI. The principles are given below:

1. The principle of user profiling: Since user satisfaction is the most important determinant of UI quality, it is important to know the 'profiles' of possible users. It is important to know details such as:

- What are the user's goals?
- What are the user's skills and experience?
- What are the user's needs?

It may be useful to come up with a list of *user dichotomies*, such as 'skilled vs. unskilled', 'young vs. old' etc., or some other means of specifying a continuum or collection of user types.

2. The principle of metaphor: It is based on the idea – 'Borrow behaviors from systems familiar to the users'. A complex software system is understood better if the UI is depicted in a way that resembles some commonplace system. For example, armor shield can be used as a metaphor for security setting. Similarly, a picture of the globe as icon can be used as a metaphor for the internet browser.

3. The principle of feature exposure: Some users prefer UIs that utilize the power of abstract models – command lines, scripts, plug-ins, macros etc. whereas others may prefer UIs that utilize perceptual abilities. In some cases, such as 'the toolbar', the program features are exposed by default. In other cases, such as 'the printer configuration dialog', they are brought up by the user's action. There are various levels of 'hiding'. A partial list is given below:

- Toolbar (completely exposed)
- Menu item (exposed by trivial user action)
- Submenu item (exposed by somewhat more involved user action)
- Dialog box (exposed by explicit user command)
- Secondary dialog box (invoked by button in the first dialog box)
- 'Advanced user mode' controls (exposed when user selects 'advanced' option)

4. The principle of coherence: 'Coherent' literally means 'stick together'. The UI should be *coherent*, i.e. it should be logical, consistent and easily followed. The behavior of the program should be internally and externally consistent. *Internal consistency* means that the program's behaviors make 'sense' with respect to other parts of the program. For example, if one attribute of an object (e.g. color) is modifiable using a pop-up menu, then it is to be expected that other attributes of the object would also be editable in a similar fashion.

External consistency means that the program is consistent with the environment in which it runs. One of the most widely recognized forms of external coherence is compliance with *UI standards*.

5. The principle of state visualization: Changes in behavior should be reflected in the appearance of the program. Each change in the behavior of the program should be accompanied by a corresponding change in the appearance of the interface. Similarly, when a program changes its appearance, it should be in response to a behavior change. A program that changes its appearance for no apparent reason will quickly teach the user not to depend on appearances for clues as to the program's state.

6. The principle of shortcuts: The UI should provide both concrete and abstract ways of getting a task done. Once a user has become experienced with an application, pre-memorized 'shortcuts' should be available to allow rapid access to more powerful features of software.

7. The principle of focus: Some aspects of UI may attract greater attention than others. Human eyes are drawn to animated areas of the display more readily than static areas. Changes to these areas are noticed readily. The mouse cursor is an intensely observed object. Users quickly acquire the habit of tracking it with their eyes in order to navigate. This is why global state changes are often signaled by changes to the appearance of the cursor. For example, the ‘hourglass cursor’ indicates that the system is busy. The text cursor is another example of a highly eye-attractive object. Changing its appearance signals some state changes.

8. The principle of grammar: UI uses a kind of language. Many of the operations within a UI require both a *subject* (an object to be operated upon) and a *verb* (an operation to perform on the object). The two most common grammars are:

- (1) Action → Object and (2) Object → Action

In ‘Action → Object’, the operation (or tool) is selected first. When a subsequent object is chosen, the tool immediately operates upon the object. The selection of the tool persists from one operation to the next, so that many objects can be operated on one by one without having to re-select the tool. ‘Action → Object’ is also known as ‘modality’, because the tool selection is a ‘mode’ that changes the operation of the program.

In ‘Object → Action’, the object is selected first and persists from one operation to the next. Individual actions are then chosen, which operate on the currently selected object or objects. This is the method seen in most word processors – first a range of text is selected, and then a text style such as bold, italic or a font change can be selected. Object->Action has been called ‘non-modal’ because all behaviors that can be applied to the object are always available.

9. The principle of help: There are five basic kinds of help, corresponding to the five basic questions that users ask:

- | | |
|-------------------------------|--|
| 1. Goal-oriented Help: | It describes the kinds of things that can be done with the application of software |
| 2. Descriptive Help: | It describes various components and features of software |
| 3. Procedural Help: | It describes the procedures about how to operate the software |
| 4. Interpretive Help: | It explains the reason for any action/response that happens during the operation of the software |
| 5. Navigational Help: | It describes how to reach different features/parts of the software |

Each of these questions requires a different sort of help interface. Hence, depending on the type of users, a trade-off may be done to provide help for each of these requirements.

10. The principle of safety: UI should provide confidence to users by providing a safety net. Novice users need to be assured that they will be protected from their own lack of skill. A program with no safety net will make this type of user feel uncomfortable or frustrated to the point that they may cease using the program. The ‘Are you sure?’ dialog box and multi-level undo features are vital for this type of user. It is important for new users that they feel safe. However, an expert user does not need these messages. He may be annoyed by these messages that are unnecessary for him. UI should permit expert users to turn off what they feel as unnecessary safety features and use the software as maestros.

11. The principle of context: Each action of the user takes place within a given context, i.e. in the context of current document, current selection, current dialog box. A set of operations that is valid in one context may not be valid in another. For example, in a drawing application, selecting a text object (text box) is a different state from selecting individual characters within that text object.

It is usually a good idea to avoid mixing these levels. For example, consider a word processor application that allows users to select multiple (range of) text characters of a document file. However, the software usually does not allow selection of text characters and document files at the same time. One way to do this is to 'dim' the selection that is not applicable in the current context.

Sometimes users may shift from one context to another. For example, they may be working in a particular task, when a dialog-box pops up asking them for confirmation of an action. This sudden shift of context may confuse them. Here, the wording of the dialog box is important. Instead of a confirmation message 'Are you sure?' a little more elaboration like 'The document has not been saved. Do you want to quit anyway?' would help the users to keep track of their current context.

12. The principle of aesthetics: UI should be attractive and appealing to users. Hence, aesthetics aspects should be given due importance. The icons should be aligned. Color and symmetry should be used properly to enhance the beauty of the interface.

13. The principle of user testing: In many cases a good software designer can spot fundamental defects in a UI. However, there are many kinds of defects that are not so easy to spot. It has been observed that the experienced software designers are often less likely to spot errors than the common person/user.

UI testing is the testing of UIs by actual end-users. It has been shown to be an effective technique for discovering design defects.

User testing can occur at any time during the project; however, it is often more efficient to build a mock-up or prototype of the application and test that before building the real program. It is much easier to deal with a design defect before it is implemented than after.

14. The principle of humility: It is important to realize that a user or a developer has only part of the picture. Hence, the designer should listen to what ordinary people have to say. The ideal is to take a lot of users' opinions. A single designer's intuition about good and bad features of any UI is insufficient.

SUMMARY

UIs can be classified into two major categories: (1) Text-based TUI and (2) Iconic or GUI. In TUI, interaction with a computer is done by typing commands. In GUI, the user issues commands by performing actions on the visual representations of the objects. GUI is also called Direct Manipulation interface or Iconic interface.

TUI could be either command language based and/or menu based. In a command language-based TUI, unique names are assigned to the different commands, called primitive commands. Command language-based interfaces usually allow users to compose complex commands by adding optional parameters to the primitive command.

The menu-based interface permits the user to select the command (options) from a number of commands displayed on the screen. Hence, the user is not required to remember the commands and their syntax. TUI and GUI can both be menu based.

GUI presents the interface to the user in the form of visual models (i.e. icons or objects). The user issues commands by selecting objects by pointing a device. GUI is a direct manipulation interface. The 'WIMP' system is the most popular form of GUI. The scrolling menu, the walking menu and the hierarchical menu, are the three important types of menus in GUI. An interface can be either mode-less or mode-based.

GUI typically consists of a large number of windows. The windows are managed by WMS. It is also called UIMS. Window manager is the component of WMS, with which the users can perform various window-related operations.

In component-based GUI development, UIs are built from a handful of standard components such as Command Buttons, Option Buttons, Check Boxes, Text Boxes, List Boxes etc. These components are called window objects or widgets.

The window system helps the programmer to create and perform various window functions.

A good quality UI should have certain desired characteristics such as being user-friendly, attractive and pleasant to work on, permitting fast entry of data and commands, features to prevent errors etc. Users' satisfaction is the most important determinant of UI quality. The design of UI has some principles that help in designing good quality UI.

EXERCISES

1. Explain the importance of UI for software.
2. List the desirable characteristics of a good UI.
3. Compare the relative advantages and disadvantages of command-based textual interface and menu-driven textual interface.
4. Differentiate between primitive command and composed command.
5. Differentiate between a mode-based interface and a mode-less interface.
6. Compare the advantages of GUI over textual interface.
7. Write how the commands are issued in an iconic interface.
8. Compare the relative advantages of scrolling menu, hierarchical menu and walking menu.
9. What is WMS? What are its advantages?
10. Explain the role of window manager in WMS.
11. Explain the reason for the popularity of component-based GUI development.
12. Name some popular widgets used in GUI.
13. List some standard labels used in command buttons.
14. Write the use of Option Buttons, Check Boxes and List Boxes in a GUI.
15. Distinguish between list box and drop-down list box.
16. Distinguish between Text Boxes, Tables and Grids.
17. List various principles of UI design. Explain the principle of user profiling and the principle of metaphor.
18. List a few metaphors that can be used for UI design.
19. What are various types of help that can be provided to facilitate user interaction?

This page is intentionally left blank.

CODING AND DOCUMENTATION

Good coding and documentation are necessary for reliable and maintainable software. The chapter covers the following topics:

- Coding standard
- Guidelines for good coding practices
- Software documentation
- Documentation standards and guidelines
- CASE tools

Many software industries have developed their own standards to be followed rigorously by their people. Sometimes clients want coding and documentation to be done in certain ways. Hence, there are several standards that exist in the programming industry. This chapter discusses various aspects of coding and documentation, but not any one particular standard or guidelines.

Software is developed to be used for many years. However, it needs modifications from time to time to meet the changing requirements of users brought about by continual changes in organization systems and its environment. Minor modification or customization of software is called software maintenance. The software personnel sometimes spend more than 50% of their time on maintenance activities. Sometimes the cost of maintenance exceeds the cost of the software. Hence, *Maintainability* is an important quality parameter of software. Maintainability can be defined as ease with which the software permits itself to be modified to suit the changing needs of users. For maintainability the software should have following features:

- Software should have modular structure.
- The source code should be in a form to facilitate easy understanding.
- There should be proper documentation relating to different aspects of the software.

The aspects relating to coding and documentation are discussed in this chapter.

11.1 CODING STANDARDS

Coding is done by several programmers. Hence, for consistency and ease of understanding, it is desirable that all programmers should follow a well-defined and common style of coding called ‘Coding Standard’.

Adherence to a coding standard has the following advantages:

- It gives a uniform appearance to the codes written by different engineers.
- Consistency is maintained throughout the program.
- It makes the code easier to understand for debugging and modification.
- It encourages good programming practices.
- It improves productivity.

A coding standard prescribes rules and guidelines about declaration and naming of variables, structure of the code (i.e. formatting features), error return conventions etc. Most software development organizations have their own coding standards and require their programmers to rigorously follow these standards.

There are three types of coding standard.

- i. **General coding standard:** These standards have general applicability for program codes across all projects written in any language.
- ii. **Language-specific coding standards:** These are specific to a particular programming language. Since features and syntax of each programming language are different, there is need for language-specific coding standards. Language standards are designed such that these supplement, rather than override, general coding standards.
- iii. **Project-specific coding standards:** These standards are specific to a particular project. Project coding standards should supplement, rather than override, general coding standards and language coding standards. A separate coding standard for each project is not desirable, because people take time to adapt to a particular standard. Also, if projects follow different coding standards, it is difficult to reuse program codes across projects.

The software firms develop their own coding standards based on the type of software they develop, type of platform and programming language they use to develop the software and programming style that their people are most conversant with. Sometimes the coding standard that must be followed for software development is specified in the contract. However, some generally accepted principles and guidelines are usually followed in developing a coding standard.

11.1.1 Formatting Features

Formatting refers to the way statements are placed in a program to enhance its appearance and readability. It is done by proper spacing between words, indentation, wrapping of lines, use of braces (brackets) etc.

Spacing: The proper and consistent use of spaces within a line of code can enhance readability. Some general conventions for spacing are as follows:

- A keyword followed by a parenthesis should be separated by a space.
- A blank space should appear after each comma in an argument list.
- All binary operators except ‘.’ should be separated from their operands by spaces.

- Blank spaces should never separate unary operators such as unary minus, increment ('+') and decrement ('-') from their operands.
- Casts should be made followed by a blank space.

Indentation: Proper and consistent indentation is important in producing easy to read and maintainable programs. Indentation is used for emphasizing the body of a control statement such as a loop, a select set of statements, a conditional statement or a new scope block.

Generally, indenting by three or four spaces is considered to be adequate. However, the indentation spacing should be applied throughout the program. Generally, it is not considered a good practice to use tabs for indentation purposes.

Wrapping Lines: When an expression cannot fit in a single line, it should be broken and continued in the next line according to following principles:

- In case of statements, break the line after a comma.

Example: `fprintf(stdout , “\nThere are %d reasons to use standards\n” , num_reasons);`

- In case of mathematical expressions, break the line after an operator.

Example: `long int total_marks = marks_in_english + marks_in_maths + marks_in_computer;`

- In case of mathematical expressions, break the line after a higher-level operator instead of a lower-level operator.

Example: `total_salary = basic_salary + basic_salary * (da_rate + hra_rate) + grade_pay;`

Instead of

`total_salary = basic_salary + basic_salary * (da_rate + hra_rate) + grade_pay;`

- In case of mathematical expressions, align the new line with the beginning of the expression of the previous line.

Example: `total_salary = basic_salary + basic_salary * (da_rate + hra_rate) + grade_pay;`

Instead of

`total_salary = basic_salary + basic_salary * (da_rate + hra_rate) + grade_pay;`

Variable Declarations: Variable declarations that span multiple lines should always be preceded by a type.

Example:

Acceptable:

```
int price , score;
int price;
int score;
```

Not Desirable:

```
int price ,
```

```
score;
```

```
int score;
```

Use of Braces: In some languages, braces are used to delimit the bodies of conditional statements, control constructs and blocks of scope. A preferable style for bracing is given below.

```
for (int j = 1; j < last_number; j++)
{
/* This set of statements are executed in a loop */
}
```

The same bracing style should be consistently used throughout the code. Braces shall be used even when there is only one statement in the control block.

Use of Parentheses: It is better to use parentheses liberally. For example it is preferable to write

`net_price = gross_price - (gross_price * discount_rate);`

Instead of

`net_price = gross_price - gross_price * discount_rate;`

Contents of headers preceding codes for different modules: The information contained in headers of different modules should be standard for an organization. The exact format in which header information is organized in the header can also be specified. The following are some standard header data:

- Name of the module
- Date on which the module was created
- Author's name
- Modification history
- Purpose of the module
- Different functions with their input/output (I/O) parameters
- Global variables accessed/modified by the module

11.1.2 Naming Convention

The name of the source file or script shall represent its function. All of the routines in a file shall have a common purpose.

Variables shall have mnemonic or meaningful names that convey the intent of its use to a casual observer. Variables shall be initialized prior to their first use. To differentiate between different types of variables, a possible naming convention can be as under.

Global variable names always start with a capital letter, local variable names are made of small letters and constant names are always capital letters.

Names of classes, subroutines, functions and methods should start with a verb followed by a noun. The names shall specify an action, e.g. 'get_name', 'compute_tax'.

Standard letters should be prefixed with the names of different graphical user interface (GUI) elements. This helps to identify different GUI elements. Standard prefixes for some important GUI elements are given in Table 11.1.

Table 11.1 Standard prefixes for some GUI elements

| UI element | Prefix | UI element | Prefix |
|----------------|--------|--------------|--------|
| Label | Lbl | DropDownList | Ddl |
| TextBox | Txt | RadioButton | Rdo |
| DataGridView | Dtg | Image | Img |
| Command-Button | Cmd | Table | Tbl |
| Checkbox | Chk | Hyperlink | Hlk |
| ListBox | Lst | | |

11.1.3 Coding Norms

Coding norms relate to good coding practices. These are included in coding standard as a guideline for good coding practices.

Line Length: It is considered good practice to keep the lengths of source code lines at or below 80 characters. Lines longer than this may not be displayed properly on some terminals and tools.

Structured Programming: Coding should be done in a structured or modular manner. The 'GO TO' statements should not be used as it makes the program hard to maintain.

Size of Classes, Subroutines, Functions and Methods: The size of subroutines, functions and methods should not be more than a certain limit. This depends upon the language being used. At the lowest level, each module should be constrained to one function or action. If a module is too large, it is usually because the programmer is trying to accomplish too many actions at one time. In that case, the task being performed can be broken down into subtasks.

Program Statements: They should be limited to one per line. Also, nested statements should be avoided when possible.

Inline Comments: They help a person to quickly understand a program code. Thus, they reduce the amount of time required to perform software maintenance tasks.

Inline comments appear in the body of the source code itself. They are used to describe the task being performed by a block of code. They can also be used to explain the logic or parts of the algorithm. However, too many inline comments are not desirable. They should make up a maximum of 20% of the total lines of code in a program, excluding the header documentation blocks.

Meaningful Error Messages: Error handling is an important aspect of computer programming. Error messages should be meaningful. When possible, they should indicate what the problem is, where the problem occurred and when the problem occurred. Error messages should be stored in a way that makes them easy to review. For non-interactive applications, the error messages should be logged into a log file. Interactive applications can either send error messages to a log file, standard output or standard error. Interactive applications can also use popup windows to display error messages.

Error return conventions and exception handling mechanisms: The way error conditions reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

Compiler Warnings: Compilers often issue two types of messages: warnings and errors. Compiler warnings normally do not stop the compilation process. The program may compile and work even if there is a warning message. The compiler can be set to suppress warning messages and continue compiling. However, this practice should be discouraged. A warning message often indicates some problem that may affect the behavior, reliability and portability of the code. Hence, compiler and linker warnings shall be treated as errors and fixed.

Rules for use of global data and variable: These rules list what types of data can be declared global and what cannot.

11.2 CODING GUIDELINES

Coding guidelines provide the programmer with a set of best practices that can be used to make program codes easier to read and maintain. Unlike coding standards, the use of these guidelines is not mandatory. However, the programmer is encouraged to review them and attempt to incorporate them into his programming style.

The following are some representative coding guidelines recommended by many software development organizations:

Do not use a coding style that is too clever or cryptic. Code should be easy to understand. Many computer professionals actually take pride in writing cryptic and incomprehensible code. Cryptic coding can obscure the meaning of the code and hamper understanding. It also makes maintenance difficult.

Avoid obscure side effects. The side effects of a function call include modification of parameters passed by reference, modification of global variables and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O operation is performed, which is difficult to infer from the name of function or its header information, it becomes difficult for anybody trying to understand the code.

Do not use an identifier for multiple purposes. Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. When an identifier is being used for multiple purposes, it is not possible to give it a descriptive name indicating its purpose. Use of a variable for multiple purposes can also lead to confusion. It makes the code difficult to understand and makes future enhancements more difficult.

The length of any function should not exceed a limit (say 20 source lines). A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

Do not use 'go to' statements. Use of 'go to' statements makes a program unstructured and makes it very difficult to understand.

Other aspects: Coding guidelines prescribe rules relating to various aspects of coding such as use of macros, defining variable constants, information hiding, domain and scope of variables etc.

Efficiency and maintainability are two important aspects to programming. These two goals may conflict with each other. Sometimes program codes that use tricky logic and complex algorithms are more efficient, but are hard to follow and maintain. The programmer needs to carefully weigh efficiency gains versus program complexity and readability. If a more complicated algorithm offers only small gains in the speed of a program, the programmer should consider using a simpler algorithm. Although slower, the simpler algorithm will be easier to understand. Coding guidelines prescribe rules for improving the overall quality of a code.

11.3 SOFTWARE DOCUMENTATION

A document is used for explaining some attributes of an object, system or procedure. It describes the structure and components, applications, procedure for operation and maintenance of a system/product. Software documentation is usually in the form of paper books or computer readable files (such as HTML pages). There are different types of software documentation. These are:

1. Requirements documentation
2. Design documentation
3. Program documentation
4. User documentation
5. Marketing documentation

Requirements documentation It is the description of what the software should do. It is used by different stakeholders such as software designers, programmers, testers, end users, project managers, sales and marketing people etc. It serves many purposes.

Requirements documentation is done in a variety of styles, notations and formality. These can be specified as narratives in natural language, as mathematical notations, as drawn figures and as a combination of them all. The content of a Software Requirement Specification (SRS) document is already discussed in an earlier chapter.

Design documentation Software design document is a written description of a software product. It usually comprises an architecture diagram with pointers to detailed feature specifications of smaller pieces of the design. Design documentation is done in two parts: (1) high-level design document and (2) low-level design document.

High-level design document gives the architectural view or overall structure of the software. Low-level design document provides the detail for constructing the software. The traditional Function-Oriented approach and Object-Oriented approach are two approaches for development of design documentation. These are already covered in earlier chapters.

Program documentation It gives a comprehensive procedural description of a program. It shows how software is written. It helps in maintenance and any future modification of the program. It describes what exactly a program does by mentioning the requirements of the input data and the effect of performing a programming task.

Program documentation is written for the sake of programmers who write the actual software and may have to modify it or use it as a part of another program, which they write in the future. While end-user documentation uses a simple user-friendly language to describe the software, program documentation describes things in a language that is more appropriate to the programmer.

There are several software tools that can be used to auto-generate the code documents. These tools extract the comments and software contracts from the source code and create reference manuals in text or HTML files. Code documents are often organized into a *reference guide* style, allowing a programmer to quickly look up an arbitrary function or class. Doxygen, NDoc, javadoc, EiffelStudio, Sandcastle, RO-BODoc, POD, TwinText are some software tools for auto-generating program documentation.

User documentation It includes hard-copy or electronic manuals that enable users and operators to interact successfully with the system. It normally consists of a user manual. The manual provides instructions for running the software. It contains descriptions of software commands, several useful examples of applications and troubleshooting guide to help with difficulties.

Typically, user documentation describes each feature of the software, and assists the user in realizing these features. A good user document should also provide thorough troubleshooting assistance. It is very important for user documents to be lucid and unambiguous. User documentation is considered to constitute a contract specifying what the software will do. There are three broad ways in which user documentation can be organized.

1. **Tutorial:** A tutorial approach is considered the most useful for new users, in which they are guided through each step of accomplishing particular tasks.
2. **Thematic:** A thematic approach is one where chapters or sections concentrate on one particular area of interest. It is more useful to a user who has some basic knowledge of the software. Some authors prefer to convey their ideas through a knowledge-based article to facilitate users' needs.
3. **List or Reference:** In this type of organizing, commands and tasks are listed alphabetically or logically grouped via cross-referenced indexes. The logical grouping via cross-referenced indexes is very useful to advanced users who know exactly what sort of information they are looking for.

A good user documentation should have the right balance of all the three approaches.

Marketing documentation For many applications it is necessary to have some promotional materials to encourage potential customers/users to spend more time learning about the software product. This form of documentation has three purposes:

1. To excite potential users about the product and instil in them a desire for becoming more involved with it
2. To inform them about what exactly the product does, so that their expectations are in line with what they will be receiving
3. To explain the position of this product with respect to other alternatives

One good marketing technique is to provide clear and memorable *catch phrases* that exemplify the point one wishes to convey, and also emphasize the interoperability of the program with anything else provided by the manufacturer.

11.4 DOCUMENTATION STANDARD AND GUIDELINES

The user documentation supplied with the software is an important element for evaluation of software quality. There are standards and guidelines for developing good user documentation.

11.4.1 Structure of Software User Documentation

It includes how it is organized into segments and in what order the segments are presented. It should aid the user in locating and understanding the information content.

Documents should be structured into units with unique content. A document is structured into logical units called *chapters*. The *chapters* are subdivided into *topics*, and *topics* into *subtopics*. The user document may consist of the following components:

- Identification data (title page)
- Table of contents
- List of illustrations
- Introduction
- Information on how to use the documentation
- Concept of operations (i.e. what are the operations that can be performed)
- Procedures
- Information on software commands
- Error messages and problem resolution
- Glossary
- Related information sources
- Navigational features
- Index

Usually, software has different categories of users having widely differing needs. For this, the user documentation can be structured in one of the following ways:

- Separate sections devoted to the needs of specific users. The users for whom the section is intended should be mentioned in the beginning of each section, allowing each user to identify the sections of his interest.

- Separate documents or document sets for each specific audience

11.4.2 Usage Mode

The content of documentation is related to its usage mode. Users of software need documents either to learn about the software (instructional mode) or to refresh their memory about it (reference mode).

Instructional mode documents may be either: (1) information-oriented or (2) task-oriented.

Information-oriented documents instruct the user on the concepts and technical information needed to use the software properly. Task-oriented documents show the user how to operate the software to complete the required tasks. It should include procedures structured according to the user's tasks. Related tasks should be grouped in the same chapter or topic.

Reference mode documentation should be arranged to facilitate random access to individual units of information. It is usually done in an alphabetic order. The reference mode documentation may also be made context-sensitive and integrated into the software, for example, pop-up or drop-down lists of acceptable data values or commands.

11.4.3 Information Content of Software User Documentation

The user document should provide information on use of the document, operations and procedures, software commands, error messages and problem resolution, technical terminology related to software and related information.

Initial components: Identification data, a table of contents, list of illustrations (if applicable) and an introduction constitute the initial component of documentation.

Identification data shall include the following:

- a) Documentation title
- b) Documentation version and date published
- c) Software product and version
- d) Issuing organization
- e) Operating environment(s), i.e. hardware, communications and operating system(s)
- f) Copyright and trademark notices
- g) Restrictions on copying or distributing the documentation

Identification data shall appear on a package label, legible without opening the package and on a title page. Each document in a document set should also have a unique title page. The title of the document should indicate its nature and scope. Generally, abbreviations or acronyms should not be included in the title.

If the document has more than eight pages, it should usually have a table of contents. The table of contents shall list the headings of the chapters or topics of a document with an access point (i.e. page number or an electronic link) for each. A table of contents may include only the first-level headings or up to second- and third-level headings. A comprehensive table of contents lists all chapters or topic titles (headings) down to the third level. Electronic documentation may display tables of contents in expandable and collapsible formats to provide top-level and detailed access to headings.

If the document contains more than five illustrations and the illustrations are not visible at the same time as text references to them, it is preferable to have a list of illustrations (including both tables and figures). The titles in the list of tables, figures or illustrations shall be identical in wording to those in the document.

The introduction should describe the intended audience, scope and purpose of the document and include a brief overview of the software purpose, functions and operating environment. Introductions should also be provided for each chapter and topic.

Information on use of the documentation: The documentation shall include information on how it is to be used (i.e. example, help on help) and an explanation of the notation. In document sets comprising many volumes or products, this information may be provided in a separate 'road map' or guide to the document set.

Concept of operations: Documentation shall explain the conceptual background for use of the software, using such methods as a visual or verbal overview of the process or workflow; or the theory, rationale, algorithms, or general concept of operation. Explanations of the concept of operation should be adapted to the expected familiarity of the users with any specialized terminology for user tasks and software functions. Documentation shall relate each documented function to the overall process or tasks. Conceptual information may be presented in one section or immediately preceding each applicable procedure.

Information on procedures: Instructional mode documentation provides directions for performing procedures. Instructions shall include general information for use of software, preliminary information, instructional steps and completion information.

General information may include instructions for routine activities that are applied to several functions:

- Software installation and de-installation, if performed by the user
- Access, or log-on to and sign-off from the software
- Navigation through the software to access and exit from functions
- Data operations (enter, save, read, print, update and delete)
- Methods of cancelling, interrupting and restarting operations

These common procedures should be presented in a common section to avoid redundancy.

Preliminary information common to several procedures may be grouped and presented once to avoid redundancy. Preliminary information for instructions shall include the following:

- A brief overview of the purpose of the procedure and definitions or explanations of necessary concepts not included elsewhere
- Identification of technical or administrative activities that must be done before starting the task
- A list of materials the user will need to complete the task, which may include data, documents, passwords, additional software and identification of drivers, interfaces or protocols
- Relevant warnings, cautions and notes that apply to the entire procedure

Instructional steps should indicate the expected result or system response. Instructional steps should provide references to explanations of error messages and recovery procedures. Instructional steps shall be presented in the order of performance.

Completion information for instructions should indicate how the user can determine whether the procedure has successfully completed, and how the user can exit from the procedure.

Information on software commands: Documentation should explain the formats and procedures for user-entered software commands, including required parameters, optional parameters, default options, order of commands and syntax. Reference mode documentation should contain a reference listing of all reserved words or commands. Documentation should explain how to interrupt and undo an

operation during execution of a command. Documentation should also describe how to recognize that the command has successfully executed.

Information on error messages and problem resolution: Documentation should address all known problems in using the software in sufficient detail such that the users can either recover from the problems themselves or clearly report the problem to technical support personnel. Reference mode documentation shall include each error message with an identification of the problem, probable cause and corrective actions that the user should take.

Information on terminology: Documentation may include a glossary. The glossary should include an alphabetical list of terms and definitions. Terms included in the glossary should also be defined on their first appearance in the printed documentation.

Information on related information sources: Documentation may contain information on accessing related information sources, such as a bibliography, list of references or links to related web pages.

Placement of critical information: Critical information should be placed in a prominent location in the documentation. General warnings or cautions that apply throughout the use of the software or documentation shall appear in the initial components. Specific warnings and cautions shall appear on the same page or screen and immediately before the procedure or step that requires care.

11.4.4 Format of Software User Documentation

The documentation format includes presentation conventions for stylistic, typographic and graphic elements. The format for documentation should be accessible and usable in the expected users' work environment.

Consistency of terminology and formats: Documentation shall use consistent terminology throughout a document set for elements of the user interface, data elements, field names, functions, pages, topics and processes. The same formatting conventions should also be applied consistently through the documentation. However, the documentation may use special formatting to identify new or changed content. Formatting conventions includes colors, borders, indenting, spacing and font variations.

Use of printed or electronic formats: The documentation may be in electronic form. However, in addition to electronic form, documentation should also be presented in printed form regarding the following information:

- Hardware, operating system and browser software requirements for the software and documentation
- Installation information, including recovery and troubleshooting information for installation instructions
- Instructions for starting the software
- Instructions for access to any electronic documentation
- Information for contacting technical support or performing recovery actions available to users

The system should also enable printing of electronic documentation. The electronic documentation should be available online whenever the user wants it during an operation. The user should be able to perform an operation and simultaneously refer to relevant function-specific electronic documentation. The user should be able to view the online documentation and navigate to related software functions during system operations.

Legibility: The documentation should be legible to the user. It shall use a font style and color that is legible against the expected background (paper color or screen background color). For legibility the font

size should not be less than 7.5 points (i.e. 3 mm approximately). Usually, the uppercase letters should not be used for continuous text of more than 25 words.

Legibility may be affected by output devices (monitors and printers) that are monochrome, have limited resolution, render colors differently or support a limited range of colors. Some output devices may apply substitute fonts or special characters if the specified font is not available. Distinctions that depend on more than two gradations of colors or shades of gray may not be visible.

Formats for warnings, cautions and notes: Warnings, cautions and notes shall be displayed in a consistent format that is readily distinguishable from ordinary text or instructional steps. The flag word (for example, *warning*, *caution* or *note*) shall precede the accompanying text. These may be identified by consistent and distinct graphics symbols (e.g. an exclamation point inside a triangle). A warning or caution should include the following parts:

- Flag word
- Graphic symbol
- Brief description of the hazard
- Instructional text on avoiding the hazard
- Consequences of incurring the hazard
- User's remedies for the adverse consequences, if any

11.4.5 Format for Instructions

Instructional steps shall be consecutively numbered. A consistent numbering or lettering convention should be established for sub-steps, alternative steps and repeated procedures.

Formats for representing user interface elements: GUI elements of the software, such as buttons, icons, variable cursors and pointers; special uses of keyboard keys or combinations of keys; and system responses should be represented in documentation by consistent graphic formats so that the various elements are each distinguished from the text.

Index: An index is an alphabetical listing of keywords, graphics or concepts with an access point for each. Printed documents of more than 40 pages should include an index, whose access points may be page numbers, topic numbers, illustration numbers or references to another index entry. Electronic documents of more than 40 topics shall include either a search tool or an index whose access points are electronic links. The index is placed at the end of the documentation.

11.5 CASE TOOLS

CASE tool stands for ‘Computer Assisted Software Engineering’. It automates methods for designing, documenting and producing structured computer code in the desired programming language.

The implementation of a new system requires many tasks to be organized and completed correctly and efficiently. CASE tools were developed to automate these processes and to ease the task of coordinating the events that need to be performed in the system development life cycle. For example, prototyping tools can be used to develop graphic models of application screens to assist end users to visualize how an application will look after development. Subsequently, system designers can use automated design tools to transform the prototyped functional requirements into detailed design documents. Programmers can then use automated code generators to convert the design documents into code.

11.5.1 Upper and Lower CASE Tools

CASE tools can be classified into two types on the basis of the activities they support in the software development life cycle. These are:

1. Upper CASE tools
2. Lower CASE tools

Upper CASE Tools: During the initial stages of system development, analysts are required to determine system requirements and analyze this information to design the most effective system possible. To complete this task, an analyst uses some standard documentation tools like data flow diagrams, data dictionaries, process specifications, documentation and structure charts. When completing these tasks manually, it becomes very tedious to have to redraw the diagrams each time a change is made to the system. CASE tools allow these types of changes to be made very quickly and accurately. In manual methods, even some minor change made in one diagram may require changes to be made in many corresponding diagrams or documents. For a very large system, one may forget to make the corresponding changes in all documentation, leading to an erroneous representation of the system. By using CASE tool's analysis feature, information shared throughout the flowcharts and documentation can be checked against each other to ensure that they match.

CASE tools are also very helpful tools to use during the design phase of system development. CASE provides tools to help develop prototype screens, reports and interfaces. These prototypes can then be checked and approved by the users and management. This avoids the problem of having to redesign the interfaces during the implementation phase, which users generally do not like.

Hence, Upper CASE tools deal with the first three parts of the system development life cycle, namely preliminary investigation, analysis and design. These are also referred to as Front-End CASE tools.

Lower (Back-End) CASE Tools: Lower CASE tools are most often used as an aid to generate program codes. Fourth generation programming languages and CASE tools code generators measurably reduce the time and cost needed to produce the code necessary to run the system. Code generated through CASE tools is also easy to maintain and is portable to other hardware platforms.

Fourth generation program code is also much easier to test. Since fourth generation code is free from rigid syntax rules, the programmer can focus more on logic of the program. The maintenance and subsequent modifications are also easier than program written in third generation language.

Code generators also have the feature that they are able to interact with the upper CASE tools. Information that was stored from the upper CASE tools can be accessed using the code generators to aid in the development of the code. Code generators also allow for specialized code to be inserted into the generated program code. This allows special features to be designed and implemented into the program.

Lower CASE tools deal mainly with the implementation and installation of software. They concentrate on the back-end activities of the software life cycle and hence support activities like physical design, debugging, construction, testing, integration of software components, maintenance, re-engineering and reverse engineering activities. These are also referred to as Back-End CASE tools.

11.5.2 CASE Workbenches and Environments

A set of CASE tools are generally designed to be compatible with each other, so that information generated from one tool can be passed to other tools. Thus, the subsequent tool in turn can use the information to complete its task and then pass the new information back to the system to be used by other tools. This allows for important information to be passed very efficiently and effectively between many planning

tools. For the CASE tools to be effective it is necessary that the information are correctly passed between stages and not lost in transit during the development process of a system. This is achieved by CASE Workbenches and CASE Environment.

Workbenches integrate several CASE tools into one application to support specific software-process activities. Hence, they achieve:

- A homogeneous and consistent interface (presentation integration)
- Easy invocation of required tools from CASE workbench application
- Access to a common data set managed in a centralized way (data integration)

CASE workbenches can be classified into following types:

1. Business planning and modeling
2. Analysis and design
3. User-interface development
4. Programming
5. Verification and validation
6. Maintenance and reverse engineering
7. Configuration management
8. Project management

A CASE environment is a collection of CASE tools and workbenches that supports the software process. CASE environments are classified based on the focus or on the basis of integration as given below:

1. Toolkits
2. Language-centered
3. Integrated
4. Fourth generation
5. Process-centered

Toolkits: Toolkits are loosely integrated collections of products easily extended by aggregating different tools and workbenches. Typically, the support provided by a toolkit is limited to programming, configuration management and project management. The toolkits are usually loosely integrated and require users to activate tools by explicit invocation. The resulting files are unstructured and could be in a different format. The access of a file from different tools may require only the conversion of the file format. This being a minor constraint, toolkits can be easily extended incrementally.

Language-centered: The environment itself is written in the programming language for which it was developed, and thus enables users to reuse, customize and extend the environment. Integration of code in different languages is a major issue for language-centered environments. Lack of process and data integration is also a problem. The strengths of these environments include a good level of presentation and control integration. Interlisp, Smalltalk, Rational and Knowledge Engineering Environment (KEE) are examples of language-centered environments.

Integrated: These environments achieve presentation integration by providing uniform, consistent and coherent tool and workbench interfaces. Data integration is achieved through the *repository* concept: they have a specialized database managing all information produced and accessed in the environment. IBM AD/Cycle and Digital Equipment Corporation (DEC) Cohesion are examples of an integrated environment.

Fourth generation: Fourth generation environments were the first integrated environments. These are sets of tools and workbenches supporting the development of a specific class of programs such as electronic data processing and business-oriented applications. Fourth generation environments include programming tools, configuration management tools, document handling tools and code generator to produce code in lower level languages. Informix 4GL and Focus are examples of fourth generation environments.

Process-centered: Environments in this category focus on process integration with other integration dimensions as starting points. A process-centered environment operates by interpreting a process model created by specialized tools. It usually consists of tools handling two functions:

- Process-model execution and
- Process-model production

Enterprise II, East, Process Wise, Process Weaver and Arcadia are examples of process-centered environments.

11.5.3 Advantages of CASE Tools

CASE tools also allow analysts to allocate more time to the analysis and design stages of development and less time coding and testing. It was estimated that prior to introduction of CASE tools, for business applications, up to 70% of the time was used to develop code and testing. CASE tools have considerably reduced this time and allowed analysts to devote more time on analysis and design activities. This resulted in systems that closely mirrored the requirements of the users. Thus, CASE tools have helped to develop more efficient and effective systems.

Business organizations generally operate in a dynamic environment where business needs, procedures and process keep changing. Hence, it is important that the development of commercial software is fast enough to save itself from becoming obsolete. Business organizations need new systems that match the current business needs, to be developed quickly, so that they can compete more effectively in the market. In a highly competitive market, staying on the leading edge can make the difference between success and failure.

Current trends are showing a significant decrease in the cost of hardware with a corresponding increase in the cost of computer software. Developing effective software packages takes the work of many people and can take years to complete. CASE tools are an important part of resolving the problems of application development and maintenance. CASE tools significantly alter the time taken by each phase and the distribution of cost within the software life cycle. Software engineers are now placing greater emphasis on system analysis and design in which the CASE tools are also a great help. Much of the code can now be generated automatically. The use of CASE tools has resulted in considerable reductions in maintenance costs. Another powerful feature of CASE tools lies in their central repository, which contains descriptions of all the central components of the system. These descriptions are used at all stages of the cycle – creation of I/O designs, automatic code generation etc. Subsequent tasks continue to add to and build upon this repository so that by the time the project is completed it contains a complete description of the entire system.

A set of CASE tools is generally designed to be compatible with each other, so that information generated from one tool can be passed to other tools. Thus, the subsequent tool in turn can use the information to complete its task and then pass the new information back to the system to be used by other tools. This allows for important information to be passed very efficiently and effectively between many planning tools with practically no resistance. Thus, CASE tools ensure that information are correctly passed

between stages and not lost in transit during the development process of a system. When CASE tools in system development are not used, incorrect information can very easily be passed between designers or can simply be lost in the shuffle of papers.

Hence, CASE tools help the software development process in the following manner:

- Ensures consistency, completeness and conformance to standards
- Encourages an interactive workstation environment
- Speeds up the development process
- Allows precision to be replicated
- Reduces costs, especially in maintenance
- Increases productivity

There are large numbers of CASE tools available from different vendors. Catelyze is a popular CASE tool used for requirements management, project management and software process. Cost Xpert is used for cost estimation, software planning and scheduling. Cradle is another CASE tool that supports all aspects of software development project, e.g. system analysis, system design, business modeling, automated documentation and code generation. Software through Pictures is another CASE tool that can handle all aspects of system design and development.

11.5.4 Pitfalls of CASE Tools

CASE tools also have some limitations and pitfalls. These are as follows:

Indiscreet Implementation: Implementing CASE technologies can involve a significant change from traditional development environments. Typically, organizations should not use CASE tools the first time on critical projects or projects with short deadlines because of the lengthy training process. Additionally, organizations should consider using the tools on smaller, less complex projects and gradually implementing the tools to allow more training time.

Inadequate Standardization: Integration of CASE tools from different vendors may be difficult if the products do not use standardized code structures and data classifications. File formats can be converted, but it is not economical. Controls include using tools from the same vendor or using tools based on standard protocols and insisting on demonstrated compatibility. Additionally, if organizations obtain tools for only a portion of the development process, they should consider acquiring them from a vendor that has a full line of products to ensure future compatibility if they add more tools.

Weak Repository Controls: Failure to adequately control access to CASE repositories may result in security breaches or damage to the work documents, system designs or code modules stored in the repository. Controls include protecting the repositories with appropriate access, version and backup controls.

Unrealistic Expectations: Organizations often implement CASE technologies to reduce development costs. Implementing CASE strategies usually involves high start-up costs. Generally, the management must be willing to accept a long-term payback period. Controls include requiring senior managers to define their purpose and strategies for implementing CASE technologies.

SUMMARY

Good coding and documentation are necessary for reliable and maintainable software. Adherence to a well-defined and common style of coding called 'Coding Standard' is necessary for good programming practices.

There are three types of Coding Standard: (1) general coding standard, (2) language-specific coding standards and (3) project-specific coding standards. Coding standard prescribes some guidelines relating to formatting features, naming convention for source file, variable, classes, subroutines, functions and methods. It also prescribes norms for good coding practices. These norms relate to desirable length of line and program statement; size of classes, subroutines, functions and methods; use of inline comments; error messages; error return conventions and exception handling mechanisms; compiler warnings; use of global data and variables.

Coding guidelines are a set of best practices. They prescribe certain dos and donts that make the program codes easy to read and maintain. Coding guidelines are not mandatory.

A document is used for explaining some attributes of an object, system or procedure. There are different types of software documentation such as (1) Requirements documentation, (2) Design documentation, (3) Program documentation, (4) User documentation and (5) Marketing documentation.

Requirements documentation is the description of what the software should do. Software design document is a written description of a software product. Program documentation gives a comprehensive procedural description of a program. It helps in maintenance and any future modification of the program. User documentation includes user manuals that enable users and operators to interact successfully with the system. It can be organized in three broad ways: (1) tutorial approach, (2) thematic approach and (3) list or reference. Marketing documentation consists of promotional materials to create interest for the software among potential buyers and users.

Documentation standards and guidelines relate to the structure of software user documentation, usage mode, content of user documentation and its formatting features.

CASE tools automate methods for designing, documenting and producing computer code in the desired programming language. CASE tools can be classified into (1) Upper CASE tools and (2) Lower CASE tools. Upper CASE tools deal with system investigation, analysis and design. Lower CASE tools are used for generating program codes.

Workbenches integrate several CASE tools into one application to support specific software-process activities. CASE environment is a collection of CASE tools and workbenches that supports the software process.

CASE tools improve the quality of software and the productivity of software development. They also have some limitations and risks due to indiscreet implementation, inadequate standardization, weak repository controls and unrealistic expectations.

EXERCISES

1. List the features of software that make it maintainable.
2. Write the importance of Coding Standard.
3. Distinguish between general coding standard, language-specific coding standards and project-specific coding standards.
4. List various formatting features relating to coding standard.
5. What should be the desirable contents of the headers preceding codes for different modules?
6. What do you mean by naming convention with reference to coding standard?
7. What you mean by coding norms? Write the various aspects that are considered for good coding practices.
8. Distinguish between coding standard and coding guidelines.

9. List various types of software documentation.
10. Write the importance of documentation standard.
11. Distinguish between program documentation and user documentation.
12. Explain the importance of user documentation.
13. Explain three approaches that are used for user documentation.
14. Describe the typical structure of software user documentation.
15. What do you mean by usage mode of documentation?
16. List the information content of software user documentation.
17. Distinguish between Upper CASE tools and Lower CASE tools.
18. What do you mean by CASE Workbenches and CASE Environments?
19. What are CASE environments? What is the basis on which these are classified?
20. List the advantages and pitfalls of CASE Tools.

SOFTWARE TESTING

This chapter describes the basics of software testing. It covers the following topics:

- *Verification and Validation*
- *Black Box Testing*
- *White Box Testing*
- *Testing Phases, e.g. Unit Testing, Integration Testing and System Testing*
- *Object-oriented Testing*

The various testing methodologies under the different categories with examples are explained in this chapter.

Software testing is a process of evaluating a software system by manual or automatic means and verifying it against specified requirements. It determines the gap between expected and actual results. In simple terms, testing is a process of evaluating particular software to determine whether it contains any defects. The main principle of testing is to find defects by subjecting the software to different odd conditions, and then correct the same. Software testing fulfils the following objectives:

- Finds errors and defects before final implementation
- Checks whether the software meets its specifications (Verification)
- Checks whether the software meets its user needs (Validation)
- Establishes confidence in the software
- Reduces uncertainty about the quality of software
- Delivers reliable software

12.1 TESTING FUNDAMENTALS

Software testing is a process in which a set of test inputs (or test cases) are given to the software or its components to examine whether these are behaving as expected. If the software fails to behave as expected, then the conditions under which it failed are noted so that debugging and correction can be done at a later stage. Some of the commonly used terms associated with testing are as follows:

Failure: This is a manifestation of an error (or defect or bug). However, the mere presence of an error may not necessarily lead to failure.

Test case: This is specified by three parameters:

Data input to the system, denoted by 'I'

State of the system when the data input should be given, denoted by 'S'

Expected output of the system, denoted by 'O'

Thus, test case is represented by a triplet, [I, S, O]

Test suite: This is the set of all test cases with which a given software product is to be tested.

12.1.1 Verification and Validation

The purpose of testing is to do verification and validation of software. Verification and Validation are the two important processes of software testing to ensure that the software is fit for the purpose.

- **Verification:** The software should conform to its specification. Verification is concerned with whether the software product is being built right.
- **Validation:** The software should do what the user really requires. Validation is concerned with whether the right software product is being built.

Software verification and validation is defined as a software engineering process in which the quality is built into the software during its development. Software verification is the process of determining whether the deliverables (work products) obtained during any stage of software development fulfil the requirements or conditions as imposed on these. The verification process can be of two types:

- i. Static verification
- ii. Dynamic verification

Software inspection and walkthroughs are done to discover problems through the analysis of static representation of the system. These are a part of static verification. On the other hand, exercising and observing product behavior is a part of the dynamic verification process. Software inspections and testing are complementary and both these processes are used for verification and validation.

Inspection can check conformance with specification but it cannot check conformance with customers' requirements. It cannot check non-functional characteristics such as performance, usability etc. Therefore, both verification and validation are needed for quality, as shown in Figure 12.1.

12.1.2 Testing Process

In a standard waterfall model, each phase must be completed before the next phase begins. However, when testing is incorporated at every stage, the process steps of the waterfall model instead of moving down in a linear way are bent upwards after the coding phase to form a 'V' shape. Thus, the V-model can be considered as a modified waterfall model as shown in Figure 12.2. Testing is emphasized more in the V-model than in the waterfall model. Testing is done at various levels. Accordingly, there are various types of testing, such as unit testing, integration testing, system testing and acceptance testing.

In the V-model, testing begins at the component or unit level and works towards integration of the entire system. Testing at the component level is referred to as '*testing in the small*'. After individual components have been tested, these are integrated and tested again as an integrated unit. This is called integration testing. After all components are integrated together, it becomes a system. The testing of the completed system is called system testing. Integration testing and system testing are known as '*testing in the large*'.

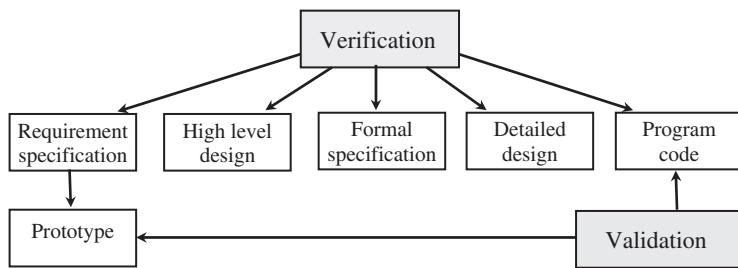


Figure 12.1 Verification and validation for V-Model

It may be noted that different testing techniques are appropriate at different points of time. Each type of testing is done by different test groups. The test groups are constituted by taking people who are not involved in the software development, so that they can identify defects (bugs) made by developers.

Unit testing can be done by the developer but integration testing is usually done by an independent testing team. Some of the important points of the testing strategy are given below.

- Specify requirements of the software in a quantifiable manner before starting the tests.
- Specify testing objectives explicitly.
- Develop a test plan that emphasizes rapid cycle testing.
- Build robust software that is designed to test itself, e.g. use anti-bugging.
- Use effective formal reviews as a filter prior to testing.
- Conduct formal technical reviews to assess test strategy and test cases.

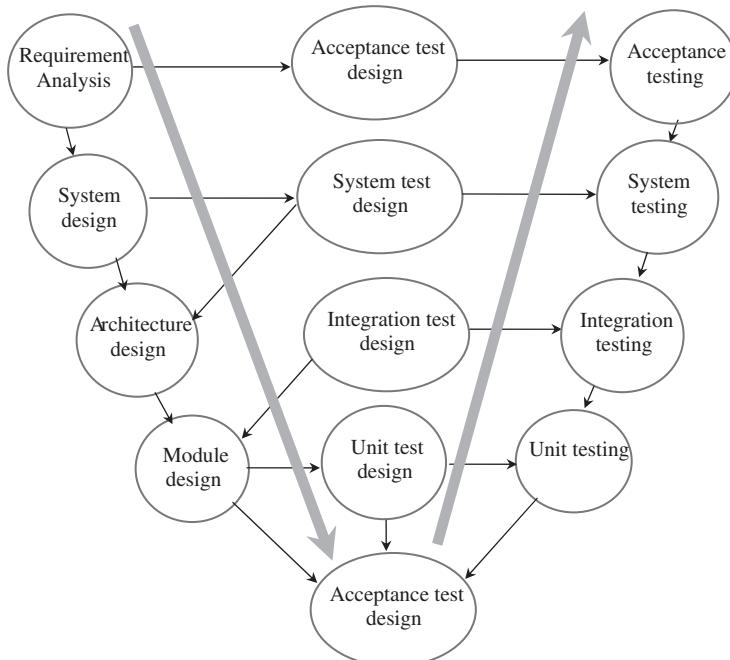


Figure 12.2 V-Model for testing

Verification and Validation (V and V) is concerned with only identifying the defects in a program. It is done by testing.

Testing is done in a planned way by constructing test cases. A test case is a triplet [I, S, O], where I is data input to the system, S is the state of the system at which the data is entered and O is the expected output of the system. The output of the program for each test data is compared with expected output O to know whether there is any defect or not. If the output of the program matches with the expected output, then there is no defect. Test cases are planned with a view to test all possible aspects of the software product. The testing process is shown in Figure 12.3.

However, debugging is concerned with locating and repairing these errors. Debugging is a process that takes the defects from the test results, repairs the error with respect to specification and then retests the program. The debugging process is shown in Figure 12.4.

Debugging involves formulating hypotheses about software behavior and then testing these hypotheses to find the error. The defect testing finds errors, whereas debugging finds the causes of error for making corrections.

12.2 BLACK BOX TESTING

In black box testing, internal logic of the software is not looked into while testing. The software or software components to be tested are considered as a 'black box'. The testing is done by entering input data and checking whether the output obtained is per requirement or not. Here, test cases are designed from user requirements. The following methodologies for black box testing are discussed in this section:

- Decision Table-based Testing
- Cause–Effect Graphs in Functional Testing
- Equivalence Partitioning
- Boundary Value Analysis

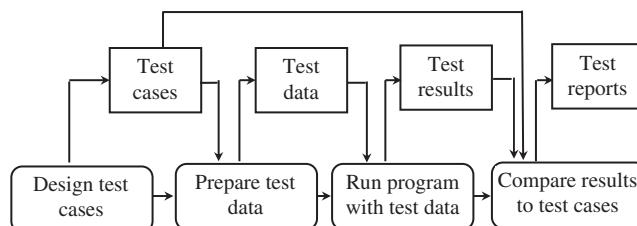


Figure 12.3 Testing process to find defects

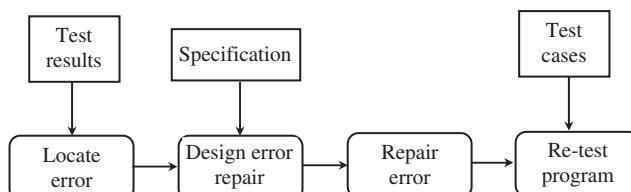


Figure 12.4 Debugging process

12.2.1 Decision Table-based Testing

Decision tables can be used to test how different combinations of inputs may generate different outputs. Various decision rules used in software requirements can be easily depicted through decision tables. The tabular form of decision tables is easy to understand and thus it is a useful method for black box testing. For example, consider the functionality ‘c1 and c2 then a1’. The first rule is ‘both c1 and c2 needs to be true for a1 to be true’. This rule is shown in the first column of decision Table 12.1. The decision table depicts decision rules in columns. Each rule can be treated as a test case. Digit 1 indicates true, 0 indicates false and ‘x’ indicates ‘no effect’, i.e. may be true, may be false.

Table 12.1 Decision table

| | r1 | r2 | --- | --- | Rn |
|----------------|----|----|-----|-----|----|
| c1 | 1 | 1 | | | 0 |
| c2 | 1 | 0 | | | 0 |
| ---- | x | x | | | 1 |
| c _n | 0 | 0 | | | 0 |
| a1 | 1 | 0 | | | 0 |
| a2 | 0 | 1 | | | 1 |
| ---- | 0 | 0 | | | 0 |
| a _m | 0 | 0 | | | 1 |

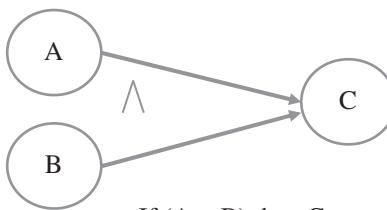
12.2.2 Cause–Effect Graphs in Black Box Testing

Causes–effects are represented as nodes of a cause–effect graph. The cause–effect graph captures the relationship between specific combinations of inputs, i.e. causes and outputs, i.e. effects. The graph also includes a number of intermediate nodes linking causes and effects. Some examples of the primitive cause–effect graphs are shown in Figures 12.5 and 12.6.



If A then B

Figure 12.5 Cause-effect graph of identity



If (A \wedge B) then C

Figure 12.6 Cause-effect graph for If (A and B) then C

The various steps in creating the cause–effect graph are as follows:

1. Study the functional requirements and divide the requirements into workable pieces.
2. Identify causes and effects in the specifications.
 - Causes: distinct input conditions
 - Effects: an output condition or a system transformation
3. Assign a unique number to each cause and effect.
4. Use the semantic content of the specification and transform it into a Boolean graph.
5. Annotate the graph with constraints describing combinations of causes and/or effects using Table 12.2.
6. Convert the graph into a limited-entry decision table.
7. Use each column as a test case.

Table 12.2 Processing nodes used in debugging

| Type of processing node | Description |
|-------------------------|--|
| AND | Effect occurs if all inputs are true |
| OR | Effect occurs if both or one input is true |
| XOR | Effect occurs if one of the inputs is true |
| Negation | Effect occurs if inputs are false |

For example, a simple ATM banking transaction system can have the following causes and effects:
Causes (inputs)

- C1: Command is withdrawal
- C2: Command is deposit
- C3: Account number is valid
- C4: Transaction amount is valid

Effects (outputs)

- E1: Print ‘invalid command’
- E2: Print ‘invalid account number’
- E3: Print ‘withdrawal amount not valid’
- E4: Print ‘successful withdrawal made’
- E5: Print ‘successful deposit made’

Then the cause–effect graph can be drawn as shown in Figure 12.7. If C1 or C2 is true, then one can have a valid command and thus, the negation of C1 or C2 will become an invalid command, i.e. E1. Similarly, if C1 is true, i.e. command is withdrawal (can choose C2 instead of C1), but C3 is false, i.e. account number is invalid, then the effect is ‘print invalid account number’, i.e. E2. If withdrawal command is correct, i.e. C1 is true, account number is valid, i.e. C3 is true, but the transaction amount is invalid, i.e. C4 is false, and then the effect is ‘print withdrawal amount not valid’, i.e. E3. Finally, for the effect E4, the causes can be C1, C3 and C4 and for the effect E5, the causes can be C2, C3 and C4. This is shown in the cause–effect graph. However, in some ATM applications where deposit is not permitted, the effect E5 and the cause C4 can be omitted.

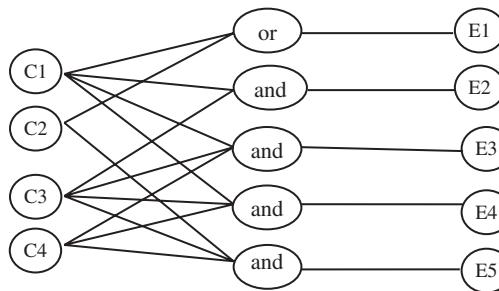


Figure 12.7 Cause-effect graph for an ATM

After, annotating the graph describing combinations of causes and/or effects, the graph is converted into a limited-entry decision table and each of its columns is treated as a test case. The decision table for the ATM example against the cause–effect graph is shown in Table 12.3. Each rule can be reflected as one column in the decision table. 1 indicates true, 0 indicates false and x indicates don't care, i.e. may be true, may be false. All the rules that are used to draw the cause–effect graph can be put in this decision table. For example, E2 effect requires C1 as true and C3 as false. Thus, in the decision table, the second column against r2 contains 1 in C1, 0 in C3 and 1 against E2. Also, x is indicated in both C2 and C4. Similarly, all other columns can be created.

Table 12.3 Decision table

| | r1 | r2 | r3 | r4 | r5 |
|----|----|----|----|----|----|
| C1 | 0 | 1 | 1 | 1 | x |
| C2 | 0 | x | x | x | 1 |
| C3 | x | 0 | 1 | 1 | 1 |
| C4 | x | x | 0 | 1 | 1 |
| E1 | 1 | 0 | 0 | 0 | 0 |
| E2 | 0 | 1 | 0 | 0 | 0 |
| E3 | 0 | 0 | 1 | 0 | 0 |
| E4 | 0 | 0 | 0 | 1 | 0 |
| E5 | 0 | 0 | 0 | 0 | 1 |

12.2.3 Equivalence Partitioning and Boundary Value Analysis

Equivalence partitioning method of black box testing takes advantage of symmetries, data equivalencies and independencies to reduce the number of necessary test cases. This method can be clubbed together with boundary value analysis. The process works as follows:

- Determine the ranges of the working system
- Develop equivalence classes of test cases
- Examine the boundaries of these classes carefully

The input data and output results often fall into classes where all members of the class are related. In such a case, an *equivalent partitioning* of the input data can be made. In this approach, the domain of input values to the software is partitioned into a set of equivalence classes. Partitioning is done such that the behavior of the software is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that class. Equivalence classes can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If an input condition specifies range, then one valid and two invalid equivalence classes are needed.
2. If a condition requires a specific value, then one valid and two invalid equivalence classes are needed.
3. If an input condition specifies a member of a set, then one valid and one invalid equivalence class are needed.
4. If an input condition is Boolean (i.e. 0 or 1), then one valid and one invalid class are needed.

For example, a user can type a four digit password for the ATM, followed by a set of typed commands, e.g. withdraw, deposit, bill pay, transfer etc. Some of the derived equivalence classes are given below. Test cases should be chosen from each partition or class.

1. Password is not numeric (invalid)
2. Password is numeric (valid)
3. Password is less than four digits in length (invalid)
4. Password is four digits in length (valid)
5. Password is more than four digits in length (invalid)

Black box tests can be done for the above partitions or classes. Test cases can cover all the partitions as shown in Table 12.4.

Table 12.4 Test cases for password validation in an ATM

| Test case # | Test Data | Expected Outcome | Classes Covered |
|-------------|-----------|------------------|-----------------|
| 1 | 6170 | T | 2, 4 |
| 2 | 23,573 | F | 1, 5 |
| 3 | 324 | F | 3 |

After the inputs and outputs of software are partitioned into classes according to the software specification, *boundary value analysis* is done. It complements equivalence partitioning and test cases are generated to exercise these boundaries. It focuses on the boundaries of the input as per the following:

- If input condition specifies a range bounded by certain values, say, a and b, then test cases should include:
 - Values for a and b
 - Values just above and just below a and b
- If an input condition specifies any number of values, test cases should include:
 - The minimum and maximum numbers
 - The values just above and just below the minimum and maximum values

For example, while entering marks of a subject having full mark 50, if a mark is entered outside its expected range, then a fault message is generated. If a student has scored at least 20, he has passed the subject. Also, all inputs are passed as integers. Accordingly, the following six equivalence partitions for the inputs may be identified:

1. $0 \leq \text{Mark} \leq 50$ (valid)
2. $\text{Mark} > 50$ (invalid)
3. $\text{Mark} < 0$ (invalid)
4. $\text{Mark} = \text{Integer}$ (valid)
5. $\text{Mark} = \text{Alphabetic}$ (invalid)
6. $\text{Mark} = \text{Real number}$ (invalid)

Then, the partitions for the outputs are identified. Valid partitions are produced by considering each of the valid outputs of the software.

1. ‘Pass’ is induced by $20 \leq \text{Mark} \leq 50$
2. ‘Fail’ is induced by $0 \leq \text{Mark} < 20$
3. ‘Fault Message’ is induced by $\text{Mark} > 50$
4. ‘Fault Message’ is induced by $\text{Mark} < 0$

Having identified all the partitions, the test cases are derived. Test cases should cover all the identified partitions and their number should be kept low. Test cases corresponding to partitions derived from the input mark are shown in Table 12.5.

Table 12.5 Test Cases for marks entry

| Test Case # | Test Data | Expected Output | Partitions Covered |
|-------------|-----------|-----------------|------------------------------------|
| 1 | 0 | Fail | $0 \leq \text{exam mark} < 20$ |
| 2 | -1 | Fault Message | $\text{exam mark} < 0$ |
| 3 | 1 | Fail | $0 \leq \text{exam mark} < 20$ |
| 4 | 19 | Fail | $0 \leq \text{exam mark} < 20$ |
| 5 | 20 | Pass | $20 \leq \text{exam mark} \leq 50$ |
| 6 | 21 | Pass | $20 \leq \text{exam mark} \leq 50$ |
| 7 | 49 | Pass | $20 \leq \text{exam mark} \leq 50$ |
| 8 | 50 | Pass | $20 \leq \text{exam mark} \leq 50$ |
| 9 | 51 | Fault message | $\text{Exam mark} > 50$ |

In boundary value analysis, all the boundary values are tested as shown in Table 12.5.

12.3 WHITE BOX TESTING

The software testing approach that uses inner structural and logical properties of the program for verification is known as White Box Testing. It is also called as Clear Box Testing, Glass Box Testing and Structural Testing. It provides usage of more information on the tested object than Black Box Testing. It is the inference of real equivalence partitioning and has good syntactic coverage. However, it is expensive and has limited semantic coverage. The following testing methods are discussed in this section:

- Statement Coverage: Every statement is executed.
- Branch/Edge Coverage: Every branch option is chosen.
- Path Coverage: Every path is executed.
- Data Flow Testing: Program variable and path are tested.
- Mutation Testing: Many versions of the software are tested.

12.3.1 Statement Coverage

The statement coverage criterion is based on the observation that an error cannot be discovered if some parts of the program containing the error and generating the failure are not executed. Therefore, testing for complete coverage of statements is made. In a conventional block-structured language, assignment statements, input/output statements and procedure calls are elementary statements. Thus, these elementary statements can be part of more complex statements. With this assumption, the criterion for this statement coverage test is as follows:

Select a test set T such that, by executing program P for each input data d in T , each elementary statement of P is executed at least once.

In general, the same input data cause the execution of many statements. For example, consider the code given in Program 12.1.

In this program:

Let I_1 and I_2 be two ‘if’ statements

and let W_1, W_2, W_3 and W_4 represent the four ‘write’ statements

Let $d \in D_i$ be a class of input values

Then, $D_1 = \{a > 0\}$, $D_2 = \{a \leq 0\}$, $D_3 = \{b > 0\}$ and $D_4 = \{b \leq 0\}$ are the classes of input

Thus, a set of test cases from D can be chosen as shown below:

$\{(a = 5, b = 2), (a = -9, b = 31), (a = 31, b = -7), (a = -4, b = -3)\}$

```

read (a);
read (b);
if a > 0 then
    write ("1");
else
    write ("2");
end if;
if b > 0 then
    write ("3");
else
    write ("4");
end if;

```

Program 12.1: Program for Statement Coverage Testing

This test case set is not minimal with respect to the criterion, because every input data belongs to two classes. Thus, one can reduce the number of test cases to two, by selecting the following representatives, one being greater than zero and other less than or equal to zero.

{ ($a = -9, b = 7$), ($a = 2, b = -3$) }

This executes every elementary statement W_i at least once.

12.3.2 Branch Coverage

In branch coverage, test cases are designed to reach every branch in the code and to exercise every condition. Therefore, the three primitive components of coding, e.g. sequence, condition (if-then-else or switch case) and iteration (for or do-while) need to be taken into account. The sequence flow of control is shown in Figure 12.8 and the condition flows of control are shown in Figures 12.9 and 12.10.

For branch coverage, test cases are selected in such a way that every branch or edge is traversed at least once during the test. The criteria for condition coverage can also be applied to this test. The criterion for condition coverage test with branch coverage is as follows:

Select a test set T such that, by executing program P for each input data d in T , each edge of P 's control flow graph is traversed and all possible values of the constituents of compound condition are exercised at least once.

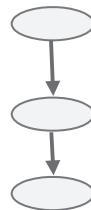


Figure 12.8 Sequence flow of control

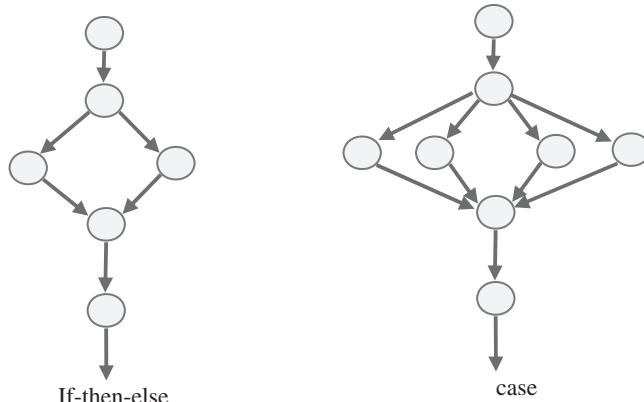
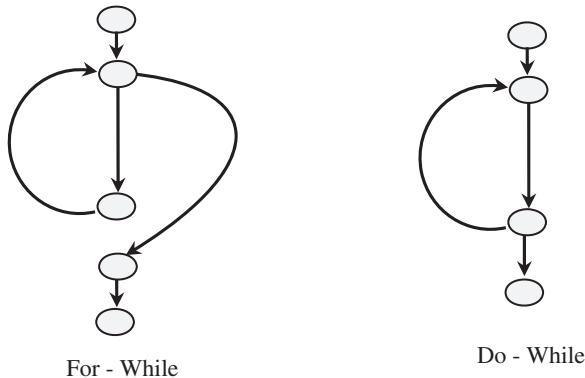


Figure 12.9 Condition flow of control

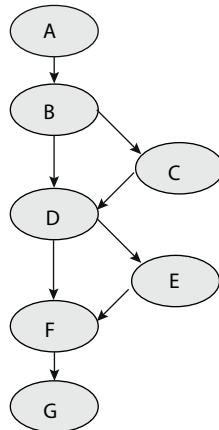
**Figure 12.10** Iteration flow of control

The complete coverage principle is applied to each of the branches of the program.

Let us consider the code shown in program 12.2 and the corresponding flow graph in figure 12.11. Accordingly, a set of test cases covering all branches and all conditions are selected below. However, it fails when $a=0$ as it encounter division by zero.

{ (a = 2, b = 0, c = 4), (a = 1, b = 1, c = 1) }

```
read (a, b, c);
if a > 1 and b = 0 then
    c = c/a;
end if; if a = 2 or c > 1 then
    c = c+1;
end if;
```

Program 12.2: Program for Branch Coverage Testing**Figure 12.11** Flow Graph of program 12.2

The branch coverage should cover every branch.

12.3.3 Path Coverage

Traversing all the branches individually as shown in the above example, i.e. program 12.2, may not be always sufficient to test the software. Therefore, another coverage known as path coverage should be followed. If there are two statements as shown in program 12.2, then there have to be four linearly independent paths that need to be checked. Thus, there can be four test cases as shown below:

```
{( a = 2, b = 0, c = 4 ), ( a = 1, b = 1, c = 1 )
( a = 3, b = 0, c = 1 ), ( a = 2, b = 1, c = 1 )}
```

The test cases will cover all four independent paths ABCDEFG, ABDFG, ABCDFG and ABDEFG of Figure 12.11, respectively. The criterion for the condition coverage test with path coverage is as follows:

'Select a test set T such that, by executing program P for each input data d in T, all paths leading from the initial to the final node of P's control flow graph (CFG) are traversed.'

12.3.4 McCabe's Cyclomatic Complexity

For more complicated software programs, it is not easy to determine the number of independent paths. McCabe's Cyclomatic complexity defines an upper bound for the number of linearly independent paths traversed through the software program and thus it is termed as a metric. The metric provides a practical way of determining the maximum number of linearly independent paths in a software program. Though the metric does not directly identify the linearly independent paths, it informs approximately how many paths to look for. There are three different ways to compute this metric. Computation by any of the three methods gives approximately the same answer.

Method 1: Given a CFG G of a program, the Cyclomatic Complexity $V(G)$ can be computed as:

$$V(G) = E - N + 2$$

where N is the number of nodes of the CFG and E is the number of edges in the CFG.

For the CFG of example shown in Figure 12.11, $E=8$ and $N=7$. Therefore, the cyclomatic complexity $= 8 - 7 + 2 = 3$.

Method 2: An alternative way of computing the cyclomatic complexity of a program from an inspection of its CFG is as follows:

$$V(G) = \text{Total number of bounded areas} + 1$$

In the program's CFG, any region enclosed by nodes and edges can be called as a bounded area. The number of bounded area increases with the number of decision paths and loops. This metric provides a quantitative measure of testing difficulty and the ultimate reliability. For example, the number of bounded areas of the CFG shown in Figure 12.13 is 2. Therefore, its cyclomatic complexity is $2+1 = 3$. Thus, examination of CFG provides an easy way of computing cyclomatic complexity.

Method 3: The cyclomatic complexity of a program can also be computed by computing the number of decision statements of the program. If N is the number of decision statements of a program, then the McCabe's metric is equal to $N+1$.

$$V(G) = \text{Total number of decision points} + 1$$

For the CFG example shown in Figure 12.14, the number of decision statements is 2. Therefore, the cyclomatic complexity is $2 + 1 = 3$.

12.3.5 Data Flow-based Testing

Data flow-based testing method examines data rather than control. It selects test paths of a software program according to the locations of the definitions and uses of different variables in a software program. Data flow testing can be performed at two conceptual levels.

1. Static Data Flow Testing
2. Dynamic Data Flow Testing

Static Data Flow Testing: In this testing, the source code is analyzed to find potential defects, i.e. finding *data flow anomalies*. There are three types of abnormal situations while using variables.

- Type 1: Defined and then defined again, e.g. $a = f1(b)$ and $a = f2(c)$.
- Type 2: Undefined but referenced, e.g. $a = a + b - c$, where c is not defined.
- Type 3: Defined but not referenced, e.g. $a = f(b, c)$, but is not used.

These are dangerous signs, but may not lead towards defects. However, the dynamic data flow testing criteria is more important and can be done only by executing the code. Here, the data can be among one of three states.

- i. *Defined*: Initialized but not used
- ii. *Used*: In computation or as argument
- iii. *Killed*: Undefined in some way

Dynamic Data Flow Testing: This testing involves actual program execution by identifying data flow paths. For this, dynamic data flow analysis is done based on data flow graph of the program code.

The paths are identified based on ***data flow testing criteria***. The occurrence of each variable in a code can be classified either as:

- (1) Variable Definition or (2) Variable Use

Variable definition: A variable needs to be defined for use in a program. It is denoted by short form 'def'. Few examples of def of variable are shown below. The def of 'a' is shown in bold italics.

e.g.

$\mathbf{a} = b * c; \text{read}(\mathbf{a}); \text{int } \mathbf{a};$

Variable use: Variable use represents the use of a variable in a statement. Few examples of use of variables are shown below. The use of "a" is shown in bold italics.

e.g.

$b = (\mathbf{a} + b) * c; \quad \mathbf{a} = \mathbf{a} + 1; \quad \text{write}(\mathbf{a}, b); \quad c = \mathbf{a}[i];$

Variable use can be of two types:

1. Computational use (denoted as c-use)
2. Predicate use (denoted as p-use)

Uses of a variable in input and assignments are classified as *c-use*. Use of variable in conditions is classified as *p-use*.

Examples of c-use and p-use are given below.

$< a = 2 * b >$ is an example of c-use as variable 'b' has been used to compute value of variable 'a'.

$< \text{if } (b > 50) \{ \dots \} >$ is an example of p-use as variable 'b' has been used in a condition.

Both c-use and p-use affect the flow of control. P-uses directly affect the flow of control as their values are used in evaluating conditions, whereas c-uses affect indirectly as their values are used to compute other variables, which in turn affect the outcome of condition evaluation.

The following points may be noted with respect variable definition and use.

- A path from node-*i* to node-*j* is said to be *def-clear* with respect to a variable 'x' if there is no '*def of x*' in the nodes along the path from node-*i* to node-*j*. Nodes *i* and *j* may have a *def* of *x*.
- '*def-clear*' path from node-*i* to edge (*j,k*) is one in which no node on the path has a *def* of *x*.

- ‘def’ of a variable ‘x’ is considered global to its block if it is the last ‘def’ of ‘x’ within that block.
- ‘c-use’ of ‘x’ in a block is considered global c-use if there is no ‘def of x’ preceding this c-use within this block.
- $def(i)$: Set of all variables for which there is a global def in node- i .
- $c\text{-use}(i)$: Set of all variables that have a global c-use in node- i .
- $p\text{-use}(i, j)$: Set of all variables for which there is a p-use for the edge (i, j) .
- $dcu(x, i)$: Set of all nodes such that each node has ‘x’ in its c-use and ‘x’ is in $def(i)$.
- $dpu(x, i)$: set of all edges such that each edge has ‘x’ in its p-use and ‘x’ is in $def(i)$.
- $DU\ pair$: A pair of definition and use for some variable, such that at least one DU path exists from the definition to the use.
- $DU\ path$: A definition-clear path on the CFG starting from a definition to the use of same variable.

Let us consider a sample program code to find greater among two variables ‘a’ and ‘b’. The program is given at program 12.3.

```

read (a, b);
if a > b then
    c = b;
else
    c = a;
end if;
write (c);

```

Program 12.3: Sample program to find greater between a and b

The def-use graph of program 12.3 is shown in figure 12.12. This graph can be constructed by associating defs, c-use, and p-use sets with nodes of a flow graph.

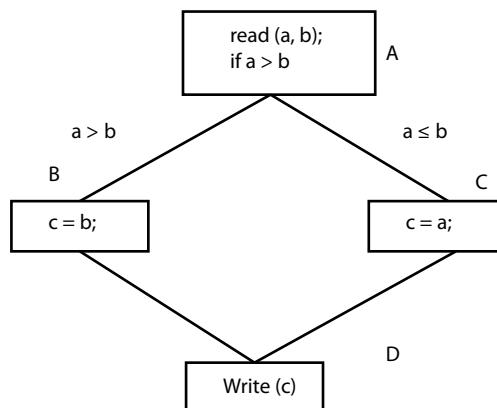


Figure 12.12: Flow graph of program 12.3

As the node A has read(a, b), it contains def of both a & b i.e. {a, b}, but the c-use is nil (i.e. \emptyset) as there is neither use of "a" nor "b". However, as the node contains "if a > b", the p-use(A,B) and p-use(A,C) will contain {a, b}, which can be put against the p-use of the node. Node B has {c} as def, {b} as c-use & \emptyset as p-use. Similarly, node C has {c} as def, {a} as c-use & \emptyset as p-use. Finally, node D has \emptyset as def & c-use, only {c} is in p-use. Then, the seven data flow testing criteria e.g. all-defs, all-c-uses, all-p-uses, all-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, all-du-paths are tested. The relative strengths of testing strategies are shown in Figure 12.13. The strength of the testing strategy reduces along the direction of the arrow as shown in the figure. Though all paths testing strategy is the strongest testing strategy, but it is expensive and time consuming. Thus, all-uses should be covered by selecting test cases covering all DU pairs. Accordingly test cases should be prepared to cover all-use criterion of the data flow based testing.

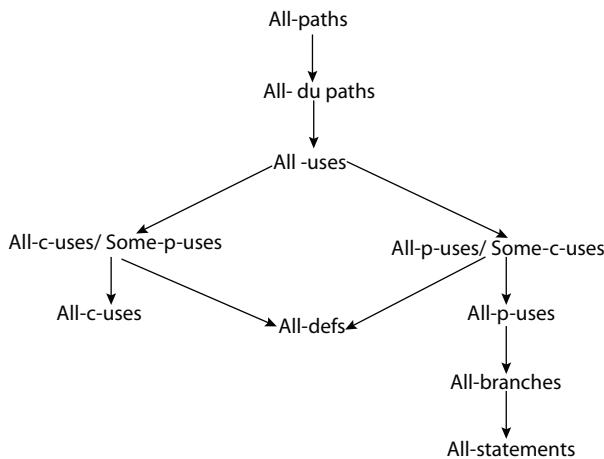


Figure 12.13 Relative strength of testing strategies

12.3.6 Mutation Testing

Mutation testing is done by introducing a single small change to the code of the software. It is a fault-based testing program. Simple faults are introduced by the mutation operator. Each change by the mutation operator is encoded in a mutant program. The modified program is called a mutant. A mutant is said to be killed by a test case, when the execution of the test case causes it to fail. Then the mutant is considered to be dead.

A mutant is an *equivalent* to the given program if it always produces the same output as the original program. A mutant is called killable or stubborn, if the existing set of test cases is insufficient to kill it. If a mutant remains alive even after all the test cases are exhausted, the test data are enhanced to kill the mutant. The process of generation and killing of mutants can be automated by predefining a set of basic changes that can be applied to the program. These changes can be alterations such as changing an operator, changing the value of a constant, changing a data type etc.

A mutation score for a set of test cases is the percentage of non-equivalent mutants killed by the test suite. The test suite is said to be mutation-adequate if its mutation score is 100%.

$$\text{Mutation Score} = 100 * D/(N - E)$$

Where:

D = Dead mutants

N = Number of mutants

E = Number of equivalent mutants

Since mutation testing generates a large number of mutants and it is required to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction with some testing tool that would run all the test cases automatically. A mutation testing tool titled 'Mothra' was developed in the mid 1980s at the Georgia Institute of Technology. It was written in C and works on FORTRAN 77 programs. Similarly, Korea Advanced Institute of Science and Technology (KAIST), South Korea and George Mason University, USA collaborated and jointly developed μ Java (μ Java), the mutation system for Java programs. It automatically generates mutants for both traditional mutation testing and class-level mutation testing.

12.4 UNIT TESTING

Software units are the independent executable components of an application. In unit testing, the units/modules of a system are tested in isolation. It is usually done by programmers who write the code.

Test cases for unit testing are selected based on code, specification, experience etc. Multiple test cases can be created per unit, e.g. one per equivalence class. A collection of closely related unit tests is called a test suite. It contains the following:

- Unit tests for all members of a class
- Unit tests for all functions in a module
- Unit tests for all functions related to specific activity

Test suites may be organized into a hierarchy. The complete test suite may contain some top-level test suites. Beneath the top-level test suites there may be some smaller test suites for subsystems. Test suites for modules can be arranged at the lowest level. The test hierarchy of an object-oriented (OO) system will contain the following:

- Unit tests of methods
- Suites of unit tests that cover all the classes
- Suites of class test suites that cover subsystems, e.g. libraries and components
- Suites of component tests that cover the entire system

The various unit testing tools/frameworks can be used to test codes in specific languages.

- For C++, CPPUnit and CxxUnit
- For Java, Junit
- For .NET Languages, e.g. C#, VB.NET, etc., NUnit

12.5 INTEGRATION TESTING

In unit testing, the individual modules are tested in isolation. However, it is also necessary to check whether the modules when integrated together as a system are able to function as required or not. The software application consists of different modules. The modules interact with each other by passing parameters or one module may invoke another module. The key idea of integration testing is to test the module interfaces through which the interactions take place. Testing is done to check that there are no errors in the parameter passing.

In OO testing as objects are defined by their interfaces. The various interface types that need to be tested are as follows:

1. Parameter interfaces: Data passed from one procedure to another
2. Shared memory interfaces: Block of memory is shared between procedures
3. Procedural interfaces: Subsystem encapsulates a set of procedures to be called by other subsystems
4. Message passing interfaces: Subsystems request services from other subsystems

During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. Thus, the integration test plan is driven by the build plan of the system architecture.

12.5.1 Big Bang Integration Testing

It is one of the common integration testing approaches. In this approach, all the modules making up a system are integrated in a single step, i.e. all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found, it is difficult to localize the error. This is because the error may potentially belong to any of the modules being integrated. Thus, debugging of errors found by big bang integration testing is costly.

12.5.2 Incremental Integration Testing

In this testing, the tests are done in incremental steps. After each integration step, the partially integrated system is tested. First Modules A and B may be integrated and tested by test sequence-1. The next test, one more Module C may be added and tested by creating test sequence-2. Similarly, more modules may be integrated in steps and the test sequence-3 can be generated by including some more test cases for testing the entire system. Incremental integration testing is shown in Figure 12.14.

12.5.3 Bottom-up Integration Testing

In bottom-up testing, each subsystem is tested separately and then the subsystems are integrated and tested as a whole. A subsystem might consist of many modules that communicate among themselves

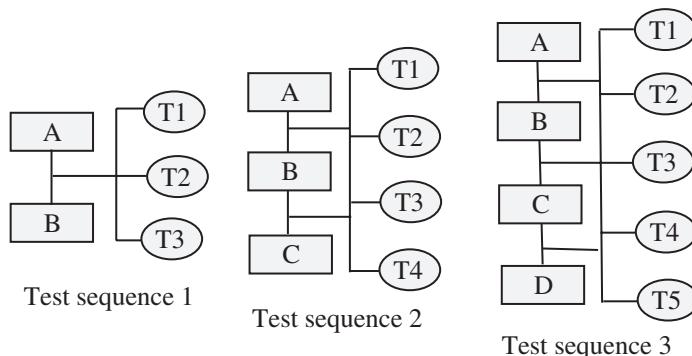


Figure 12.14 A sample incremental testing

through well-defined interfaces. The test cases are devised to use the interfaces in all possible manners. Large software systems normally require testing of the subsystem to be done at several levels. After testing at one level is completed successfully, lower-level subsystems are combined to form higher-level subsystems. For example, consider the software, where the main Module A is decomposed into Modules B, C and D. Module C is further decomposed into Modules E, F and G as shown in Figure 12.15. Bottom-up testing is done by starting from the lowest-level modules and then proceeding to higher-level modules.

Here, a test driver is first designed to integrate the lowest-level Modules E, F and G, as shown in Figure 12.16.

The return values generated by one module are used in other modules. In some way the test driver mimics the expected behavior of Module C to integrate E, F and G. After testing the test driver is replaced by actual Module C as shown in Figure 12.17. Similarly, a new test driver is used for integrating more modules (e.g. Modules B, D etc.). The new test driver mimics the behavior of Module A and the test driver is replaced by Module A. Further tests are performed, if required.

The main advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing, no stubs are required, only test drivers are required. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems.

12.5.4 Top-down Integration Testing

The top-down integration testing starts with the main routine and one or more subordinate routines in the system. It requires the use of program stubs to simulate the effect of lower-level routines. In the example shown in Figure 12.15, Modules A and B are integrated using stubs C and D and then stub D is replaced by its actual instance D. The stub C is replaced by the actual Module C and new stubs E, F and G.

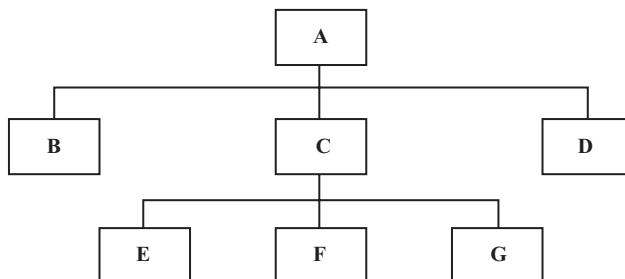


Figure 12.15 A sample incremental testing

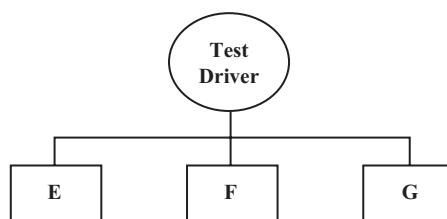


Figure 12.16 Bottom-up integration of Modules E, F and G

In pure top-down integration, no driver routines are required. The major advantage of this method is that there is a strong psychological boost when major modules are done. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines it may become difficult to subject the top-level routines to all desired conditions for testing.

12.5.5 Sandwiched Testing

A sandwiched integration testing follows a combination of top-down and bottom-up testing approaches. In the top-down approach, testing can start only after the top-level modules have been coded and unit tested. However, in sandwiched testing approach, a system is integrated using a mix of top-down, bottom-up and big bang approaches. Here, a hierarchical system is viewed as consisting of three layers. The bottom-up approach is applied to integrate the modules in the bottom layer. The top layer modules are integrated by using the top-down approach. The middle layer is integrated by using the big bang approach after the top and the bottom layers have been integrated. In fact, in this mixed approach, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approach.

12.5.6 Backbone Integration Testing

This is almost same as a sandwiched integration testing. The only exception is that here the backbone or the kernel modules of the system are tested as big banged together using the bottom-up testing approach. Then, the system controls are tested using the top-down approach, until the backbone has been included. This testing is mainly used to verify interoperability among tightly coupled subsystems in embedded system applications.

12.5.7 Thread Integration Testing

A thread is a portion of several modules that together provide a user-visible program feature. In this type of testing the entire system is tested by integrating one thread with another. It is more useful for OO systems and can be used in OO integration testing. A thread integrates classes required to respond to one input or event. The thread integration testing starts with single event threads and then go on to multiple event threads.

12.6 OBJECT-ORIENTED TESTING

The class is the basic testing unit of software developed through OO methodology. This is especially appropriate if strong cohesion and loose coupling is applied effectively in the class design. The V model for testing, which is shown earlier in Figure 12.2, can be modified for OO testing model as shown in Figure 12.18.

In OO methodology testing is done at various levels as given below:

1. The *algorithm level* involves checking the routine (method) on some data.
2. The *class level* testing is concerned with the interaction between the attributes and methods for a particular class.
3. The *cluster level* testing considers interactions between groups of cooperating classes.
4. The *system level* involves testing an entire system in the aggregate.

Use cases, CRC cards and class diagrams are revisited in OO testing. Once the class hierarchy has been determined, the testing process for the classes can start.

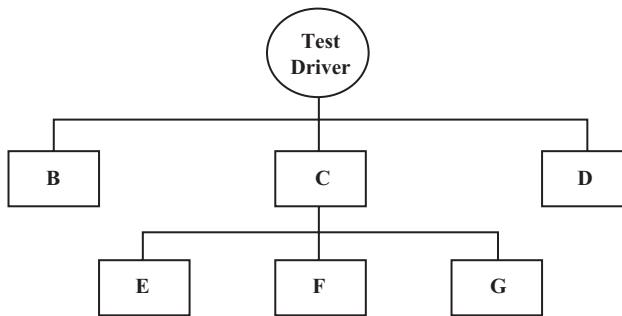


Figure 12.17 Bottom-up integration of Modules B, C and D with E, F and G

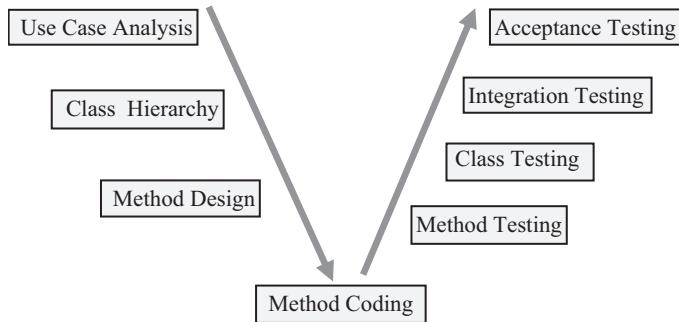


Figure 12.18 OO testing model

In OO testing methodology, the intra-class testing can be treated as unit testing and inter-class testing as integration testing.

First, the individual methods are tested. The methods are tested by programmers. They can be tested using black box methods. However, they are not the basic units of testing. Any amount of testing of an individual 'method' cannot ensure that a 'class' has been satisfactorily tested. Therefore, class testing is important in OO methodology.

Class test cases are created by examining the specification of the class. This includes testing of both private and public methods of class. The behaviors of the class and the interaction between attributes and methods are addressed in class testing. The following techniques are used for functional testing of class:

1. *State Transition testing*: Methods are modeled as state transition. Test cases are a sequence of method calls that traverse the state machine model.
2. *Transaction Flow testing*: A unit of work seen from the user's viewpoint is known as a transaction. This testing might also be called scenario-based testing.
3. *Exception testing*: This testing shows the exception throw from the unit tested.

Some of the functional test cases of Graphical User Interface (GUI) components are shown in Table 12.6.

Sometimes class testing may pose considerable problems, if the class is a subclass and inherits data and behavior from a superclass. A complicated class hierarchy can pose significant problems in testing. All class methods are tested during class testing. Testing a method in isolation from the rest of the class is usually meaningless.

Table 12.6 Test cases for functionality testing of GUI components

| Sr. No. | Test Case Id | Test Case Name | Steps/Action | Expected Results |
|---------|--------------|--|--|---|
| 1 | TC1 | Checking functionality of the Password textbox: i) Textbox for Password should accept more than six characters ii) Data should be displayed in encrypted format | i) The user enters only two characters in the password textbox ii) The user enters more than six characters in the password textbox iii) The user checks whether his data is displayed in the encrypted format | An error message is displayed when user enters less than six characters in the password textbox The system accepts data when user enters more than six characters into the password textbox The system accepts data in the encrypted format, else displays an error message |
| 2 | TC2 | Textbox for UserId should: i) allow only alphanumeric characters {a-z, A-Z} ii) not allow special characters like{\$', '#', '!', '~', '^', ...} iii) not allow numeric characters like{0-9} | i) The user types numbers into the textbox ii) The user types alphanumeric data into the textbox iii) The user types some special data into the textbox | An error message is displayed for numeric data The text is accepted when user enters alphanumeric data into the textbox An error message is displayed for special characters |
| 3 | TC3 | Checking functionality of SUBMIT button | i) The user checks whether SUBMIT button is enabled or disabled ii) The user clicks on the 'SUBMIT' button and expects to view the 'Home' page of the application | The system displays SUBMIT button as enabled The system is redirected to the 'Home' page of the application as soon as he clicks on the 'SUBMIT' button |
| 4 | TC4 | Checking functionality of CANCEL button | i) The user checks whether 'CANCEL' button is enabled or disabled ii) The user checks whether the textboxes for UserId and Password are reset to blank by clicking on the 'CANCEL' button | The system displays 'CANCEL' button as enabled The system clears the data available in the UserId and Password textbox when the user clicks on the 'CANCEL' button |

The interactions among a group of cooperating classes are tested under integration testing. Cluster testing is done using incremental testing. In incremental testing each unit (i.e. module or class) is tested in isolation and then integrated with other units and with the overall system. In the OO approach, integration testing can be applied on three different incremental strategies as given below:

1. *Thread-based testing*: This integrates classes required to respond to an input or event.
2. *Use-based testing*: This integrates classes required by one use case.
3. *Cluster testing*: This integrates classes required for collaboration.

Class clusters are a set of classes that are part of a class hierarchy or are the classes determined from a given use case.

12.6.1 Issues in OO Testing

OO features such as encapsulation, inheritance, polymorphism and dynamic binding are the major issues in OO testing. Since there is no proper hierarchy in the OO approach, the interactions among the various objects are of major importance in OO testing. Unit and system level testing is done in the same way as traditional testing. However, integration testing in the OO approach is different and a little more complex.

Integration testing involves testing of message quiescence and event quiescence. A method is programmed in an imperative language and performs a single cohesive function. Therefore, it corresponds to the unit level of traditional testing. High level functional requirements representing threads are tested by thread testing and thread interaction testing. These tests correspond to the traditional system of testing.

Encapsulation, inheritance and polymorphism are the three major aspects considered in integration testing. Encapsulation is not a source of errors. However, it is an obstacle to testing. It prevents accessing attribute values by a debugger. If the tester violates encapsulation for testing purposes, then the validity of test is questionable. One of the solutions to this issue is to have state reporting methods. Similarly, as inheritance is about “is-a” relationships between subclass and superclass, Unit testing a class with a superclass is impossible to do without the superclass methods. It is still complicated while dealing with multiple-inheritance. As a new context of usage results while deriving a subclass, retesting of inherited methods is required. In this context, a class is never considered as fully tested since it needs retesting when new subclasses are created. In polymorphism, an attribute may have more than one set of values. Thus, each possible binding of a polymorphic component requires separate testing. The above issues need to be taken care of in testing of OO systems.

12.6.2 State Transition Testing

A state-transition model describes the different states and transitions of a class in the context of its position in the inheritance hierarchy. It consists of states and transitions. The test cases are designed to cover all state transitions

For an illustration of state-transition, consider the checkout process in an e-commerce site. The state transition diagram of the process is shown in Figure 12.19.

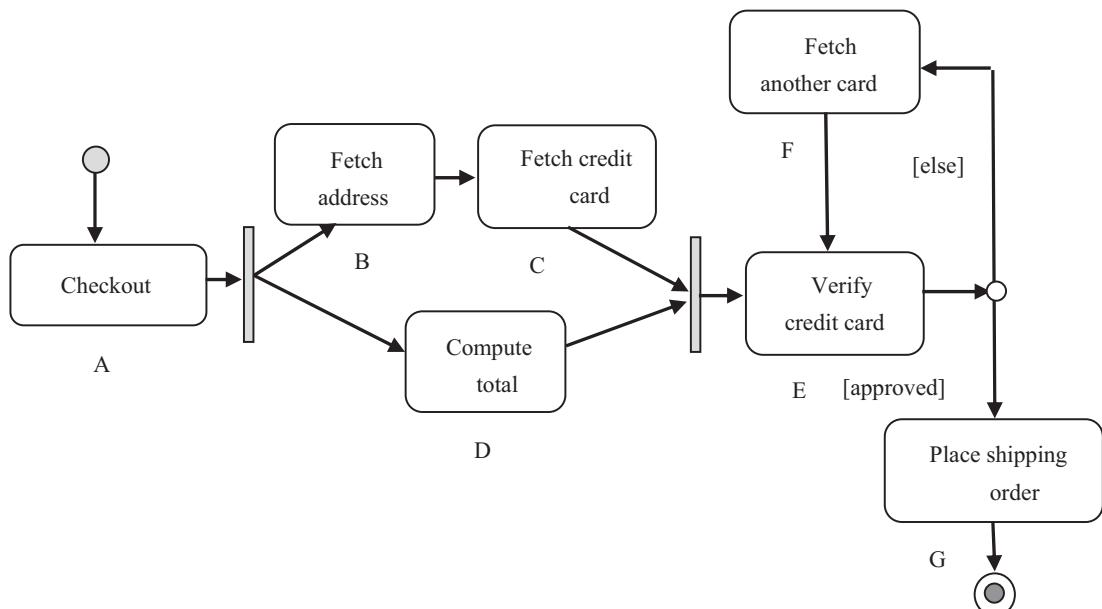
The checkout process can have two concurrent states, one to compute the total cost and another to process the credit card by fetching the card number and address. After the second synchronization bar, the card can be verified and if not approved another card can be fetched. After the card is approved, the shipping order is placed. Test cases corresponding to each transition path that represent a full object life cycle are created. Therefore, the paths ABCEG, ABCEFEG, ADEG and ADEFEG should be tested. Each transition should be tested at least once. Just testing the transition paths covering all states is weak level of test coverage, e.g. ABCEG and ADEFEG

A state-transition table can be written, which lists all possible state-transition combinations. State-transition is listed in tabular form by using four columns. These four columns are (1) Current state, (2) Event, (3) Action and (4) Next state. The state-transition is shown in Table 12.7.

Table 12.7 State transition table

| Current State | Event | Action | Next State |
|--------------------|-------------------|-----------------|--------------------|
| Null | buyItem | startPayTimer | Checkout |
| Null | Cancel | ---- | Null |
| Null | payTimerExpires | ---- | Null |
| Checkout | findsAddress | ---- | Fetch address |
| Checkout | computesTotal | ---- | Compute Total |
| Fetch address | produceCreditCard | Credit card no. | Fetch credit card |
| Fetch address | Print | ---- | Fetch address |
| Fetch credit card | checkPayment | ---- | Verify CreditCard |
| Compute Total | checkPayment | ---- | Verify CreditCard |
| Verify CreditCard | checkCreditCard | Credit card no. | FetchAnotherCard |
| Verify CreditCard | correctCreditCard | ---- | PlaceShippingOrder |
| Fetchdifferentcard | checkPayment | ---- | Verify CreditCard |

State transition combinations may contain some invalid ones that may not be considered while creating test cases. However, in critical systems such as medical or aviation devices, every state transition pair needs to be tested.

**Figure 12.19** State-transition diagram for Checkout

12.6.3 Transaction Flow Testing/Scenario-based Testing

As discussed above, a state transition can be used for making a transaction flow testing or a scenario-based testing. For illustration purposes, let us consider a particular scenario for authentication of a user in an ATM. The authentication scenario is shown in program 12.4.

The corresponding state diagram is shown in Figure 12.20.

Two extra states, e.g. Eject Card and Display Main menu are included in this state diagram.

After descriptive scenarios have been transformed into state transition diagrams, test cases for system test are generated by path traversal in the state transition diagrams. Any method to traverse all paths in the state transition diagrams according to a given criterion can be used to derive test cases. Path traversal in state transition diagrams generates tests for valid sequences of events.

Table 12.8 Test cases for the authentication scenario of an ATM card

| ID | State | Sequence of Actions | Expected Output |
|-----|------------|--|-------------------------|
| 1.1 | Check card | Card validated, valid Personal Identification Number (PIN) entered within the time frame | Main menu displayed |
| 1.2 | Check card | Card validated, invalid PIN entered at first try | Retry message displayed |
| 1.3 | Check card | Card validated, invalid PIN entered at second try | Retry message displayed |
| 1.4 | Check card | Card validated, invalid PIN entered at third try | Card Retained |
| 1.5 | Check card | Card invalid | Card Eject |

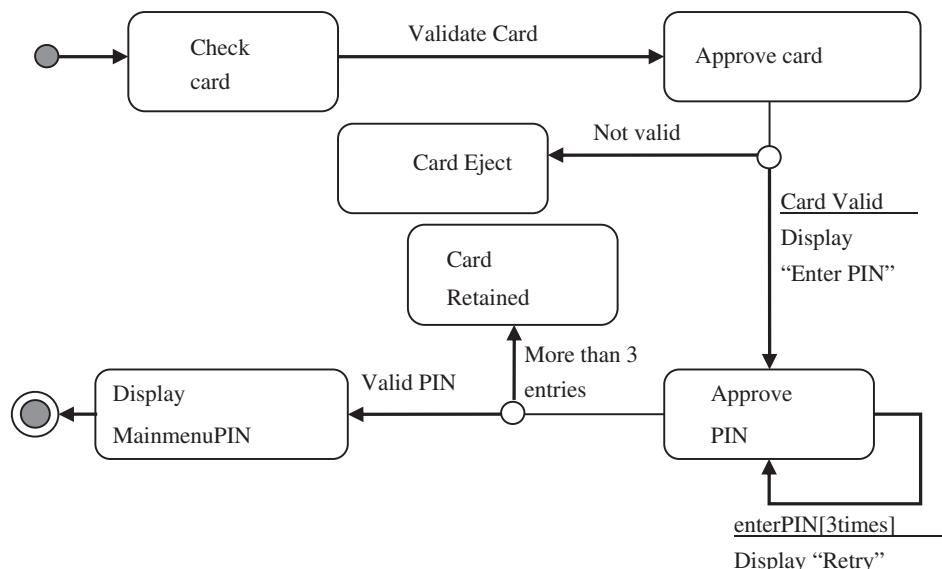


Figure 12.20 State diagram for authentication scenario

Derivation of test cases from state transition diagrams by path traversal is shown in Table 12.8. The table lists five test cases. Test case 1.1 follows the normal flow of actions; the card is inserted and the card as well as the PIN is validated. Test case 1.2 considers the exception of an incorrectly entered PIN. Similarly, Test case 1.3 considers a second invalid PIN entered and test case 1.4 indicates an invalid PIN being entered for the third time. Another exception of the card is that it may not be valid. This is put in test case 1.5.

12.7 SYSTEM TESTING

System testing is the testing of the whole system based on its specification. It is a comprehensive testing and verification of the system against the requirement specification. There are three major types of system testing. These are:

1. **Alpha Testing:** It refers to the system testing carried out by the test team within the developing organization.
2. **Beta testing:** It is the system testing performed by a select group of users. The users who are likely to have a positive attitude towards the development of the software are identified and invited to do the test. Here, the users act as a third party to perform the test and give their constructive feedback to the developers.
3. **Acceptance Testing:** It is the system testing performed by the customer(s) to determine whether they should accept the delivery of the system.

In each of the above types of tests, various kinds of test cases are designed by referring to the Software Requirement Specification (SRS) document. Mostly, these tests can be classified into functionality and performance tests. Functionality tests check whether the software meets or satisfies the functional requirements as documented in the SRS document.

Performance tests check whether the software meets the non-functional requirements of the system. All performance tests can be considered as black box tests. There are several types of performance testing. Some of these are listed below.

1. **Stress/peak-load testing:** It is also known as endurance testing. Stress testing evaluates system performance when it is stressed for abnormal resource demand. For example, if the Non Functional Requirement (NFR) specification states that the response time should not be more than 20 seconds per transaction when 50 concurrent users are working, then during stress testing the response time is checked with 50 concurrent users.

1. The customer inserts the card
2. The system checks the card's validity
3. The system displays the "Enter PIN" Dialog
4. The customer enters his PIN
5. The system checks the PIN
6. The system displays the main menu

Program 12.4: Scenarios for authenticating an ATM Card

2. **Volume testing:** It is especially important to check whether the data structures, e.g. arrays, queues, stacks etc., have been designed for extraordinary situations. For example, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled.
3. **Configuration testing:** It is used to analyze system behavior in various hardware and software configurations specified in the NFR. Sometimes systems are built in variable configurations for different users. For example, if the system is designed to serve a single user and there is an extension configuration to serve additional users, then the system should be configured in each of the required configurations and checked if it behaves correctly in all required configurations.
4. **Compatibility testing:** It is required when the system interfaces with other types of systems and is done to check whether the interface performs well as required. For example, if the system needs to communicate with a database system to retrieve huge information, compatibility testing is required to test the speed and accuracy of data retrieval.
5. **Recovery testing:** It is the performance testing that tests the response of the system when there is failure of power or failure of hardware. For example, the power may be shut down to check how the system recovers from the data loss and corruption.
6. **Documentation testing:** It is used for checking whether the software documentation complies with the system. Here, the user manual, maintenance manual and technical manual are tested against the requirements. If the requirements specify the types of users for whom a specific manual should be designed, then the manual should be checked for consistency.

The test cases should rerun every time after the system or its component has been corrected (made bug free). This is called *regression testing*.

12.8 USABILITY TESTING

Usability testing is done to find out whether the general users are experiencing any difficulty in using the software. In other words, usability testing determines the user-friendliness or ease of usage of the software. These testings are done by various groups of users. So the alpha, beta and acceptance testing as discussed in the system testing can also be part of usability testing. Usability testing is mostly concerned with checking user interface specifications. During this testing, the various GUI interfaces and aspects related to the user interface requirements are tested.

Usability is a quality attribute that assesses the easiness with which the user interfaces are used. Usability is defined by five major quality components:

- *Learnability:* It is the easiness with which the users accomplish basic tasks the first time they encounter the design.
- *Efficiency:* It refers to how quickly the users can perform tasks, once they learn the design.
- *Memorability:* It means the easiness with which the users can re-establish proficiency, when they return to the design after a period of time.
- *Errors:* It refers to the severity of the errors by the user and how easily they can recover from the errors.
- *Satisfaction:* It refers to the degree of satisfaction of using the design by the user.

With the growth of various web-based systems, the importance of usability testing has greatly increased. The web site visitors are connected to the Internet through a variety of communication channels. They use different web browsers with a variety of configurations. Some guidelines for usability of web-based system are listed below.

- It should contain simple and natural dialogue.
- It should speak the users' language, e.g. instead of providing a pop-up box indicating that there is some internal error, the system should state what the user should do.
- It should have simple pages and require minimum mouse travel and keystrokes.
- It should have shortcuts and clearly marked exits.
- It should minimize the users' memory load.
- It should be consistent and use standard icons, e.g. standardized navigation bars.
- It should provide feedback.
- It should have precise and constructive error messages.
- It should have help and documentation, e.g. user documentation.

Besides this some performance issues should be considered. Technology issues such as the speed of accessibility should also be considered. The GUI test design can be accomplished by scenarios and testing those as per the various requirements. Two scenarios for a web-based mail system are shown in Table 12.9.

Table 12.9 Scenario design for a web-based mail system

| | |
|-------------------|--|
| Scenario-1 | Registering for the mail system (Time: 10 minutes) |
| Scenario-2 | Reading, composing and sending messages (Time: 15 minutes) |
| | Task A: Check for email messages received from the system |
| | Task B: Write an email message and send it |
| | Task C: Check for a new message and respond to it |

Respective interfaces from both the scenarios can be tested. A detail test plan can be made considering the user interfaces and user interface navigation diagrams. This will take care of the functionality issues.

SUMMARY

Software testing is a process for verification and validation of software. Verification is concerned with whether the software product is being built right. Validation is concerned with whether the right software product is being built.

Testing is done at various levels. Accordingly, there are various types of testing, such as unit testing, integration testing, system testing and acceptance testing.

Testing is done in a planned way by constructing test cases. Test cases are devised with a view to test all possible aspects of the software product. Defect testing finds errors, whereas debugging finds the causes of error for making corrections.

Testing can be categorized into two types: (1) Black Box Testing and (2) White Box Testing.

The internal logic of the software is not looked into in black box testing. There are many methodologies for black box testing, namely, (1) decision table-based testing, (2) functional testing using cause–effect graphs, (3) testing by equivalence partitioning and (4) testing by boundary value analysis.

The internal structure and logical properties of the program are used for verification in White Box Testing. Some methods used for White Box Testing are: (1) Testing by Statement Coverage, (2) Testing by Branch/Edge Coverage, (3) Testing by Path Coverage, (4) Data Flow Testing, and (5) Mutation Testing.

In unit testing, the units/modules of a system are tested in isolation.

Integration testing is done to check whether the modules when integrated together as a system are able to function as required or not. Various types of integration testing are: (1) Big Bang Integration Testing, (2) Incremental Integration Testing, (3) Bottom-Up Integration Testing, (4) Top-Down Integration Testing, (5) Sandwiched Testing, (6) Backbone Integration Testing and (7) Thread Integration Testing.

The class is the basic testing unit of software developed through OO methodology. State transition testing, transaction flow testing and exception testing are some important techniques used for functional testing of classes. Intra-class testing can be treated as unit testing and inter-class testing as integration testing. Integration test is usually done in incremental steps. Thread-based testing, use-based testing and cluster testing are some strategies used in incremental testing. Integration testing involves testing of message quiescence and event quiescence.

In state transition testing, the test cases are designed to cover all state transitions. The testing can also be flow-based testing or a scenario-based testing.

System testing is the testing of the whole system based on its specification. There are three major types of system testing, namely, (1) Alpha Testing, (2) Beta Testing and (3) Acceptance Testing. Performance tests check whether the software meets the non-functional requirements of the system. Stress/Peak-load Testing, Volume Testing, Configuration Testing, Compatibility Testing, Recovery Testing, Documentation Testing are some important performance tests used for acceptance of a system. Usability testing determines the user-friendliness or ease of usage of software.

EXERCISES

1. Differentiate between verification and validation in software testing. Explain with the help of an example.
2. Write down differences between functional testing and structural testing. Explain using examples.
3. How can black box testing suite for software based on its SRS document be designed? Explain.
4. Identify guidelines for the design of equivalence classes for a problem.
5. Give an account of the various structural testing methods. Among statement coverage-based testing, branch coverage-based testing and condition coverage-based testing, which is the strongest structural testing technique? Explain it in the context of an e-commerce system.
6. What do you mean by data flow-based testing approach?
7. How can mutation testing be done? What are the advantages of performing mutation testing?
8. Give an account of a complete OO testing process. Explain using an ATM cash withdrawal process.
9. What do you mean by integration testing? What are the different types of integration testing methods that can be used for testing of a large software product?
10. Do you agree with the statement 'System testing is a pure black box test'? Justify your answer.
11. What do you understand by performance testing? Write down the different types of performance testing.
12. Give an account of usability testing. Explain its importance.

This page is intentionally left blank.

SOFTWARE METRICS

This chapter describes the basics of Software Metrics. It covers the following topics:

- *Product and Process Metrics*
- *Metrics in Various Phases of Software Engineering*
- *Design Metrics*
- *Metrics for object-oriented (OO) Systems*

The various metrics for measuring the cost of the software with examples are explained in this chapter.

The aim of a successful software project is to develop quality software within time, cost and resource constraints. This can be achieved consistently, only through effective management of the software development process. Well-defined measures of the process and the product are necessary to exercise control and to bring about improvement in the software development process. Software metrics are quantitative measures that provide the basis for effective management of the software development process. Software metrics is used to improve software productivity and quality. The various aspects of software metrics are discussed in this chapter.

13.1 SOFTWARE METRICS AND ITS CLASSIFICATION

Metric is a quantitative measure of the degree to which a given attribute is possessed by a system or its component or by a process. Software metrics are measures that are used to quantify different attributes of a software product, software development resource and software development process.

Software metrics deals with the estimation and measurement of different attributes of the software product and the software development process.

Software metrics can be classified into three major types. These are:

1. Product Metrics
2. Process Metrics
3. Resource Metrics

Product Metrics are the measures of different characteristics of the software product. The two important characteristics of software are:

1. Size and complexity of software
2. Quality and reliability of software

The metrics can be devised to measure software characteristics for different stages of its development, e.g. complexity of software design and size of the final program.

Process Metrics are the measures of different characteristics of the software development process. They quantify different aspects of the process being used to develop the software, e.g. efficiency of fault detection. They are used for measuring the characteristics of methods, techniques and tools that are used for developing the software system.

Resource Metrics are the quantitative measures of various resources used in the software project. Software projects use three major types of resources. These are: (1) Human resources, (2) Hardware resources and (3) Software resources. Resource metrics denote how efficiently the available resources are being used in the project. Project managers are concerned with these types of metrics.

The metrics can also be hybrid metrics that combine product, process and resource metrics, e.g. cost per function point (FP).

The metrics can also be classified as:

1. Internal Metrics and
2. External Metrics

The metrics used for measuring properties that are viewed to be of greater importance to a software developer are called internal metrics. Size metrics such as Line of Code (LOC) may be regarded as internal metrics. The metrics used for measuring properties that are viewed to be of greater importance to the user are called external metrics. The properties of a product visible to the users such as reliability, functionality, usability, performance etc. may be regarded as external metrics. They are mostly of subjective nature and are harder to measure compared to internal metrics. These are normally expressed in terms of internal metrics.

13.2 SOFTWARE SIZE METRICS

Most of the initial work in product metrics dealt with characteristics of source code. The most important product metric is the size of software. A number of different software size metrics have been proposed and used. Size is typically a direct count of selected characteristics to describe the volume of a software product. Size metrics can be expressed in three following ways:

1. Metrics expressed in terms of physical size of the program
2. Metrics expressed in terms of meaningful functions or services it provides
3. Metrics expressed in terms of logical size

The physical size is the count of architectural (and physical) items at a selected architectural level. Some examples of this kind of metrics are:

- LOC
- Number of classes
- Number of bytes used by the program

LOC is the most widely used size metric.

The functional size metrics try to quantify meaningful functions or services that the software provides to the users. The Function Point Analysis (FPA) as defined by the International Function Point User's Group (IFPUG) is a well-known metric used in this category. There are a number of variants of FP. Object Point (OP) method is one of them.

The logical size metrics try to combine logical complexity together with physical and functional size. Halstead's software science metrics and Function Bang can be regarded as logical size metrics.

13.2.1 LOC Metrics

Traditionally, the LOC or thousands of LOC (KLOC) is the primary measure of software size. Most LOC metrics count all executable instructions and data declarations, but exclude comments, blanks and continuation lines. LOC can be used to estimate size through analogy by comparing the new software's functionality to similar functionality found in other existing applications. Having more detailed information about the functionality of the new software clearly provides the basis for a better comparison. It enables recording size data needed to prepare accurate estimates for future projects. The most significant advantage of LOC estimates is that they directly relate to the software to be built.

However, it is difficult to relate software functional requirements to LOC, especially during the early stages of development. Estimation of LOC requires a level of detail that is hard to achieve. As LOCs are language specific, the definition of how LOCs are counted has been troublesome to standardize. This makes comparisons of size estimates between applications written in different programming languages difficult even though conversion factors are available.

13.2.2 Feature Point Metrics

Feature point (FP) metrics were developed in IBM in the year 1977 by A. J. Albrecht. They were put into the public domain in October 1978. A non-profit body called International Function Point Users Group (IFPUG) that was formed in 1986 assumed responsibility for developing counting rules and FP definitions. Today, IFPUG has grown to more than 3000 members and has affiliates in 24 countries. In addition to standard FP of IFPUG, more than 24 FP variations have been developed. Some of these are Back-fired FPs, COSMIC FPs, Finnish FPs, Engineering FPs, Feature Points, The Netherlands FPs, Unadjusted Function Points (UFPs) etc. However, standard IFPUG FP remains the primary version and majority of projects are measured using the IFPUG model.

FPs of an application are determined by counting the number and types of functions used in the applications. Various functions used in an application can be categorized into five types, namely: (1) External Inputs, (2) External Outputs, (3) Queries, (4) External Interfaces and (5) Internal Files. A brief definition of the five attributes used for FPs is as explained in Table 13.1.

Table 13.1 Definition of FP Attributes

| | |
|-------------------------------|--|
| 1. External inputs (EI): | All unique data or control inputs that cross the system boundary and cause processing to occur. Examples are input screens and tables |
| 2. External outputs (EO): | All unique data or control outputs that cross the system boundary after processing has occurred. Examples are output screens and reports |
| 3. External inquiries (EQ): | All unique transactions that cross the system boundary to make active demands on the system. Examples are prompts and interrupts |
| 4. Internal files (ILF): | All logical groupings of data that are stored within a system according to some pre-defined conceptual schema. Examples are databases and directories |
| 5. External interfaces (EIF): | All unique files or programs that cross the system boundary and are shared with at least one other system or application. Examples are shared databases and shared mathematical routines |

Each of these is then individually assessed for complexity. The value of weight varies from 3 (for simple external inputs) to 15 (for complex internal files). Values of standard weights for each category of function types are given in table 13.2.

Table 13.2 Weights of Various FP Attributes

| Function Type | Low | Average | High |
|-------------------------|-----|---------|------|
| Logical Internal File | 7 | 10 | 15 |
| External Interface File | 5 | 7 | 10 |
| External Input | 3 | 4 | 6 |
| External Output | 4 | 5 | 7 |
| External Inquiry | 3 | 4 | 6 |

The functional complexities are multiplied with the corresponding weights against each function and all the values are added up to determine the UFP of the subsystem.

UFPs are adjusted by considering some General System Characteristics (GSCs). A set of 14 GSCs has been developed by the New Environments Committee of IFPUG, which is given in Table 13.3.

Table 13.3 Categories of GSCs

| | |
|--------------------------------|-----------------------|
| 1. Data Communications | 8. On-Line Update |
| 2. Distributed Data Processing | 9. Complex Processing |
| 3. Performance | 10. Reusability |
| 4. Heavily Used Configuration | 11. Installation Ease |
| 5. Transaction Rate | 12. Operational Ease |
| 6. On-Line Data Entry | 13. Multiple Sites |
| 7. End-User Efficiency | 14. Facilitate Change |

The method for adjusting UFP is as under. First the Degree of Influence (DI) for each of 14 GSCs is assessed on a scale of 0 to 5. If a particular GSC has no influence its weight is taken as 0 and if it has strong influence its weight is 5. DIs and the corresponding weights are given in Table 13.4.

Table 13.4 Weights for Different DIs

| DI | Weight |
|-----------------------------|--------|
| Not present or no influence | 0 |
| Incidental influence | 1 |
| Moderate influence | 2 |
| Average influence | 3 |
| Significant influence | 4 |
| Strong influence throughout | 5 |

The score for all 14 GSCs is totalled to determine Total Degree of Influence (TDI). Then Value Adjustment Factor (VAF) is computed from TDI by using the formula:

$$\text{VAF} = (\text{TDI} * 0.01) + 0.65$$

The value of VAF lies within the range from 0.65 to 1.35. VAF is multiplied by the UFP to determine the Final FP Count.

$$FP = UFP \times VAF$$

FP metrics are mostly used for measuring the size of management information system (MIS) software. Some differences between FP metrics and LOC metrics are given in Table 13.5.

Table 13.5 Difference between FP and LOC

| FP | LOC |
|----------------------|--------------------|
| Specification based | Analogy based |
| Language independent | Language dependent |
| User oriented | Design oriented |
| Extendable to LOC | Convertible to FP |

LOCs of an application can be estimated from FPs. LOC is language specific. For the same application, LOC will be different depending on programming language being used. For comparison purposes, the LOC per FP for different programming languages is shown in Table 13.6.

Table 13.6 Ratios of LOC to FP for Different Languages

| Programming Languages | Ratio (LOC/FP) |
|-----------------------|----------------|
| Assembly Language | 320 |
| C | 128 |
| COBOL | 106 |
| FORTRAN | 106 |
| Pascal | 90 |
| C++ | 64 |
| Ada95 | 53 |
| Visual Basic | 32 |
| Smalltalk | 22 |
| Powerbuilder | 16 |
| SQL | 12 |

These data can be used to compute LOC from the FP measure.

13.2.3 Feature Point Metrics

Feature Points is another size metric. It is similar to 'FPs'. It is used primarily for software having high algorithmic complexity such as real-time and embedded systems.

OP is similar to FP. It looks at the number of objects (screen, reports etc.) likely to be generated in the software. The feature point method gives more weight to algorithm. For applications in which the number of algorithms and logical data files are the same, function and feature point counts generate the same numeric values. However, when there are more algorithms than files, feature points produce a greater total than FPs. This is illustrated in Table 13.7, which shows ratio of FP to feature point counts for different types of applications.

Table 13.7 Ratios of Feature Points to FP

| Application | FP | Feature Point |
|-----------------------------|----|---------------|
| Batch MIS Projects | 1 | 0.8 |
| Online MIS Projects | 1 | 1 |
| Online Database Projects | 1 | 1 |
| Switching Systems Projects | 1 | 1.2 |
| Embedded Real-time Projects | 1 | 1.35 |
| Factory Automation Projects | 1 | 1.5 |

13.2.4 Bang Metrics

DeMarco defines system Bang as a function metric, indicative of the size of the system. It measures the total functionality of the software system delivered to the user. Bang can be calculated from certain algorithm and data primitives available from a set of formal specifications for the software. The model provides different formulas and criteria for distinguishing between complex algorithmic systems versus heavily data-oriented systems. Since Bang measures the functionality delivered to the user, DeMarco suggests that a reasonable project goal is to maximize “Bang per Buck”, i.e. Bang divided by the total project cost.

13.2.5 Halstead's Metrics

Most of the product metrics are applied to only one particular aspect of the software product. In contrast, Halstead's software science proposed a unified set of metrics that apply to several aspects of programs as well as to the overall software production effort. Thus, it is the first set of software metrics unified by a common theoretical basis. The Halstead product metrics estimate the program vocabulary (n), length (N) and volume (V) metrics as shown below:

$$\text{Program length } (N) = N_1 + N_2$$

Where

N_1 – total number of operators

N_2 – total number of operands

$$\text{Program vocabulary } (n) = n_1 + n_2$$

Where

n_1 – number of distinct operators

n_2 – number of distinct operands

$$\text{Estimated length for well structured programs } (N') = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

$$\text{Purity ratio } (\text{PR}) = N'/N$$

$$\text{Volume, i.e. number of bits to provide a unique designator for each of the } n \text{ items in the program vocabulary } (V) = N \log_2 n$$

For example, the program segment as shown in program 13.1 has the following distinct operators and operands from which the metrics can be estimated.

Different operators used in this piece of code are:

if () if () { } > < = * ;

```

if (a > 20)
{
    if (a < 30) then
        b = b * a;
}

```

Program 13.1: Program for Statement Coverage Testing

Among these the distinct operators are:

if () { } > < = * ;

Similarly different operands used are:

a 20 a 30 b b a

And distinct operands are:

a 20 30 b

Hence, the total number of operators (N_1) and operands (N_2) used are 13 and 7, respectively. Similarly, total number of distinct operators (n_1) and distinct operands (n_2) used are 10 and 4, respectively.

Hence, program length (N) is $13 + 7 = 20$

and program vocabulary (n) is $10 + 4 = 14$

The estimated length for well structured programs

$$\begin{aligned}
 N' &= n_1 \log_2 n_1 + n_2 \log_2 n_2 \\
 &= 10 \log_2 10 + 4 \log_2 4 \\
 &= 41.2
 \end{aligned}$$

$$PR = N'/N = 41.2/20 = 2.06$$

These metrics apply specifically to the final software product.

13.3 QUALITY METRICS

Quality is an important aspect of any product. Early work on quality metrics were done by Boehm, McCall and others. There are a number of characteristics of quality. Correctness, efficiency, portability, maintainability, reliability etc. are some of the quality characteristics of any software. The quality characteristics may overlap and conflict with one another. For example, increased portability, which is desirable, may result in lowered efficiency, which is undesirable. Thus, useful definitions of general quality metrics are difficult to devise.

McCall's quality factors were proposed in the early 1970s. They are even valid today as they were in that time. It is expected that the software built to confirm these factors will exhibit high quality. McCall identified 55 quality characteristics that have significant influence on quality, and called them "factors". For reasons of simplicity, McCall then reduced the number of characteristics to the following eleven:

- | | |
|------------------|------------------------|
| i) Efficiency | vii) Testability |
| ii) Integrity | viii) Flexibility |
| iii) Reliability | ix) Interface facility |
| iv) Usability | x) Re-usability |

- v) Accuracy xi) Transferability
- vi) Maintainability

Boëhm proposed another set of quality factors in 1978 as given below.

- | | | |
|------------------|------------------------|-------------------|
| i) Usability | viii) Documentation | xv) Integrity |
| ii) Clarity | ix) Resilience | xvi) Validity |
| iii) Efficiency | x) Correctness | xvii) Flexibility |
| iv) Reliability | xi) Maintainability | xviii) Generality |
| v) Modifiability | xii) Portability | xix) Economy |
| vi) Re-usability | xiii) Interoperability | |
| vii) Modularity | xiv) Understandability | |

There are certain quality characteristics that are more visible to the users. These are called external quality. Users of the software generally care about the external qualities. However, it is the internal qualities that help developers achieve external qualities. Internal qualities deal with the structure of the software. Some external quality and internal product quality characteristics are given in Table 13.8.

Table 13.8 External and Internal Product Quality

| External Quality | Internal Quality | |
|------------------|------------------|--------------------|
| Integrity | Efficiency | Interface Facility |
| Reliability | Maintainability | Reusability |
| Usability | Testability | Transferability |
| Accuracy | Flexibility | |

Regardless of the various software qualities, the quality metrics mainly concentrate on two aspects: (1) *Intrinsic Product Quality* and (2) *Customer Satisfaction*.

Mean Time to Failure (MTTF) and Defect Density (DD) are two important metrics of product quality. The MTTF metric is most often used to specify safety aspects of critical systems, e.g. airline traffic control systems. These metrics attempt to measure and predict the probability of failure during a particular time interval.

The mean time to repair (MTTR) is the probability of repairing the software during a particular time interval. Thus, MTTR should be less for the software to be highly maintainable. Availability is the probability that the system will still be operating to the requirements at a given time. It can be measured by the formula shown.

$$\text{availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100$$

In contrast, the DD metric is used in many commercial software systems. The DD metrics measure the defects relative to the software size. DD is a measure of the total known defects divided by the size of the software entity being measured.

$$\text{DD} = \text{Number of known defects}/\text{Size}$$

The number of known defects is the count of total defects identified against a particular software entity during a particular time period. Examples include:

- Defect to date since the creation of the module
- Defects found in a program during an inspection
- Defects to date since the shipment of a release to the customer

The number of defects in the software product should be readily derivable from the product itself and, hence, it qualifies as a product metric. However, since there is no effective procedure for counting the defects in the program, the following alternative measures are proposed in various phases of the software development life cycle.

- Number of design changes
- Number of errors detected by code inspections
- Number of errors detected in program tests
- Number of code changes required

The customer satisfaction metrics are often measured by the customer survey data. Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys.

Companies use various parameters of customer satisfaction. For example, IBM uses CUPRIMDSO that includes capability (functionality), usability, performance, reliability, installability, maintainability, documentation, service and overall satisfaction. For Hewlett-Packard (HP), the customer satisfaction is measured by FURPS, i.e. functionality, usability, reliability, performance and service.

In addition to forming percentages for various satisfaction or dissatisfaction categories, the weighted index approach can also be used. For example, some companies use Net Satisfaction Index (NSI) to facilitate comparisons across the product. NSI has the following weighting factors:

- Completely satisfied = 100%
- Satisfied = 75%
- Neutral = 50%
- Dissatisfied = 25%
- Completely dissatisfied = 0%

13.4 PROCESS METRICS

The modern management approach gives much emphasis on process. It stresses that if the process is right and it is rightly managed, the output (product) of the process will be right. Process metrics indicates whether the process being followed for development of software is right and whether the process is being managed well.

There is need for software metrics to predict, plan and control software development, thereby improving the ability of software project managers to manage the software development process

The various software process metrics can be divided into the following categories based on process parameters as given in Table 13.9.

The process metrics can also be divided into:

(1) External Process Quality and (2) Internal Process Quality

The important internal and external process qualities are shown in Table 13.10.

Table 13.9 Types of Process Metrics based on Process Parameters

| Process Parameter | Description of Metrics |
|-------------------------------|---|
| Quality-related | Focus on quality of work products and deliverables |
| Productivity-related | Production of work-products related to effort expended |
| Statistical quality assurance | Data, e.g. error categorization and analysis |
| Defect removal efficiency | Propagation of errors from process activity to activity |
| Component reuse | The number of components produced and their degree of reusability |

Table 13.10 External and Internal Process Quality

| External Process Quality | Internal Process Quality |
|--------------------------|----------------------------|
| Cost-effectiveness | Time |
| Stability | Effort |
| | Number of requirement spec |
| | Number of design spec |
| | Number of coding fault |

13.4.1 Halstead's Metrics

The Halstead software science equations can also be used as a theoretical model of the software development process. The effort required to develop the software is given by the equation $E=V/L$, which can be approximated by:

$$E = \{n_1 n_2 [n_1 \log_2 n_1 + n_2 \log_2 n_2] \log_2 n\}/2 n_2$$

The units of E are elementary mental discriminations. The corresponding programming time in seconds is simply derived from E by dividing by the Stroud number, S , i.e. E/S .

The value of S is usually taken as 18 for these calculations. If only the value of length, N , is known, then the following approximation can be used for computing T :

$$T = N^2 \log n / 4S$$

Where n can be obtained from the relationship

$$N = n \log_2 (n/2)$$

13.4.2 Defect Estimation

A number of dynamic models have been developed to estimate software defects. These models attempt to describe the occurrence of defects as a function of time, allowing one to define the reliability, R , and MTTF. One example is the model described by Musa, which is based on the following four assumptions:

- Test inputs are random samples from the input environment.
- All software failures are observed.

- Failure intervals are independent of each other.
- Times between failures are exponentially distributed.

Based upon these assumptions, the following relationships can be derived:

$$d(t) = D(1 - e^{-bct}) ,$$

where

D is the total number of defects;

'b' and 'c' are constants. Their values are determined from historical data for similar software;

$d(t)$ is the number (cumulative total) of defects discovered in time t ;

MTTF is determined as:

$$\text{MTTF}(t) = e^{bct}/c^D$$

As in many other software models, the determination of b, c and D is sometimes considered as a non-trivial task. However, it is vitally important for the success of the model.

13.5 DESIGN METRICS

FP metrics are good enough for estimating cost. They are most suitable for estimation during the requirement stage. Design metrics are more suitable for estimation during the design stage. While the architectural design (high-level design) metrics mostly include structural metrics, component-level design metrics include coupling and cohesion metrics. Some of the interface design metrics are also discussed in this section.

13.5.1 High-level Design Metrics

High-level or architectural design metrics focus on the characteristics of program architecture. They emphasize on the architectural structure and effectiveness of modules. Three types of measures may be defined for software design complexity, namely: (1) Structural complexity, (2) Data complexity and (3) System complexity.

The structural complexity of a module ' i ' is defined as:

$$S(i) = f_{\text{out}}^2(i)$$

Where $f_{\text{out}}(i)$ is the fan-out of the module i

Fan out is the number of modules subordinate to the module ' i ', i.e. immediately followed and directly invoked.

Data complexity provides an indication of the complexity of the internal interface for a module i as:

$$D(i) = \frac{v(i)}{f_{\text{out}}(i) + 1}$$

Where $v(i)$ is the number of inputs and outputs passed to and from module i

System complexity is defined as the sum of structural and data complexity, specified as:

$$C(i) = S(i) + D(i)$$

As the value of each of these complexities, e.g. $S(i)$ and $D(i)$, increases, the overall complexity of the architecture increases. This leads to a greater likelihood that integration and testing effort will also increase.

Fenton suggests a number of *simple morphology* metrics that enable different program architectures to be compared using a set of straightforward dimensions.

$$\text{Size} = n + a$$

where ' n ' is the number of nodes and ' a ' is number of arcs.

Depth, width and arc-to-node ratio are some other *simple morphology* metrics.

Depth = the longest path from the root node to a leaf node

Width = maximum number of nodes at any one level of the architecture

13.5.2 Component-level Design Metrics

Component-level design metrics focus on internal characteristics of a software component and include measures of the 'three Cs'. These are: (1) Cohesion, (2) Coupling and (3) Complexity. These measures can quantify the quality of a component-level design. However, the complexities of the components are already discussed in McCabe's metrics. Cohesion and coupling have already been discussed in earlier chapters. Though, object-oriented (OO) metrics are discussed in a separate section following this section, only the cohesion and coupling metrics related to the component design are discussed in this section.

Cohesion metrics are defined by collection metrics that provide an indication of the cohesiveness of a module. The metrics are defined in terms of five concepts and measures.

1. Data slice is a backward walk through a module that looks for data values that affect the module location from which the walk began. It should be noted that both program slices consisting of statements and conditions and data slices can be defined.
2. Data tokens are the variables defined for a module.
3. Glue tokens are a set of data tokens that lie on one or more data slices.
4. Superglue tokens are data tokens common to every data slice in a module.
5. Stickiness is the binding of the number of data slices by the glue token.

Metrics are developed for strong functional cohesion (SFC), weak functional cohesion (WFC) and adhesiveness, i.e. the relative degree to which glue tokens bind data slices together. All of these cohesion metrics range between 0 and 1. These metrics have a value of 0 when a procedure has more than one output and exhibit none of the cohesion attributes indicated by a particular metric. A procedure with no superglue tokens (i.e. no tokens that are common to all data slices) exhibits zero SFC. A procedure with no glue tokens, i.e. no tokens common to more than one data slice, exhibits zero WFC and zero adhesiveness.

A metric for SFC can be described by:

$$\text{SFC}(i) = \text{SG}[\text{SA}(i)] / (\text{tokens}(i))$$

where $\text{SG}[\text{SA}(i)]$ denotes superglue tokens the set of data tokens that lie on all data slices for a module. As the ratio of superglue tokens to the total number of tokens in a module increases toward a maximum value of 1, the functional cohesiveness of the module also increases.

Coupling metrics provide an indication of the 'connectedness' of a module to other modules, global data and the outside environment. The metric for module coupling mainly includes three types of coupling: (1) data and control flow coupling, (2) global coupling and (3) environmental coupling.

Data and control flow coupling is a function of:

- d_i as the number of input data parameters
- c_i as the number of input control parameters
- d_o as the number of output data parameters
- c_o as the number of output control parameters

Global coupling is a function of:

- g_d as the number of global variables for data
- g_c as the number of global variables for control

Environmental coupling is a function of:

- w as the number of fan in
- r as the number of fan out

Hence, module coupling is a function of all the above parameters, which denote the amount of interaction of the module with other modules or environment. When the interaction is less, coupling is more.

13.6 OBJECT-ORIENTED METRICS

Design metrics can be divided into both traditional and OO metrics. Traditional design metrics are already discussed in the last section and OO design metrics are discussed in this section.

OO metrics have been developed for various aspects such as Complexity, Coupling, Cohesion, Inheritance, Polymorphism and Information hiding. CK Metrics Suite and Metrics for Object-Oriented Design (MOOD) are two popular OO metrics.

13.6.1 CK Metrics Suite

Methods design involves defining procedures that implement the attributes and operations undergone by objects. Based on these design issues, Chidamber and Kemerer defined a metrics suite popularly known as CK metrics. This metrics suite has generated a significant amount of interest and is currently a well-known suite of measurements for OO software. Chidamber and Kemerer proposed six metrics as discussed below.

Weighted methods per class (WMC): It relates directly to the definition of complexity of an object, since methods are properties of objects and complexity of an object is determined by the cardinality of its set of properties. Therefore, if c_1, c_2, \dots, c_n are the static complexities of the n methods m_1, m_2, \dots, m_n , then

$$WMC = \sum_{i=1}^n c_i$$

Depth of inheritance tree (DIT): It is a measure of how many ancestor classes can potentially affect this class. It is the maximum length from a node to the root, i.e. the base class.

Number of children (NOC): It is a measure to find how many subclasses are going to inherit the methods of the parent class.

NOC = Number of immediate subclasses subordinated to a class in the class hierarchy

Coupling between objects (CBO): It relates to the notion that an object is coupled to another object if two objects act upon each other, i.e. methods of one class use methods or attributes of another class.

CBO of a class = Number of non-inheritance related couples with other classes

Response for a class (RFC): The response set is a set of methods available to the object and its cardinality is a measure of the attributes of an object. Since it specifically includes methods called from outside the object, it is also a measure of communication between objects.

RFC of a class = Number of methods that could be called in response to a message to a class including both local and remote

Lack of Cohesion in Methods (LCOM): This uses the notion of degree of similarity of methods. The degree of similarity for the methods in class C_i is given by:

$$I_1 \cap I_2 \cap \dots \cap I_n$$

where class C_i with n methods M_1, \dots, M_n and I_j is the set of instance variables used by M_j

LCOM = Number of disjoint sets formed by the intersection of ' n ' sets

13.6.2 Metrics for Object-oriented Design

Metrics for Object-oriented Design (MOOD) metrics set refers to a basic structural mechanism of the OO paradigm as encapsulation (method hiding factor (MHF) and attribute hiding factor (AHF)), inheritance (method inheritance factor (MIF) and attribute inheritance factor (AIF)), polymorphism factor (PF) and coupling factor (CF) as discussed below:

MIF: MIF metrics are based on the sum of inherited methods in all classes of the system under consideration. It is defined as the ratio of the sum of the inherited methods in all classes of the system as follows:

$$MIF = \frac{\sum_{i=1}^n M_a(C_i)}{\sum_{i=1}^n M_a(C_i)}$$

where $M_a(C_i)$ is the number of methods that can be invoked with class C_i and ' n ' is the total number of classes. The number of methods that can be invoked with a class is the sum of number of methods declared in class and number of methods inherited by the class.

AIF: AIF metrics are based on the sum of inherited attributes in all classes of the system under consideration. They are defined as the ratio of the sum of the inherited attributes in all classes to the total number of available attributes for all classes. The AIF of the system is expressed as follows:

$$AIF = \frac{\sum_{i=1}^n A_a(C_i)}{\sum_{i=1}^n A_a(C_i)}$$

Where $A_a(C_i)$ is the number of attributes defined in class C_i and ' n ' is the total number of classes. The number of attributes defined in a class is the sum of number attributes declared in the class and number of inherited attributes.

MHF: MHF metrics are based on the sum of invisibilities of all methods in all classes of the system under consideration. They are defined as the ratio of the sum of the hidden methods in all classes of the system as follows:

$$MHF = \frac{\sum_{i=1}^n M_h(C_i)}{\sum_{i=1}^n M_d(C_i)}$$

Where $M_d(C_i)$ is the total number of methods defined in class C_i and ' n ' is the total number of classes. The number of methods defined in the class is the sum of the number of methods visible in the class and number of methods hidden in the class.

AHF: AHF metrics are based on the sum of invisibilities of all attributes in all classes of the system under consideration. They are defined as the ratio of the sum of the hidden attributes in all classes to the total number of available attributes for all classes. The AHF of the system is defined as follows:

$$AHF = \frac{\sum_{i=1}^n A_h(C_i)}{\sum_{i=1}^n A_d(C_i)}$$

Where $A_d(C_i)$ is the total number of attributes defined in class C_i and ' n ' is the total number of classes. The total number of attributes defined in a class is the number of visible attributes and the number of hidden attributes.

PF: PF metrics represent the actual number of possible different polymorphic situations. Thus, they represent the maximum number of possible distinct polymorphic situations for class C_i . They are defined as follows:

$$PF = \frac{\sum_i M_o(C_i)}{\sum_i [M_n(C_i) * DC(C_i)]}$$

Where $M_o(C_i)$ is the number of overriding methods in class C_i , $M_n(C_i)$ is the number of new methods declared in class C_i and $DC(C_i)$ is the number of descendants count in C_i .

CF: CF metrics measure the coupling between classes excluding coupling due to inheritance. They are defined as the ratio of the maximum possible number of couplings in the system to the actual number of couplings as defined below:

$$CF = \frac{\sum_i \sum_j is_client(C_i, C_j)}{(n^2 - n)}$$

Where $is_client(C_i, C_j) = 1$ if class i has a relation with class j ; otherwise, it is zero. The relationship might be that class i calls a method in class j or has a reference to class j or to an attribute in class j . This relationship cannot be an inheritance.

SUMMARY

Software metrics are measures that are used to quantify different attributes of the software product, software development resource and software development process.

Software metrics deal with the estimation and measurement of different attributes of the software product and the software development process.

Software metrics can be classified into Product metrics, Process metrics and Resource metrics. Product metrics are the measures of different characteristics of software product, such as size, complexity, quality, reliability etc. Process metrics are the measures of the software development process, e.g. efficiency of fault detection. They are used for measuring the characteristics of methods, techniques and tools employed in developing the software system. Resource metrics are the quantitative measures of various resources used in a software project.

The metrics can also be classified as internal metrics and external metrics. The metrics that are of greater importance to the software developer are called internal metrics whereas metrics that of greater importance to the user are called external metrics.

Size metrics are important product metrics. They can be expressed as physical size of the program in terms of LOC, meaningful functions in terms of FP or logical size in terms of Halstead's software science metrics.

Quality is an important aspect of any product. Quality characteristics that are more visible to the users are called external qualities. Quality characteristics that may not be visible to the user but help to achieve external qualities are called internal qualities. Integrity, reliability, usability and accuracy are some external quality characteristics. Efficiency, interface facility, maintainability, reusability, testability, transferability and flexibility are some external quality characteristics.

Software quality metrics mainly concentrate on Intrinsic Product Quality and Customer Satisfaction. MTTF and DD are two important metrics of product quality. Functionality, usability, reliability, performance and service are some measures of customer satisfaction. These are often measured by the customer survey data.

Process metrics indicate whether the process is right and whether it is rightly managed. The metrics for various aspects of software process may be related to quality, productivity, statistical quality assurance, defect removal efficiency and component reuse. The Halstead software science equations can also be used as a theoretical model of the software development process.

Design metrics are more suitable for estimation during the design stage. Structural complexity, data complexity and System complexity are some architectural design (high-level design) metrics. Cohesion, Coupling and Complexity are some component-level design metrics.

OO design metrics have been developed to measure various OO aspects such as Complexity, Coupling, Cohesion, Inheritance, Polymorphism and Information hiding. The CK Metrics Suite and MOOD are two popular OO metrics.

EXERCISES

1. Explain how software metrics can be classified.
2. Give an account of FP Metrics and using it compute the FP value for a project with the following information domain characteristics:

Number of user inputs = 32

Number of user outputs = 60

Number of user inquiries = 24

Number of files = 8

Number of external interfaces = 2

3. Describe the difference between product and process metrics. Explain citing examples.
4. Give an account of the LOC metric. What is the difference between FP and LOC? How to find FP from LOC? Explain using an example.
5. What are the various types of functions that are counted for determining FPs of an application? Why do UFPs need to be adjusted to determine final FP count?
6. Describe the various metrics in the different phases of the software development life cycle. Explain the metrics with examples.
7. Create a design model that can compute structural and data complexity. Explain using examples.
8. Explain how Halstead metrics can be used for software product and software process.
9. Distinguish between external quality and internal quality characteristics with examples.
10. Explain MTTF and DD.
11. What do you mean by design metrics? What are different high-level design metrics and low-level design metrics?
12. Give an account of various OO metrics and explain a few among them that directly or indirectly address information-hiding characteristics.
13. Give the formal definition of the LCOM metric. Explain using an example.

This page is intentionally left blank.

SOFTWARE PROJECT ESTIMATION

Effort, time, resources and money are required for doing any project. The estimates of these parameters are the basis of any project plan. This chapter covers approaches, techniques and models used in software project estimation. It discusses:

- Various parameters of a software project
- Various approaches to project estimation
- Classification of software projects
- Overview of COCOMO

The objective of a software project is to develop software for a given scope and quality requirements within specified time and cost limits. The cost of software development is the main determinant of its price. The time limit specified in the software contract is based on the estimate of time needed to complete the project. Availability of resources (people, infrastructure, equipments) is also an important parameter. Underestimating the project time, cost and resource requirements means loss to the developer. Overestimating these parameters means that the project may be considered not viable and thus it may not be taken up. Hence, making accurate estimates of time, cost and resources is crucial for planning and execution of a project. For this, the project estimation needs to be done in a systematic and structured manner.

14.1 SOFTWARE PROJECT PARAMETERS

Estimation of various project parameters is the first step of project planning. The cost of development, effort, resources and time are important parameters of software projects.

Cost of development: It determines the viability and success of a software development project. A project is considered successful, if the revenue earned from the completed project is more than the cost incurred. Estimation of cost is necessary to determine project feasibility. It also helps in quoting the project cost or how much money should be charged to develop the software for a client.

Effort: The cost of software development is a function of effort required to develop the software. The effort is expressed in terms of person-months (PMs). The execution of any work generally requires effort

from a number of people. Suppose three persons devote 3 months, 2 months and 1 month of their time, respectively, to do a piece of work, then the total effort used in the work is 6 PMs.

Resource: Human resource is the major resource of a software development project. The software development is knowledge work. It requires effort from a number of people such as software engineer, system analyst, programmer, data-entry operator etc. Each of these computer personnel may have different types and levels of computer-related expertise/skill. The estimation of resource involves determination of the number of persons of different categories based on levels and types of computer skill/expertise needed for software development. Since high-skilled people are paid more, the cost of the software project depends not only on the number of human resource but also on their skill level.

Time: Duration of a project is an important criterion for the success of any project. The benefit of any project accrues only after the project is completed. An incomplete project is like locked capital or inventory. Project duration has relationship with cost, effort and resource. Sometimes by assigning more people to the project or by incurring more cost, it is possible to reduce project duration to some extent. However, relationship is not linear. The time needed to complete any knowledge work is not necessarily reduced by adding more people. For example, time needed to perform a heart operation cannot be reduced by adding more number of doctors. This example is also applicable to software development. Barry Boehm, a renowned expert of software project management, has stressed this fact by his following statement:

“Adding more people to a late project will make it later.”

-- Barry W Boehm

The correct estimate of the above parameters is the basis of project planning. Incorrect estimation is a major cause of software project failure.

14.2 APPROACHES TO SOFTWARE ESTIMATION

There are two approaches to cost estimation: (1) top-down approach and (2) bottom-up approach. A top-down approach starts at the system level. It is done by examining the overall functionality of the software product and how the functionality is provided by interacting subfunctions. The costs of system-level activities such as integration, configuration management and documentation are taken into account.

Top-down estimation is easier to apply. Top-down estimation has two major drawbacks:

1. It underestimates the costs of solving difficult technical problems associated with specific users' requirements, such as problems relating to system integration with nonstandard hardware, system security and audit, real-time distributed processing etc.
2. It does not provide detailed justification of the estimate, which is often necessary for getting into a contract with a client.

The bottom-up approach, by contrast, starts at the component level. The system is decomposed into components, and the effort required to develop each of these individual components is estimated. Then these component costs are added up to compute the effort required for the whole system development. Hence, the bottom-up approach provides detailed justification of the estimate. The detailed cost break-up appears to be a rational basis to software estimation, which is more acceptable to the clients.

Hence, to apply bottom-up approach, it is necessary to identify various components of the software to be estimated. This may require some form of preliminary high-level system analysis and design. Hence, bottom-up estimation is more costly than top-down estimation. Bottom-up approach is more likely to underestimate the costs of system activities such as integration, system testing etc.

14.3 PROJECT ESTIMATION TECHNIQUES

Estimation of software projects can be done by different techniques. The important techniques are:

1. Estimation by Expert Judgement
2. Estimation by Analogy
3. Estimation by Available Resources
4. Estimation by Software Price
5. Estimation by Parametric Modeling

14.3.1 Estimation by Expert Judgement

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes a subjective assessment of the software size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the software and then combines them to arrive at the overall estimate.

This technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have the knowledge of all aspects of a project. For example, a person may be expert in Data Base Management System (DBMS) but may not be knowledgeable enough in network programming. To overcome this problem, the estimation is made by a group of experts. Estimation by a group of experts minimizes errors due to individual oversight, personal bias and lack of familiarity with a particular aspect of a project. However, the process of estimation by a group of experts may face two types of problems:

1. Since in a group the responsibility is shared among members, the individual members may not devote enough effort to ensure correct estimation of the software project.
2. All members of the expert group do not have equal knowledge, experience and status. Also, behaviorally they may have different nature. Hence, the group may be dominated by one or two assertive members and members who are not assertive may fail to contribute to the group decision.

Various techniques have been developed to enhance group decision-making and group creativity.

Delphi Method: The Delphi method is a systemic procedure to obtain the best solution by using the collective wisdom of all the experts. In Delphi method the working of group activities for estimation of the software project are coordinated in a planned manner by a group coordinator. Usually, one of the experts of the group acts as the coordinator. However, sometimes an outsider may also be the coordinator.

The coordinator provides the copy of the software requirements specification (SRS) and other relevant details to the members of the expert group. Then each member completes his estimate individually in a specified format and submits it to the coordinator. The experts also note their observation regarding existence of any special characteristic of the software product, which has influenced their estimation. The coordinator prepares the summary of the responses (that also includes remark of the experts) and distributes it among the members of the expert group. The members of the expert group review their earlier estimates in light of this summary, and submit their revised estimates individually. Total anonymity is observed in the entire process and no discussion among the members is allowed. Thus, one member of the expert group is not influenced by the personality and status of other members. This process may be repeated for several rounds. The gap between individual estimates gets reduced in

successive iterations till the estimates by individual members are quite close to each other. This helps in preparing the final estimate.

14.3.2 Estimation by Analogy

If the project to be estimated is similar to other projects that have been completed in the past, then estimation is done easily by analogy with these completed projects. However, in the field of software development no two software are totally identical or the same. There will be some differences between two similar projects even if these are in the same application domain. The software project to be estimated may have some similarity as well as some differences to varying degrees with a number of other completed projects. The problem here is to identify the similarities and differences between two projects and to quantify the degree of their similarity. One approach to determine the similarity is to rank the similarity between two projects on various software and project metrics/parameters. For this, models as well as software have also been developed to determine the degree of similarity between two projects. The estimation of cost of development, effort, project duration and resource requirement can be made accordingly based on degree of similarity with completed projects.

14.3.3 Estimation by Available Resources

It is based on Parkinson's Law, which states that work expands to fill the time available. The cost is determined based on time and resources available for the project rather than by objective assessment. If time allowed for completion of the software is 12 months and 6 persons are available, the effort required is estimated to be 60 PMs.

14.3.4 Estimation by Software Price

The software cost is estimated based on price at which the software can be sold to the customer. The estimated effort depends on the specified budget and not on the software functionality.

In many cases, the costs of projects must be estimated using only incomplete user requirements. Getting the complete requirement specification and its analysis involves cost. Hence, an initial cost estimate for the system is needed to get the budget approved for developing detailed system requirements or the prototype.

Pricing to get the contract for software development is a common strategy used by software firms. This strategy is popularly called "**pricing to win**" strategy. A project cost is agreed on the basis of an outline proposal. Negotiations then take place between the client and the software developer to establish detailed project specification. This specification is constrained by the agreed cost. The buyer and seller must agree on what is acceptable system functionality. The fixed factor in many projects is not the project requirements but the cost. The scope or requirements may be reduced so that the cost is not exceeded.

For example, a software firm is bidding to get a contract to develop software for a certain client. The client gives only an outline description about the major functions that the software needs to perform but there is no detailed requirements specification available for this system. By looking at the size of company, outline description of functions required and market scenario, the software firm estimates that a price of Rs 6.5 lakhs is likely to be competitive. With this price the firm should be able to deliver software that fulfils the basic functionality. After the award of the contract, the detailed requirements of the system are determined and usually many additional requirements/features are discovered and negotiated. The contract is usually modified to include costs for these additional features.

This technique is popular because clients generally have a fixed budget for computerization/system development. Project cost estimates are often self-fulfilling. The estimate is used to define the project budget, and the product is adjusted so that the budget figure is realized.

Further, it is very difficult to give the exact specification of the required software in the beginning. The requirements are often amended/added during the middle of software development. Hence, clients generally award contract to firms they trust. Hence, in the software industry, contracts are usually awarded based on the reputation of software firms and their past performance. The estimated cost based on budget forms a base to start with. Extra cost for additional features is met from budget of the next period.

14.3.5 Estimation by Parametric Modeling

A model is developed to determine the relationships among the different project parameters by some suitable mathematical expressions. The parameters to be estimated (dependent parameters) are determined by substituting the value of the basic parameters (independent parameters) in the mathematical expression. In its simplest form, the parametric model takes the following mathematical expression:

$$e = c \times s^k$$

where

e is the dependent parameter to be estimated

s is the basic parameter of the software (independent variable)

c and k are constants based on characteristics of the software and project

'Software Size' expressed in terms of metrics such as Lines of Code (LOCs), Function Points (FPs), Object Points (OPs) etc. is the most important basic parameter (independent variable) used in the parametric model. The dependent parameter to be estimated could be cost of development, effort, project duration, resource requirement etc.

For example, if size of software is the independent variable and amount of effort needed to develop the software is the dependent variable to be estimated then the relationship between them can be expressed as under:

$$\text{effort} = c \times \text{size}^k$$

A parametric model can have multiple numbers of independent variables. A multivariable parametric estimation model takes the following mathematical expression:

$$e = c_1 s_1^{k_1} + c_2 s_2^{k_2} + \dots + c_n s_n^{k_n}$$

Where variables s_1, s_2, \dots, s_n are the basic parameters of the software. And c_1, c_2, \dots, c_n and k_1, k_2, \dots, k_n are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single-variable models, but it is difficult and costly to apply. Hence, requirement of accuracy in estimation and cost to be incurred for estimation are two important factors where trade-off should be made for selection of appropriate estimation technique.

14.3.6 Limitations of Estimation Techniques

There is no simple way to make an accurate estimate of the effort required to develop a software system. All the techniques of software estimation rely on experience-based judgments of project managers/

software professionals and past data of completed projects. However, there may be differences between a project being undertaken and the projects that were undertaken in the past. In the field of computer science, technological advancements are taking place continuously. Some of the new trends in software projects that are different from earlier projects are listed below:

- Preference for distributed object systems rather than mainframe-based systems
- Greater use of web services
- Greater use of Enterprise Resource Planning (ERP) or database-centered systems
- Greater use of off-the-shelf software rather than original system development
- Development with reuse rather than new development of all parts of a system
- Greater use of Computer Aided Software Engineering (CASE) tools and program generators rather than unsupported software development

If project managers/software professionals have not worked with these techniques, their previous experience may not be of help in estimating the software project costs.

There is difficulty in assessing the accuracy of cost estimation by every approach. Each estimation technique has its own strengths and weaknesses. Each uses different information about the project and the development team. Hence, it is advisable not to make a final estimation based on one single model. For large software projects, several cost estimation techniques should be used and their results should be compared. If there is significant variation in estimates made by different techniques, it indicates that there is insufficient information and/or understanding about the software project. In that case more information about the product, process or team should be obtained and estimation should be done afresh.

14.4 CLASSIFICATION OF SOFTWARE PROJECTS

The requirements of cost, effort and time for software projects depend on their development complexities. Boehm classified software development projects into three categories based on the development complexity.

These are:

1. Organic software
2. Semidetached software
3. Embedded software

The definitions of organic, semidetached and embedded systems are given below.

Organic: A development project can be considered of organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small and the team members are experienced in developing similar types of projects. The complexities involved in organic software projects are low.

Embedded: In contrast to organic software projects, the complexities involved in embedded software projects are high. A development project is considered to be of embedded type if the software being developed is strongly coupled to complex hardware or if regulations on the operational procedures are too stringent.

Semidetached: The complexities involved in semidetached software projects lie in between organic and embedded. A software project may be considered of semidetached type if some of the staff involved in the development are experienced and some are inexperienced. The team members may have limited experience on related systems but may not be unfamiliar with all aspects of the system being developed.

For classification of a software project, Boehm not only considered the characteristics of the product but also the characteristics of the development team and development environment.

As discussed earlier (Chapter 1, Section 1.1), software is generally classified as (1) Application software and (2) System software. Application software is meant to perform specific tasks and solve the problems of users. Data processing programs are examples of application software. Application software is generally classified as organic software. An operating system is system software. It controls the execution of programs, manages the storage and processing resources of the computer and drives other peripherals. The operating system interacts directly with the hardware and typically involves meeting timing constraints and concurrent processing. Hence, the operating system is an example of embedded software. Systems software can be of two types: (1) System development software and (2) Operating system software. System development software assists in the creation of application programs. Examples are language translators such as a BASIC interpreter and compilers such as FORTRAN, C/C++, Java etc. These are examples of semidetached software. Besides these, there is software for Engineering Applications, Medical Applications, Computer Aided Design (CAD)/Computer Aided Manufacturing (CAM) software, Animation software, Statistical packages, Simulation packages etc, which are not so simple and may also be classified as semidetached type.

14.5 CONSTRUCTIVE COST ESTIMATION MODEL

Constructive Cost Estimation Model (COCOMO) was proposed by Boehm in 1981 as a model for estimating effort, cost and schedule for software projects. Original COCOMO was later modified and published in 2000 as COCOMO II. Hence, the original model is now referred as COCOMO 81. In COCOMO 81, the estimation of software projects are done in three stages/models:

1. Basic COCOMO
2. Intermediate COCOMO
3. Complete COCOMO

14.5.1 Basic COCOMO

Basic COCOMO gives an approximate estimate of effort for software project based on a single attribute. The size of the software expressed in terms of Kilo Lines of Code (KLOCs) is the most important attribute for estimation of effort. According to basic COCOMO, the expressions for estimation of effort required for software development is given below:

$$\text{Effort} = A \times (\text{Size})^B$$

where

'A' and 'B' are constants. These values are based on whether the software product being developed is of organic, semidetached or embedded type.

Size of the software product is expressed in KLOCs

The effort estimation is expressed in units of PMs. For example, if times devoted for development of an application are 6 months, 4 months and 2 months, respectively, then total effort is 12 PMs. It may be noted that an effort of 12 PMs does not imply that 1 person should be employed for 12 months nor does it imply that 12 persons should work for 1 month. In fact if 12 persons are employed for the work, the work may take more than 1 month time to complete. Similarly, if 1 person is employed it is not necessary that work will be completed in 12 months. Time of completion of work does not have a

linear relationship with number of persons employed. Hence, the amount of effort required to complete a software project is not fixed. It may vary according to number of persons employed. The human resource employed should be just what is required, neither more nor less. There exists an optimal resource level where requirement of effort to do the job is least. The effort expressed in units of PMs denotes the volume of work estimated based on optimal number of persons employed. The relationship between software size and requirement of development effort is shown graphically in Figure 14.1.

Estimation of development time: The amount of effort estimated for software development is the main parameter for estimation of time. Expressions for estimation of time required for software development is given below:

$$T_{dev} = C \times (\text{Effort})^D$$

where

C and D are constants. Their values are based on whether the software product being developed is of organic, semidetached or embedded type.

Effort required to develop the software product is expressed in PMs and estimated time to develop the software, T_{dev} , is expressed in months.

Since effort is a function of software size, so also is the development time. The relationship between software size and development time required for the project is shown graphically in Figure 14.2.

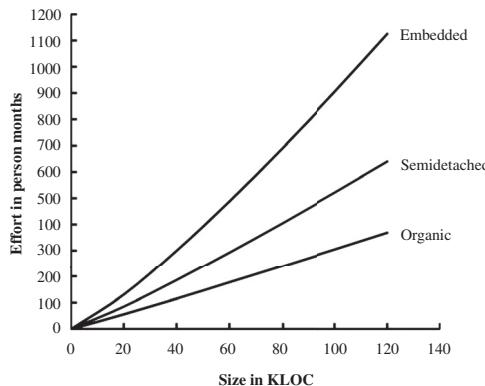


Figure 14.1 Requirement of effort versus Software size

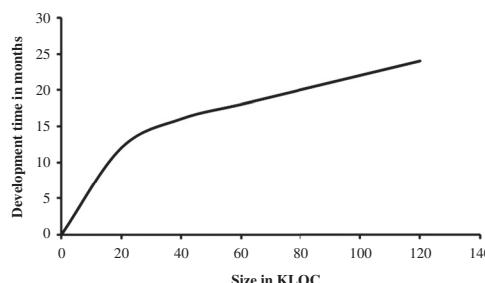


Figure 14.2 Development time versus Software size

Boehm examined the historical data collected from a large number of actual projects to determine the values of constants A, B, C and D for organic, semidetached and embedded software projects. These values for three classes of software projects are given below:

| Type of software | Values of constants | | | |
|------------------|---------------------|------|-----|------|
| | A | B | C | D |
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semidetached | 3.0 | 1.12 | 2.5 | 0.38 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.38 |

It may be noted that the relationship between requirement of effort and size of software is somewhat linear (value of exponent B is in the range 1.05 to 1.2). However, development time required for the software project does not bear a linear relationship with software size. It can be observed that the effort required to develop a product increases rapidly with project size whereas the development time increases only marginally. For example, when the size of the software project is doubled, the requirement of effort increases by about 120% whereas project duration is increased by about 30% only. This can be explained by the fact that for a big project, more number of activities can be done in parallel/concurrently. This reduces the time to complete the project. Further, it can be observed that the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached or embedded type. However, requirement of effort varies greatly depending on the type of the software project. For example, an embedded type software project may require about three times more effort than the effort required for organic type. The project cost is a function of effort. From effort estimation, the project cost can be obtained by multiplying estimated effort with unit cost of effort.

Example: The size of an organic type software project is 50 KLOC. If unit cost of effort is Rs.20,000/- per PM, estimate the development time and cost of the software project.

The formula for estimation of effort is given by:

$$\text{Effort} = A \times (\text{Size})^B$$

Software Size = 50 KLOC

$A = 2.4$ $B = 1.05$ (Values for the organic software project)

Hence, $\text{Effort} = 2.4 \times (50)^{1.05} = 146 \text{ PMs}$

$$T_{\text{dev}} = C \times (\text{Effort})^D$$

$C = 2.5$ $D = 0.38$ (Values for the organic software project)

Hence, $\text{Development time} = 2.5 \times (146)^{0.38} = 16.6 \text{ months}$

And Cost of software project = $146 \times 20,000 = \text{Rs. } 29,20,000/-$

14.5.2 Intermediate COCOMO

The purpose of basic COCOMO is to get a rough estimate of effort and time required for software development. It considers only one attribute of the software, i.e. software size expressed in KLOC. However, besides the software size, there are various other factors that affect the amount of effort and time duration needed for completion of the software project. The Intermediate COCOMO recognizes 15 factors

or cost drivers that affect the software project. These 15 factors can be categorized into four types. These are factors or attributes relating to:

1. Software Product 3. Human Resource
2. Computer System 4. Software Project

The 15 factors/cost drivers are listed in Table 14.1.

Table 14.1 Cost Drivers of Intermediate COCOMO

| Software Product Attributes | Human Resource Attributes |
|-------------------------------|---|
| 1. Reliability requirement | 8. Analyst capability |
| 2. Database size | 9. Virtual machine experience |
| 3. Product complexity | 10. Programmer capability |
| | 11. Programming language experience |
| | 12. Application experience |
| Computer System Attributes | Software Project Attributes |
| 4. Execution time constraints | 13. Use of modern programming practices |
| 5. Main storage constraints | 14. Use of software tools |
| 6. Virtual machine volatility | 15. Required development schedule |
| 7. Computer turnaround time | |

The software project is rated on each of these attributes (cost drivers) in a scale varying from very low to very high and extra high. The subjective (qualitative) rating is quantified based on numerical values assigned for the rating for each cost driver. The numerical value corresponding to subjective rating on a cost driver is called effort multiplier (EM). To account for a particular software project attribute/cost driver, the initial effort estimate made by basic COCOMO is refined (scaled up or down) by multiplying it with the corresponding EM. Values of the EM for different ratings for each of the 15 cost drivers as per COCOMO 81 are given in Table 14.2.

Table 14.2 Cost Driver Ratings and Corresponding Values of Effort Multipliers

| Cost Drivers (attributes) | Code | Ratings | | | | | |
|-------------------------------|------|----------|------|---------|------|-----------|------------|
| | | Very Low | Low | Nominal | High | Very High | Extra High |
| Product attributes | | | | | | | |
| 1. Reliability requirement | RELY | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 | |
| 2. Database size | DATA | | 0.94 | 1.00 | 1.08 | 1.16 | |
| 3. Product complexity | CPLX | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 |
| Hardware attributes | | | | | | | |
| 4. Execution time constraints | TIME | | | 1.00 | 1.11 | 1.30 | 1.66 |
| 5. Main storage constraints | STOR | | | 1.00 | 1.06 | 1.21 | 1.56 |
| 6. Virtual machine volatility | VIRT | | 0.87 | 1.00 | 1.15 | 1.30 | |
| 7. Computer turnaround time | TURN | | 0.87 | 1.00 | 1.07 | 1.15 | |

Personnel attributes

| | | | | | | |
|-------------------------------------|------|------|------|------|------|------|
| 8. Analyst capability | ACAP | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 |
| 9. Applications experience | AEXP | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 |
| 10. Programmer capability | PCAP | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 |
| 11. Virtual machine experience | VEXP | 1.21 | 1.10 | 1.00 | 0.90 | |
| 12. Programming language experience | LEXP | 1.14 | 1.07 | 1.00 | 0.95 | |

Project attributes

| | | | | | | |
|-----------------------------------|------|------|------|------|------|------|
| 13. Modern programming practices | MODP | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 |
| 14. Use of software tools | TOOL | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 |
| 15. Required development schedule | SCED | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 |

For example, if software tools (CASE tools) are used to a large extent in the development of software, the cost driver ‘use of software tools’ may be rated as high. The corresponding value of EM for high rating of this parameter is 0.91. Hence, the initial estimates are scaled downward accordingly. Similarly, if rating for the software product complexity is high, this initial estimate is scaled upward. To account for all the factors, the overall effort adjustment factor (EAF) is obtained by taking the product of the EMs of all relevant cost drivers.

$$EAF = \prod_{i=1}^{15} EM_i$$

$$i.e. EAF = (RELY \times DATA \times CPLX \times TIME \times STOR \times VIRT \times TURN \times ACAP \times AEXP \times PCAP \times VEXP \times LEXP \times MODP \times TOOL \times SCED)$$

The adjusted effort ($\text{Effort}_{\text{adj}}$) as per intermediate COCOMO is obtained by multiplying initial estimate of effort ($\text{Effort}_{\text{in}}$) with EAF.

$$\text{Effort}_{\text{adj}} = \text{Effort}_{\text{in}} \times EAF$$

If a certain cost driver (attribute) cannot be rated, its rating is assumed to be nominal (normal). It may be noted that for nominal rating on any cost driver, the value of EM is always 1. The typical value of EAF ranges from 0.9 to 1.4.

14.5.3 Complete COCOMO

Any large software system will consist of different components or subsystems. The complexities of these software components/subsystems may not be the same. For example, some subsystems may be too simple whereas some may be quite complex. Though the company may have done a similar project before, some of the system’s components may be quite novel. The development team members may have no prior experience in developing some of these components. Hence, some of the subsystems of a large system may be considered as organic, some may be considered as semidetached and some others may be considered as embedded type. The other attributes such as inherent complexity of subsystem, requirement of reliability, time constraint on schedule etc. may also be different.

For example, consider the case of software development for a University. The University has a number of affiliated colleges that are located at different places across the state. The University wants to have a computerized system to provide support for regulation of academics and conduct of examination in its affiliated colleges. The system may have the following subcomponents:

- The graphical user interface (GUI) through which different categories of users such as University officials, authorized officials and faculty members of affiliated colleges, students can access the system
- Database containing data relating to courses, academic records of students, faculties etc.
- An application subsystem for real-time data update

Of these, the GUI part may be considered as organic system whereas the application subsystem for real-time data update may be considered as embedded type. Depending on complexity, the database part could be organic or semidetached software.

Basic and intermediate COCOMOs consider a software product as a single homogeneous entity. However, complete COCOMO considers the system to be heterogeneous in nature. It considers that a computerized system consists of number of subsystems, and the attributes of each subsystem are different from the others. The complete COCOMO uses different EMs for each subsystem as well as phase of project.

To apply complete COCOMO, the software is broken down into subsystems. The subsystems are categorized in to organic, semidetached or embedded type. The subsystems are estimated separately and then added up to get the estimates of the complete project.

COCOMO also provides estimates of phase-wise distribution of effort. The phase-wise distribution of effort is very useful in project monitoring. Since cost of effort is different for different types of work, phase-wise distribution of effort helps in realistic cost estimation.

COCOMO 81 was based on study of software projects ranging in size from 2 KLOC to 100 KLOC done mostly in assembly language and third-generation procedural languages. The projects were based on the waterfall model, which was the prevalent software development process during that time. Hence, COCOMO 81 has almost become outdated and is succeeded by a new version called COCOMO II.

14.6 COCOMO II

COCOMO 81 was developed in 1981, based on computer technology, software development methodologies, and practices prevailing at that time. Since then lot of changes have taken place in the field of computer science. Hence, there was a need to develop a new COCOMO. Accordingly, the author of the earlier COCOMO, Barry Boehm, supported by others, developed a new COCOMO in 1998. This model was further revised and published in 2000. This new model is called COCOMO II. This model is better suited for estimating modern software development projects. COCOMO II has two main models for estimation of effort. These are:

1. Early design model
2. Post-architecture model

14.6.1 Early Design Model

The estimates in this level are made after the requirements have been agreed upon. The effort required for the software project is estimated by using the formula:

$$\text{Effort}_{NS} = A \times \text{Size}^E \times M$$

And the amount of development time, T_{dev} is estimated by the formula:

$$T_{dev} = C \times (\text{Effort})^F$$

where

$A = 2.5$ in initial calibration, Size in KLOC

B varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity

$C = 3.67$ and $D = 0.28$ in the initial calibration

The values of A, B, C and D may be calibrated based on local conditions and past data of the firm.

COCOMO II does not categorize the software project as organic, semidetached or embedded type. Instead it uses 'Scaling Factors (SFs)'. There are five different SFs as listed below:

1. *Precededness (PREC)*: Similarity with previously developed projects
2. *Process Flexibility (FLEX)*: Degree of flexibility in the development process
3. *Risk Resolution (RESL)*: Extent of risk analysis that needs to be carried out
4. *Team Cohesion (TEAM)*: Extent to which team members know each other and have worked together
5. *Process Maturity (PMAT)*: Capability maturity level of the organization

SFs are rated on a six-point scale from very low to extra high. The weight assigned to extra high rating for an SF is 0. Similarly, weights are assigned to different ratings for each SF as given in Table 14.3.

Table 14.3 SF Values, SF_j , for COCOMO II

| SFs | Very Low | Low | Nominal | High | Very High | Extra High |
|-----------------|----------|------|---------|------|-----------|------------|
| PREC (SF_1) | 6.20 | 4.96 | 3.72 | 2.48 | 1.24 | 0.00 |
| FLEX (SF_2) | 5.07 | 4.05 | 3.04 | 2.03 | 1.01 | 0.00 |
| RESL (SF_3) | 7.07 | 5.65 | 4.24 | 2.83 | 1.41 | 0.00 |
| TEAM (SF_4) | 5.48 | 4.38 | 3.29 | 2.19 | 1.10 | 0.00 |
| PMAT (SF_5) | 7.80 | 6.24 | 4.68 | 3.12 | 1.56 | 0.00 |

The value of E and F are calculated by using the following formula:

$$E = B + 0.01 \times \sum_j^5 SF_j$$

$$\text{and } F = D + 0.2 \times (E - B)$$

The value of ' M ' is determined as follows. The early design model has seven cost drivers/attributes, namely, (1) Product Reliability and Complexity (RCPX), (2) Developed for Reusability (RUSE), (3) Platform Difficulty (PDIF), (4) Personnel Capability (PERS), (5) Personnel Experience (PREX), (6) Schedule (SCED) and (7) Support Facilities (FCIL). These attributes are rated using a six-point scale from very low to extra high.

Each rating level of every cost driver has a value, called an EM , associated with it. The value assigned to a cost driver's nominal rating is 1.00. If rating level of a cost driver is higher than nominal then value of the corresponding EM is more than 1.0. Conversely, if the rating level is lower than nominal, the value of EM is less than 1.0.

The value of M is computed as:

$$M = (PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED)$$

14.6.2 Post-architecture Model

This model is a detailed model. It is intended to be used when the software life-cycle architecture has been developed. The formula used for effort estimation is same as that used in the early design model. However, instead of seven, the post-architecture model uses 17 cost drivers as listed below:

Product Attributes

1. Required system reliability (RELY)
2. Complexity of the system (CPLX)
3. Documentation requirement (DOCU)
4. Size of database (DATA)
5. Requirement for reuse (RUSE)

Computer Attributes

6. Execution time constraint (TIME)
7. Development platform volatility (PVOL)
8. Memory constraints (STOR)

HR Attributes

9. Capability of project analysts (ACAP)
10. Personnel continuity (PCON)
11. Programmer capability (PCAP)
12. Programmer domain experience (PEXP)
13. Analyst domain experience (AEXP)
14. Language and tool experience (LTEX)

Project Attributes

15. Use of software tools (TOOL)
16. Schedule compression (SCED)
17. Extent of multisite working (SITE)

The value of an EM is 1.0 for nominal rating of each cost driver. For lower ratings, the value of EM is less than 1.0 and for higher rating, it is more than 1.0. The post-architecture model of COCOMO II has 17 cost drivers whereas COCOMO 81 has 15 cost drivers. The 17 cost drivers of post-architecture model can also be related to the seven cost drivers of the early design model as shown in Table 14.4.

Table 14.4 Early Design and Post-Architecture Cost Drivers

| Early Design Model | Post-Architecture Model |
|--------------------|-------------------------|
| PERS | ACAP, PCAP, PCON |
| RCPX | RELY, DATA, CPLX, DOCU |
| RUSE | RUSE |
| PDIF | TIME, STOR, PVOL |
| PREX | AEXP, PLEX, LTEX |
| FCIL | TOOL, SITE |
| SCED | SCED |

In the formula used in COCOMO, the size is expressed in KLOC. The software size is a very important parameter for estimation of the software project. In modern software projects, the size is usually determined by FPs. COCOMO II provides means to convert FPs into equivalent KLOC of program in any given programming language. For determining LOCs, it also considers the factors relating to adapted codes, reused codes and automatically translated code. It can also be used to determine phase/activity distributions of effort and schedule for various lifecycle models.

More details about COCOMO are beyond the scope of this book. Interested readers may refer to original books in which COCOMO was published. The details of these books are given below:

1. Barry Boehm, 'Software Engineering Economics', Englewood Cliffs, NJ:Prentice-Hall, 1981, ISBN 0-13-822122-7
2. Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece, 'Software cost estimation with COCOMO II', Englewood Cliffs, NJ:Prentice-Hall, 2000. ISBN 0-13-026692-2

The reference manuals, software and more details about COCOMO can also be obtained from the website:

http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

14.7 CONCLUSION

Proper planning is necessary for successful execution of a project. Plan is a commitment for future action. Commitments to unrealistic time and resource estimates result in delays, poor quality work, budget overrun, customer dissatisfaction and consequently project failure. Proper planning requires accurate estimates.

The following steps are usually followed for software project estimation:

Determine size of the product. The size of software is usually determined by FPs. FPs are then converted into equivalent LOCs.

The next step is to estimate various parameters of the software/software project. This is usually done based on subjective assessment by technical experts or people having experience in the relevant fields.

The development effort needed to complete the project is determined from estimates of size and other parameters. Estimation is also done for phase-wise and activity-wise distribution of effort. COCOMO can be used for this purpose.

The development time needed to complete the project is determined from estimates of effort and other parameters.

Software cost is a function of effort. Rough estimate of cost can be done based on estimate of total effort required for the project. A software project requires effort of different people. The major cost in a software project is the cost of human resource. Other costs are administration cost, traveling cost, cost of utilities etc. Since the unit cost of effort is different for different activities, phase-wise and activity-wise distribution of effort is used for more accurate estimation of cost.

The average requirement of resources can be determined by dividing total estimated effort with estimated development time.

The steps are shown through block diagram in Figure 14.3.

Human resource is the most important resource in a software project. However, it may be noted that requirements of human resource are not constant over the entire duration of the project. Usually it is less during the initial stages as well as the final stages of the project. Similarly, skill-set of people in different stages of the project will be different. Since it is difficult to hire and lay off people as and when required, one of the important objectives of planning is to ensure that all people have equitable workload and also the workload is spread equally, over the period of time. Generally, the software firms undertake a number of projects at a time. People are hired for a given capacity. The activities of the projects are planned for a certain level of resource. There is also transfer of people across projects as well as across activities. That

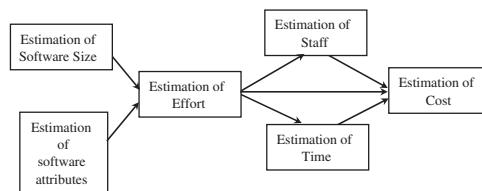


Figure 14.3 Step of software project estimation

is the reason why software firms prefer people having multiple skills. All these aspects come under the purview of different areas of management such as capacity planning, resource leveling, portfolio management, HR planning etc.

SUMMARY

Estimation of various project parameters is the first step of project planning. The cost of development, effort, human resource and time are important parameters of software projects.

The top-down approach and the bottom-up approach are two approaches to project estimation. Estimation by Expert Judgement, Estimation by Analogy, Estimation by Available Resources, Estimation by Software Price and Estimation by Parametric Modeling are some important techniques of software project estimation.

The software projects can be classified into three categories based on the development complexity: (1) Organic, (2) Semidetached and (3) Embedded. A software project is considered as organic if the complexities are low, whereas if the complexities are high involving software–hardware integration, it is considered as an embedded software project. The complexities involved in semidetached software projects lie in between organic and embedded.

COCOMO is a popular parametric model developed by Boehm. This model was first developed in 1981 and is referred to as COCOMO 81. In COCOMO 81, the estimation of software projects are done in three stages/models: (1) Basic COCOMO, (2) Intermediate COCOMO and (3) Complete COCOMO. Basic COCOMO gives an approximate estimate of effort for a software project based on a single attribute, i.e. size of the software. Hence, it gives only a rough estimate. Intermediate COCOMO recognizes 15 additional factors for software project estimation. These 15 factors are termed as cost drivers. They are categorized into four types relating to: (1) Software Product, (2) Computer System, (3) Human Resource and (4) Software Project. In Complete COCOMO, the software is broken down into subsystems and each subsystem is ranked individually as organic, semidetached or embedded type. The subsystem is estimated separately and then added up to give total estimates of the project.

COCOMO II is the revised version of COCOMO 81. This model is better suited for estimating modern software development projects. COCOMO II has two main models: (1) Early design model and (2) Post-architecture model. COCOMO II does not categorize the software project as organic, semidetached or embedded type. Instead it uses five different ‘SFs’. These are: (1) Precedentedness (PREC), (2) Process Flexibility (FLEX), (3) Risk Resolution (RESL), (4) Team Cohesion (TEAM) and (5) Process Maturity (PMAT). The early design model uses seven cost drivers for adjustment of effort. The post-architecture model is the detailed model. It uses 17 cost drivers. COCOMO II provides means to convert FP into equivalent KLOC of program in any given programming language.

Software cost is a function of effort. The major cost in a software project is the cost of human resource. The average requirement of resources can be determined by dividing total estimated effort by estimated development time.

Generally, software firms undertake a number of projects at a time. People are hired for a given capacity. The activities of the projects are planned for a certain level of resource. There is also transfer of people across projects as well as across activities. All these aspects come under the purview of different areas of management such as capacity planning, resource leveling, portfolio management, HR planning etc.

EXERCISES

1. Explain the importance of software estimation.
2. Enumerate various techniques of software estimation. Should more than one technique be used for estimation? Explain.
3. Describe the Delphi method of software estimation.
4. Compare estimation by expert judgement technique with parametric modeling technique.
5. Write down the differences between organic, semidetached and embedded software product.
6. Differentiate among basic COCOMO, intermediate COCOMO and complete COCOMO 81.
7. What are the various types of cost drivers in COCOMO 81?
8. As the manager of a software project to develop a product for business application, if you estimate the effort required for completion of the project to be 50 PMs, can you complete the project by employing 50 engineers for a period of 1 month? Justify your answer.
9. For the same number of LOCsand the same development team size, rank the following software projects in order of their estimated development time. Show reasons behind your answer.
 - a. A text editor
 - b. An employee pay roll system
 - c. An operating system for a new computer
10. Write the advantages of COCOMO II over COCOMO 81.
11. What are the various SFs in COCOMO II?
12. What are the different types of cost drivers in post-architecture COCOMO II? Are they different from those of COCOMO 81?
13. Suggest the steps for carrying out software project estimation.

This page is intentionally left blank.

SOFTWARE PROJECT MANAGEMENT

Development of software is a project. The tools, techniques and principles of 'Project Management' are applicable in the development of software. This chapter discusses different aspects of software projects, their objectives, constraints and responsibilities, types of project plans and steps for the software development plan. This chapter describes the different tools and techniques of project planning, such as Work Breakdown Structure (WBS), Activity Network, Critical Path Method (CPM) and Program Evaluation and Review Technique (PERT), and Gantt chart. It also covers Risk management and Configuration management.

Development of any commercial software requires great amount of effort, time and money. It has a definite beginning and end. It also requires wide varieties of skill and interrelated activities. Hence, development of software is a project. The tools, techniques and principles of 'Project Management' are useful in the development of software.

15.1 INTRODUCTION TO SOFTWARE PROJECT MANAGEMENT

Project management is the discipline of planning, organizing and managing the resources (e.g. people) to complete a project within defined scope, quality, time and cost constraints. A project is a temporary and one-time endeavor undertaken to create a unique product or service, which brings about beneficial change or added value. This property of being a temporary and one-time undertaking differentiates a project from an ongoing operation.

Project Management as defined by the Project Management Institute (PMI), in the guide to Project Management Body of Knowledge (PMBOK), is given below:

'Project management is the application of knowledge, skills, tools and techniques to project activities to meet project requirements.'

The PMBOK lists nine knowledge areas of project management as given below:

1. Integration Management
2. Scope Management
3. Time Management
4. Cost Management
5. Quality Management
6. Human Resource Management
7. Communication Management
8. Risk Management
9. Procurement Management

15.1.1 Salient Features of a Software Project

The salient features of software development have been discussed in Section 1.8 of the first chapter. Software project management is application of project management to software projects. Software projects have several properties that make them different from other kinds of engineering projects.

The software product is intangible in nature. Unlike an engineering or construction project, the progress made in software project is not visible to the eyes. Thus, software projects pose greater need and complexity for monitoring.

Software engineering is a new discipline. Hence, we do not have fixed and proven methodology to undertake large-scale software projects.

Large software projects are often made to order. Due to uniqueness of software projects, there is little scope for duplicating the experience gained from one project in other projects.

Computer technology is changing very fast. Hence, technology used in a new project is different from the previous ones.

Objectives and Constraints: The projects need to be performed and delivered within three main constraints, namely **scope**, **time** and **cost**. Time constraint refers to the amount of time available and cost constraint refers to the budgeted amount available to complete a project. Scope constraint refers to what must be done to produce the project's end result. These three constraints are also the important parameters for assessing the performance of project management. These three parameters are related to each other and can be represented by three sides of a triangle as shown in Figure 15.1. Just as one side of the triangle cannot be changed without impacting the others, change in any one parameter will affect the other two. Increased scope typically means increased time and increased cost. If the allowed time

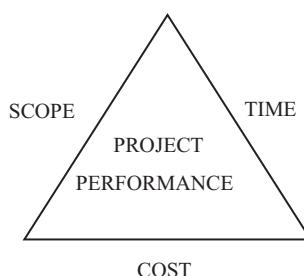


Figure 15.1 Three parameters of software project management

duration is reduced, it would require extra cost to expedite the project and/or reduce the scope of the project. Similarly, increase in scope would require more time and extra cost.

The main objective of project management is to meet the scope of the project at minimum cost and time.

15.1.2 Responsibilities of a Software Project Manager

The project manager has the overall responsibility for successfully completing the software project. He is often a client representative and has to determine and implement the exact needs of the client. His job responsibility ranges from invisible activities like 'building up team morale' to highly visible activities like 'customer presentations'. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management (SCM), risk management, interfacing with clients, managerial report writing and presentations etc. These activities are certainly numerous, varied and difficult to enumerate, but can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with a view to ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

15.1.3 Qualities of the Software Project Manager

A software project manager must know about different project management techniques such as cost estimation, risk management, configuration management (CM), quality management etc. However, some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations and team building are largely acquired through experience. In addition to this, a project manager should possess good qualitative judgment and decision-taking capabilities. He leads a group of people to implement projects. His main job is to coordinate the work of others and to get work done. Hence, he should also have good communication and behavioral skills. His ability to adapt to the procedures of the contracting party, and to form close links with their nominated representatives, helps to realize the key issues of time, cost, quality and client satisfaction.

15.2 PROJECT PLANNING

Planning is the first step for any systematic action. It precedes execution of the project. Planning is a commitment for future actions towards achieving some specified objectives. The development of software project plans is based on some chosen Life Cycle model. The plan includes estimation of costs, time and resource. It involves specifying the procedures and arranging the activities to be performed for a project and assigning time and resources for each activity. It also specifies milestones to help review the actual progress of work. The review of project plans at periodic intervals allows the management to see the current state of software development and to take appropriate corrective actions. Monitoring the project against plan throughout the development process reduces the risk of facing unpleasant surprises in the later stages.

Developing a realistic project plan is essential to gain an understanding of the resources required, and how these should be used. The work for software development is carried out based on the project plan. The project plan forms the basis for signing of software development contracts.

Table 15.1 Types of Software Project Plan

| | |
|------------------------------------|---|
| i) Software development plan | The central plan, which describes how the system will be developed |
| ii) Testing and validation plan | Defines how the software will be tested during development and how it will be validated on completion by the client |
| iii) Quality assurance plan | Specifies the quality procedures and standards to be used |
| iv) Configuration Management plan | Defines how changes will be managed for consistency to ensure configuration and installation of correct version of software |
| v) Maintenance plan | Defines how the system will be maintained |
| vi) HR plan/Staff development plan | Describes how the number of people required for different projects will be determined and how the skills of existing HR will be developed |

15.2.1 Types of Plans

Software projects require many types of plans. Different types of software project plans are listed in Table 15.1.

15.2.2 Software Development Plan

Software development plan forms the basis for execution of a software project. This is the central plan. Unless specifically mentioned, the project plan usually means software development plan. It specifies the following details:

- What are activities that have to be carried out?
- For each activity, specify who should carry out that activity.
- For each activity, specify what resource will be needed.
- Specify the time duration for each activity.
- Specify the schedule or timetable for carrying out these activities.
- Identify and estimate possible risks and spell out contingency actions for risk abatement.
- Specify how progress made during execution will be accessed and monitored.

Developing a project plan is as important as software design and coding. Project planning for software development consists of the following essential activities:

i) Estimate Attributes of the Project: The correct estimations of project attributes determine the effectiveness of planning. Some important attributes of any project are given below.

Project size: What will be the complexity and size in terms of effort and time required to develop the software product?

Cost: How much will it cost to develop the project?

Effort: How much effort would be required?

Duration: How long will it take to complete development?

ii) Make Work Breakdown Structure (WBS): The project is broken down successively into a number of smaller and manageable work components and the work components are arranged in a tree-like hierarchical structure. This is called 'WBS' of the project.

- iii) **Schedule Project Work Component:** Determine the schedule for carrying out each work component specified in the WBS of the project. It also involves allocating work for human resources, i.e. who will do what and when.
- iv) **Determine hardware and software resources:** Determine what hardware and software resources will be required for carrying out each work component of the project and project as a whole.
- v) **Risk Management Plan:** Identify, estimate and analyze various risks associated with the project and plan for such contingencies.
- vi) **Project Monitoring and Control Plan:** Specify how the expenditure incurred and progress made in the project will be accessed and monitored. Also, identify various check points for monitoring of the project.

15.3 WORK BREAKDOWN STRUCTURE

Work Breakdown Structure (WBS) is a hierarchical depiction of all components of work required to accomplish the entire work of the project. A complex project is made manageable by breaking it down into individual components and organizing the work components in a hierarchical structure. Such a structure defines the logical relationships between work components and the total work content of the project.

In the first level, the work of project is decomposed into few major components. Then each of these level-1 work components is broken down into more second-level work components. The process is continued till the work components are small enough to facilitate resource allocation, assignment of responsibilities, measurement and control. Each descending level represents an increasingly detailed division of a project component. WBS is used throughout the life cycle of a project to identify, assign and track a specific work component.

WBS coding scheme: WBS can be illustrated in a block diagram as shown in Figure 15.2. The WBS elements are usually numbered sequentially to depict the hierarchical structure. The higher-level work components are generally performed by groups. The lowest level in the hierarchy often comprises tasks performed by individuals.

Terminal element: A terminal element is the lowest element in a WBS. It is not subdivided any further.

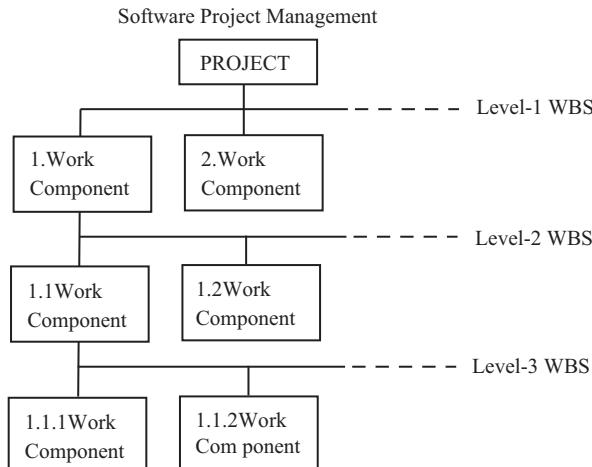


Figure 15.2 Block diagram of work breakdown structure

Terminology for Different Levels: Organizations use their own terminology for classifying WBS components according to their level in the hierarchy. For example, some organizations refer to different levels as tasks, subtasks and work packages. Others use the terms phases, entries and activities or tasks. In our book we shall refer to terminal elements as activities.

Level of Detail: The breaking down of a project into its component parts facilitates resource allocation and the assignment of individual responsibilities. Care should be taken to use a proper level of detail when creating the WBS. A very high level of detail is likely to result in a large number of activities. On the other extreme, if the size of activities is too big, it is difficult to manage effectively. The project is broken down so that the duration of activities is between few days to a few weeks. This facilitates the execution in most cases. Activities are the smallest elements of WBS that are estimated in terms of resource requirements, budget and duration.

The 100% Rule: WBS should include all the work (100%) defined by the project scope. Also, it should capture all deliverables required for completion of the project.

To illustrate the application of WBS, let us take an example of a simple software development project. The entire work of the project can be broken down into few major level-1 WBS components as shown in Figure 15.3.

Each of the level-1 WBS components can be further decomposed into level-2 work components. Level-2 WBS of software project is shown in Table 15.2.

The level-2 WBS components can be further decomposed into level-3 work components and so on. For example, the work component 'testing' is further decomposed into level-3 work components as shown in Table 15.3.

WBS can be of three types:

1. Activity-based WBS
2. Product-based WBS
3. Hybrid WBS

In activity-based WBS, the entire work involved in the project is decomposed into smaller work elements. An example of an activity-based WBS is given in Table 15.2 and Table 15.3.

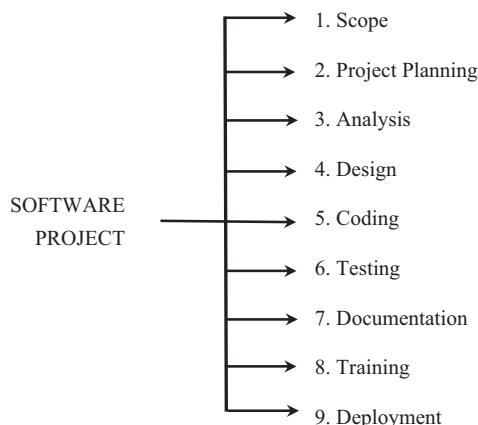


Figure 15.3 Level-1 WBS of software project

Table 15.2 Level-2 WBS of Software Project

| | |
|-------------------------------------|--------------------------------------|
| 1. Scope | 5.3 Construct database |
| 1.1 Determine project scope | 6. Testing |
| 1.2 Define preliminary resources | 6.1 Unit Testing |
| 1.3 Secure core resources | 6.2 Integration Testing |
| 2. Project Planning | 6.3 Pilot Testing |
| 2.1 Develop budget | 7. Documentation |
| 2.2 Develop schedule | 7.1 Help System |
| 2.3 Secure required resources | 7.2 User Manual |
| 3. Analysis | 8. Training |
| 3.1 Conduct needs analysis | 8.1 Develop training specifications |
| 3.2 Draft software specifications | 8.2 Identify training methodology |
| 3.3 Review software specifications | 8.3 Develop training materials |
| 3.4 Feedback on specification | 8.4 Conduct training usability study |
| 4. Design | 8.5 Finalize training materials |
| 4.1 Review software specifications | 8.6 Impart training |
| 4.2 Develop prototype | 9. Deployment |
| 4.3 Incorporate feedback into spec. | 9.1 Determine deployment strategy |
| 4.4 Module structure design | 9.2 Secure deployment resources |
| 4.5 Detailed module design | 9.3 Train support staff |
| 4.6 Design Interfaces | 9.4 Deploy software |
| 4.7 Design Database Structure | 9.5 Post Implementation Review |
| 5. Coding | |
| 5.1 Develop code | |
| 5.2 Debug Code | |

Table 15.3 Level-2 and level-3 WBS of a work component

| | | |
|--|--|------------------------------------|
| 6.1 Unit Testing | 6.2 Integration Testing | 6.3 Pilot Testing |
| 6.1.1 Develop unit test plans | 6.2.1 Develop integration test plans | 6.3.1 Identify test group |
| 6.1.2 Review modular code | 6.2.2 Test module integration | 6.3.2 Install/deploy software |
| 6.1.3 Test component modules | 6.2.3 Identify anomalies to specifications | 6.3.3 Obtain user feedback |
| 6.1.4 Identify anomalies to specifications | 6.2.4 Modify code | 6.3.4 Evaluate testing information |
| 6.1.5 Modify code | 6.2.5 Re-test modified code | |
| 6.1.6 Re-test modified code | | |

In product-based WBS, the software product is broken down into constituent work products. Product-based WBS of software (also called Product Breakdown Structure) specifies the work products or deliverables that go into developing the software.

For example, level-1 product-based WBS of software may consist of following work products:

1. Scope Statement
2. Project Plan

3. Analysis Document
4. Design Specification
5. Program Codes
6. Tested Software
7. Documentation
8. Training programs
9. Deployed Software

Similarly each level-1 work product can be further broken down into level-2 work products. For example, project plan will comprise following plans:

- 2.1 Software development plan
- 2.2 Testing and Validation plan
- 2.3 Quality assurance plan
- 2.4 CM plan
- 2.5 Maintenance plan
- 2.6 Human resource plan

Product-based WBS focuses on output of work called deliverables or the work product. The proponents of this approach claim that product-based WBS gives better control over the project. On many occasions, a hybrid approach that constitutes a combination of activity-based approach as well as product-based approach is used for making WBS.

Identification of dependencies between activities and estimation of activity durations are done after the development of WBS. Hence, WBS is the foundation of project planning.

15.4 PROJECT SCHEDULING

Project-task scheduling is an important project planning activity. It involves deciding which work would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all major work needed to complete the project.
2. Break down a large work component into a number of smaller activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

Then, a master schedule of activities is prepared. The master schedule is based on precedence relationship between activities and estimate of their duration.

15.4.1 Activity Network

After the creation of WBS, the next step is to find the dependency among activities. For example, the activity testing of program code cannot be undertaken unless coding activity is completed. Similarly,

coding activity cannot be done before design. Precedence Analysis involves reviewing activities and determining dependencies. There can be three types of dependencies:

- (i) Mandatory dependencies: These are inherent in the nature of the work, e.g. testing can be done only after coding.
- (ii) Discretionary dependencies: These are defined by the project team, e.g. the project team may decide to wait for feedback on prototype before detail design.
- (iii) External dependencies: These involve relationships between project and non-project activities, e.g. certain activities may have to wait till certain hardware is received from supplier.

Dependency among the different activities determines the order in which the different activities would be carried out.

An activity may be independent if it has no preceding activity. However, most activities have proceeding activities. These activities cannot be completed unless the proceeding activities have been completed. These constraints determine the sequence in which the activities have to be carried out to complete the project. The sequential relationship among various activities can be effectively represented by activity network diagram. For illustration purposes, the activities of a small project, their respective duration and requirements of preceding activities are shown in Table 15.4.

In the said project, activities C and D cannot be started unless activity A is completed. Similarly, activity E cannot be started unless activity B is completed and so on. Different activities making up a project, their estimated durations and interdependencies can be effectively represented by the activity network diagram. There are two ways of showing activities and their relationships. These are:

1. Activity on Nodes (AON)
2. Activity on Arrows (AOA)

In AON network diagram, activities are represented by circles or ovals and the sequence in which the activities have to be carried out is shown by 'arrows'.

For the project activities given in Table 15.4, the AON network diagram is shown in Figure 15.4. The circles representing the activities are labeled as A, B, C and so on. The duration of an activity is shown below the labels. The arrows show the sequence in which the activities need to be executed.

The activities of the above project can also be represented by the AOA network diagram as shown in Figure 15.5.

Table 15.4 Activities of a Project

| Activity | Estimated duration in days | Preceding activities |
|----------|----------------------------|----------------------|
| A | 15 | — |
| B | 25 | — |
| C | 12 | A |
| D | 20 | A |
| E | 22 | B |
| F | 16 | C |
| G | 21 | D, F |

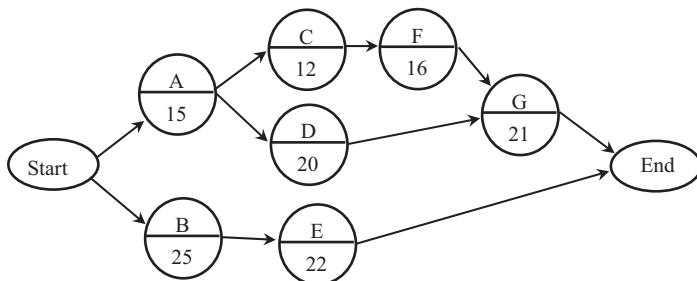


Figure 15.4 Network diagram showing activities on nodes

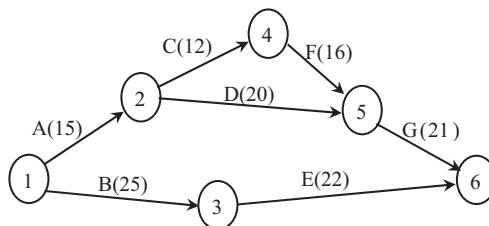


Figure 15.5 Network diagram showing activities on arrow

In the AOA network diagram, activities are represented by arrows. The nodes are the events that are either the beginning or end of activities. The nodes are numbered serially and the activities are identified by their start and end node numbers.

Sometimes imaginary or artificial dummy activities are included in the AOA network diagram to represent precedence relationship between activities. To illustrate the use of dummy activities, let us consider the relationship between activities as given below.

| Activity | Predecessor activity |
|----------|----------------------|
| A | — |
| B | — |
| C | A, B |
| D | B |

The AON network diagram to show the relationship between above activities is very easy to draw and is shown in Figure 15.6.

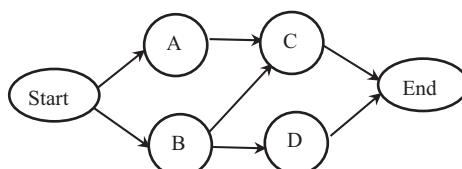


Figure 15.6 AON network diagram

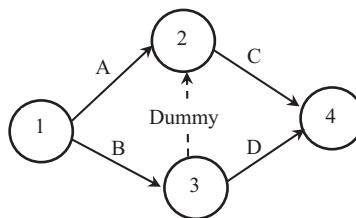


Figure 15.7 AOA network diagram showing dummy activity

However, representation of above activity relationship through the AOA diagram will require inclusion of dummy activity as shown in Figure 15.7.

It may be noted that activity C is done after A and B, whereas activity D is done only after B. Hence, to distinguish activity C from D, a dummy activity starting from node 3 to 2 is included in the network diagram. A dummy activity is an imaginary activity that does not require any resource, cost and time. The AOA approach often requires inclusion of dummy activities. However, the AOA approach has one advantage. In the AOA approach, the activities can be specified as a two-dimensional array $A_{i,j}$, where i is the start node and j is the end node. Hence, there is no need to separately mention the predecessor activities. Thus, AOA network diagrams are sometimes more suitable for implementing in a computer.

The network diagram shows the dependency relationship between activities. There can be four types of dependency between activities.

- i. **Finish-to-start (FS):** Activity B cannot start until activity A is completed
- ii. **Start-to-start (SS):** Activity B cannot start until activity A starts
- iii. **Finish-to-Finish (FF):** Activity B cannot finish until activity A finishes
- iv. **Start-to-finish (SF):** Activity B cannot finish until activity A starts

These four types of dependencies between activities are shown in Figure 15.8.

FS is the most common type of dependency. The network diagrams shown earlier are all based on 'FS' dependencies.

To complete a project early, effort is made wherever possible to execute activities in parallel or concurrently. The network diagram helps to identify parallel activities in a project. It facilitates better

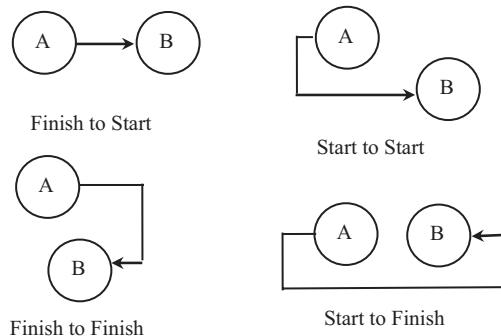


Figure 15.8 Four types of dependency between activities

understanding of relationship among various activities of a project and aids in planning, scheduling and monitoring the progress of the project.

15.4.2 Critical Path Method (CPM) and Program Evaluation and Review Technique (PERT)

Consider the simple project given earlier in Table 15.4 and the AON network diagram shown in Figure 15.4. Various paths from start to end of activity network, and the total duration of activities for each path are given below:

| PATH | DURATION |
|-----------------------------|-------------------------------|
| Start – A – C – F – G – End | $15 + 12 + 16 + 21 = 64$ days |
| Start – A – D – G – End | $15 + 20 + 21 = 56$ days |
| Start – B – E – End | $25 + 22 = 47$ days |

From the above, we find that the sum of duration of all the activities of path A-C-F-G is 64. It has the longest duration as compared to other paths. Hence, the required duration for completing the project is 64 days. The path A-C-F-G is called the critical path and activities A, C, F and G are the critical activities. If there is any delay in carrying out these activities, the entire project is delayed. However, other activities are not that critical. For example, there can be a delay of 8 days in completing activity D. Similarly, activities B and E can also be delayed by total of 14 days without affecting the completion time of the project. Thus, comparatively more attention and resources may be given to critical activities (A, C, F and G) than to non-critical activities (B, D and E). This technique for analyzing various activities of a project, for monitoring and allocation of resources is called CPM.

Let us consider another example. The activities, their duration and activity dependency are shown in Table 15.5.

The steps of CPM are as under:

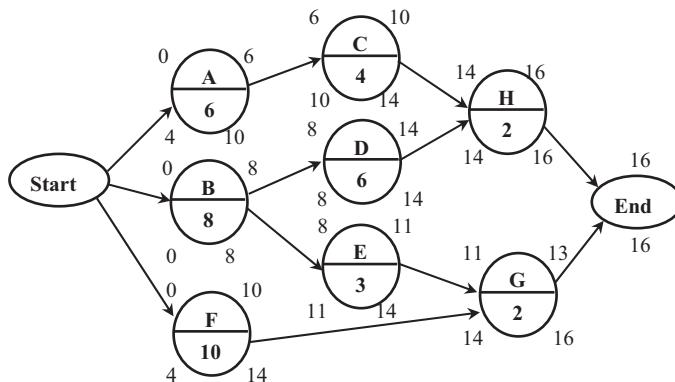
Find Earliest Start Time (EST) and Earliest Finish Time (EFT) of each activity. EST of an activity is the highest value of the EFTs of all preceding activities. If an activity has no preceding activity, then its EST is zero. EFT of an activity is the sum of EST and duration of activity. EST and EFT of activities are determined in a chronological order from the start activities to the end activities. This is called **forward pass**. EFT of the last activity determines the project duration.

Table 15.5 Activities of a Project

| Activity | Description | Duration | Predecessor activity |
|----------|-------------------------|----------|----------------------|
| A | Hardware selection | 6 | – |
| B | Software design | 8 | – |
| C | Install hardware | 4 | A |
| D | Code and test software | 6 | B |
| E | File take-on | 3 | B |
| F | Write user manuals | 10 | – |
| G | User training | 2 | E, F |
| H | Install and test system | 2 | C, D |

Table 15.6 Start and Finish Time of Activities

| Activity | Duration in weeks | Predecessor activity | EST | EFT | LST | LFT | Slack/Float |
|----------|-------------------|----------------------|-----|-----|-----|-----|-------------|
| A | 6 | — | 0 | 6 | 4 | 10 | 4 |
| B | 8 | — | 0 | 8 | 0 | 8 | 0 |
| C | 4 | A | 6 | 10 | 10 | 14 | 4 |
| D | 6 | B | 8 | 14 | 8 | 14 | 0 |
| E | 3 | B | 8 | 11 | 11 | 14 | 3 |
| F | 10 | — | 0 | 10 | 4 | 14 | 4 |
| G | 2 | E, F | 11 | 13 | 14 | 16 | 3 |
| H | 2 | C, D | 14 | 16 | 14 | 16 | 0 |

**Figure 15.9** Activity network showing EST, EFT, LST and LFT

Next, find Latest Finish Time (LFT) and Latest Start Time (LST) of each activity. LFT of an activity is the lowest value of LSTs of all its succeeding activities. LFT of an activity that has no succeeding activity is the same as finish time of the project (i.e. the EFT of last activity). LST of an activity is the difference between its LFT and duration. LFT and LST are determined starting from the last activity to the first activity. This is called **backward pass**.

EST, EFT, LST and LFT of activities are shown in Table 15.6 and network diagram in Figure 15.9.

Please note that numbers written on four corners of each activity denote EST, EFT, LST and LFT, respectively. The difference between LST and EST (or between LFT and EFT) determines how much the activity can be delayed. This is called slack or float. The float or margin for delay is zero for activities whose LST and EST are same. Hence, these activities are the critical activities and path (line) joining these activities from start to end is the critical path. Hence, 'Start-B-D-H-End' is the critical path.

CPM is a deterministic model because it assumes that activity duration is certain and is estimated correctly. Hence, CPM is a suitable method for analyzing the project where there is some amount of certainty in execution of the activities.

However, in many situations, execution of activities is affected by conditions that are quite uncertain. In an uncertain environment, duration and funds requirement of activities cannot be exactly estimated. In such cases, another technique called PERT is used. PERT is based on probability concepts.

In PERT three time estimates are made for every activity, as follows:

- Optimistic time (t_o): It is the estimated time for completing an activity if all conditions while doing the activity turn out to be favorable. Hence, it can be considered as the shortest time in which it is practically possible to complete the activity.
- Most likely time (t_m): It is the estimated completion time having the highest probability. Note that this time is different from the *expected time*.
- Pessimistic time (t_p): It is the estimated time for completing an activity if all conditions while doing the activity turn out to be unfavorable. It is the longest time that an activity might require.

PERT assumes a beta probability distribution for the time estimates. For a beta distribution, the expected time (t_e) for each activity can be approximated using the following weighted average:

$$t_e = \frac{t_o + 4t_m + t_p}{6}$$

And the standard deviation (σ) for each activity completion time is given by:

$$\sigma = \frac{t_p - t_o}{6}$$

In PERT, the probable duration of the project is estimated based on expected duration and standard deviation in expected duration of activities. The expected durations of activities are also shown on the network diagram.

15.5 EXECUTION, MONITORING AND CONTROL

A good plan determines successful execution. The success of a software project much depends on its project plan (software development plan). The plans are usually made under various constraints such as time, cost, scope, resources etc. Sometimes the software project is required to be done in less than its normal duration. This is called project crashing. It is done by allocating more resources at higher cost. Sometimes there is a constraint of resources, which limits the number of activities to be executed simultaneously. The other issue is resource leveling. The activities should be so scheduled that the amount of work is uniform over time. The work should also be equitably distributed among human resources. Project planning is not one shot activities. Changes take place during execution. Hence, the project plan generally needs to be revised or updated from time to time.

Uncertainties are generally inherent in projects. Changes in the project environment cause plans to be updated when:

- there are additional requirements and changes that are necessary but were not foreseen at the early stages;
- there is work that was not firmly estimated earlier due to unavailability of necessary information;
- productivity changes and market fluctuations cause variations from the data used to produce early estimates and
- there are slippages in some work due to various assignable and random causes and these affect the schedule of subsequent work.

Hence, proper monitoring of activities is necessary for corrective action. Monitoring is collecting information concerning the progress of the project. Control involves using the data obtained through monitoring to bring actual performance into congruence with the plan.

15.5.1 Earned Value Monitoring

Time, cost and quality are the three major aspects of a software project that need to be controlled. Earned value can be used to analyze the performance of a project with regard to schedule and cost.

Creating a baseline budget is the first step in earned value monitoring. The value of each work unit is estimated in monetary terms called 'earned value'. In people-intensive projects such as a software project, the value may also be estimated in person-hours. If the rate of a person-hour is known, estimation of earned value in monetary terms is easy and may be more convenient.

Costs are budgeted period by period for each work unit. Once the project begins, the actual work and actual cost are monitored periodically and compared with the plan. Where tasks have been started some consistent method is used to assign earned value. The 50/50 rule is the most popular method for assigning earned value. According to this method if a task is completed, its earned value is 100%. If the task has started but is not complete, it is considered half complete and its earned value is 50%.

The status of a project is assessed with three parameters as given below.

Budgeted Cost of Work Scheduled (BCWS): It is the sum of budgeted cost of all work units that should have been completed plus apportioned cost of work units that should have started as per planned schedule at any given instance of time.

Actual Cost of Work Performed (ACWP): It is the sum of actual expenditure incurred in a given time period. It is the actual cost incurred on all completed work units plus incomplete work.

Budgeted Cost of Work Performed (BCWP): It is the sum of budgeted cost of all work units that are actually completed plus apportioned cost of semi-complete work units at any given instance of time.

Based on the above parameters, the following variances are determined for analyzing the performance of the project:

$$\begin{array}{ll} \text{Schedule Variance,} & SV = BCWP - BCWS \\ \text{Cost Variance,} & CV = BCWP - ACWP \\ \text{Accounting Variance,} & AV = BCWS - ACWP \end{array}$$

The above variances denote whether a project is lagging behind its schedule, overrunning its budgeted cost or both.

15.5.2 Gantt Chart

The project schedule is the timetable of activities. It specifies the start date and end date of each activity. The schedule is decided based on:

- Activity dependency
- Criticality of activity. The activities that are critical are scheduled on priority.
- Resources required for activities and availability of these resources at the given time. The human resource is the most important resource for a software project.

The project schedule can be graphically depicted by the Gantt chart. It is named after its developer Henry Gantt. It is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of bar of an activity is proportional to its duration.

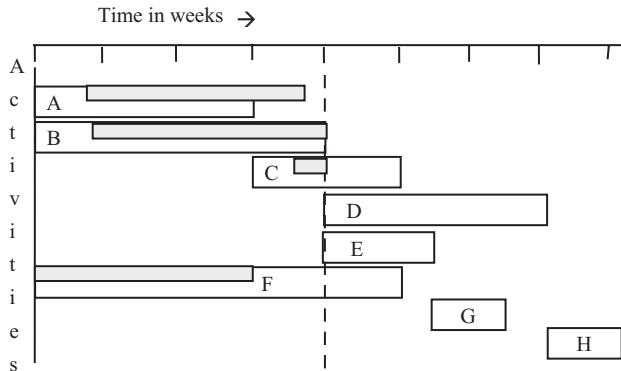


Figure 15.10 Gantt chart

The chart depicts the schedule of various activities of a project. The Gantt chart is also used for depicting the progress of the project. Its representation for the problem given in Table 15.5 is shown in Figure 15.10.

The white bar shows the schedule of all activities and their progress. In the above Gantt chart the actual progress is shown after 8 weeks from the scheduled start date. Hence, Gantt chart is a popular tool for scheduling as well as for schedule monitoring.

15.5.3 Causes of Project Failure

The three criteria of success of any software project are performance (quality), cost and time. Software projects fail when they do not meet one or more of the above criteria. The software project can be considered to have failed when:

- It is not delivered on time.
- It is over budget.
- The system does not work as required.

Only a few projects achieve all three. Many more are delivered, which fail on one or more of these criteria, and a substantial number are cancelled having failed badly.

Hence, what are the key factors for success? Organizations and individuals have studied a number of projects that have both succeeded and failed and some common factors emerge. A key finding is that there is no one overriding factor that causes project failure. Some possible causes of software project failures are listed below.

Incorrect Time Estimation: Error in time estimation of project activities may occur due to superficial understanding of project complexities, lack of experience of estimator, not considering risk factors due to overoptimism etc. Incorrect time estimation results in incorrect project schedule and consequent project failure.

Unrealistic Schedules: Sometimes managers think that pushing for an aggressive schedule would accelerate the work, but it actually delays it. When faced with an unrealistic schedule, people often behave irrationally. They race through the requirements, produce a superficial design and rush into coding. This bid to somehow build something not only results in a poor-quality product but also causes delay due to more occurrences of errors.

Insufficient Budget: Sometimes low budget is quoted for a software project in a desperate bid to get the project approved. However, the project may fail due to scarcity of funds.

External Dependencies: Unexpected delay in activities being done by others may cause project failure. Some of these activities are: Slow response of clients in giving feedback, delay by client in according approval of prototype, delay by subcontractors etc.

Unexpected Expansion of Project Scope: Addition of features by clients during the course of the project disturbs the project plan. Expansion in project scope causes delay and budget overruns as such expansions may not have been considered in the original estimates.

Insufficient Software Testing: The functionality of software and software performance is concerned with the technical aspects. To compensate for schedule overrun, downstream activity times such as time for software testing are sometimes reduced. This causes risks due to improper software testing and consequent problems during the later stages of software development.

Breakdown in Communications: Effective team communication is an essential part of project management. Particularly in people-intensive projects such as software projects, there is a strong need for established communication structure and conflict resolution process.

Poor Understanding of Users' Requirements: The developers generally do not have any real knowledge of the business. They build the software based on their understanding of users' requirement. Hence, if the requirements are vague, and not clearly stated, the software so developed is likely to be faulty. Hence, when the system is delivered it fails to fulfill users' requirement. This problem is closely linked to lack of user involvement. Users must know what they want, and be able to specify it precisely.

Improper Monitoring and Control: As a project progresses things change and these changes can have a significant impact. It is important to monitor progress on a regular basis so that challenges can be overcome early and stakeholders warned of possible delays and changes to the product. Sometimes too much monitoring also causes delay because much effort and time is used in updating the status of work rather than on the work.

Lack of Experience and Training: Software development is knowledge work. It requires sufficiently trained people to do the work. When some experienced people leave the job, quick replacement is not available. Training of people is also affected due to work pressure.

A delay in a critical activity has a cascading effect on the entire project. Delay of projects may also have certain penalty costs associated with them, which may cause budget overrun.

15.6 RISK MANAGEMENT

As discussed in the previous section, there are large numbers of potential causes of project failure. The software project may fail due to any of the causes, which may be due to some weakness in the organization's infrastructure, its people or its management. With proper planning, many of these causes can be removed so that projects do not fail. However still, due to uncertainty factor some unwanted events may occur during the execution of the project, which may adversely affect the software project. The occurrences of these unwanted events are the risks of the software project.

Risk is defined by International Organization for Standardization (ISO) as the effect of uncertainty on objectives. Risks are future uncertain events with a probability of occurrence and a potential for loss. Risks are simply potential problems. People are aware of the potential dangers that permeate even simple daily activities, such as driving at high speed, investing money in stocks, going to unknown and secluded

places at late night, having a surgery due some medical complication etc. People generally have an innate sense of risk that comes from three reasons:

1. There is some uncertainty about the outcome.
2. The process and the environment are not familiar.
3. The stakes are high.

All the above conditions are present in the case of software development projects. Hence, while planning and execution of software projects, various risks factors involved should be given due consideration. Risk identification and management are the main concerns in every software project. Effective analysis of software risks will help to effective planning and assignments of work.

15.6.1 Types of Risks

Dr. Barry W Boehm, a renowned expert in software engineering, in his article 'Software Risk Management: Principles and Practices' lists ten major software risk items. These are given below.

1. Personnel Shortfalls
2. Unrealistic schedules and budgets
3. Developing the wrong functions and properties
4. Developing the wrong user interface
5. Gold-plating, i.e. trying to make an ideal product
6. Continuing stream of requirements changes
7. Shortfalls in externally furnished components
8. Shortfalls in externally performed tasks
9. Real-time performance shortfalls
10. Straining computer-science capabilities

In addition to the above, there can be many more risks associated with software projects. The risks can be generic, i.e. common to all projects, or specific to particular software project. The various categories of risks associated with software project management can be grouped into the following categories:

- | | |
|---------------------------|--|
| 1. Schedule Related Risks | Likely events that can disrupt project schedule or cause delay in project completion |
| 2. Financial Risks | Likely events that can cause budget overruns or scarcity of finance |
| 3. Technical Risks | Likely errors made due to technical complexity of the software project and lack of adequate technical expertise/experience. Risks associated with technical aspect of software |
| 4. Operational Risks | Likely events that can adversely affect administrative and operational activities |
| 5. Other Risks | Unavoidable risks due to likely changes in economic and political conditions |

Risks are identified, classified and managed before actual execution of the program. These risks are classified into different categories.

15.6.2 Risk Management Activities

Dr. Boehm describes two categories of activities for risk management. These are:

1. Risk Assessment
2. Risk Control

Risk assessment is concerned with identifying the possible causes that can impede the software development, estimating the adverse effect of these causes and the probability of their occurrence. The steps are:

- Make a list of all of the events (risk items) that can adversely affect the project.
- Assess the probability of occurrence of each of these listed items.
- Assess the potential adverse effect of each of these listed items.
- Rank the items from most to least dangerous based on their potential adverse effect and probability of occurrence.

The purpose of risk assessment is to select the major risk items to focus on from the list containing a large number of possible risk items. The idea is to identify the vital few from the trivial many.

Risk Control is concerned with coming up with techniques and strategies to mitigate the major risks that are identified through risk assessment. The steps of risk control are:

- Make strategies and contingency plans for major risk items.
- Set up procedures to ensure that the contingency plans are executed when needed.
- Implement the strategies to resolve the major risk items.
- Monitor the actual implementation of risk abatement strategies and contingency plans throughout the project.

The purpose of risk control is to do something about the major risks so that either the occurrence of these risks is prevented or their adverse effect is minimized. While common sense is essential for effective risk management, many formal techniques, methods and tools can be used to enhance the abilities to deal with risks.

The assessment includes identification as well as prioritization of risks. The control function includes planning and application of resources to reduce the probability of occurrence and/or adverse effect of unwanted events. **Risk management** can therefore be considered the identification, assessment and prioritization of risks followed by coordinated and economical application of resources to minimize, monitor and control the probability and/or impact of unwanted events.

15.6.3 Risk Management Strategies

Depending on assessment of risks, various strategies may be adopted for their control/mitigation. These strategies are given below.

Risk Acceptance/Retention: This technique recognizes the risk and its uncontrollability. Acceptance is a 'passive' technique that focuses on allowing whatever outcome to occur without trying to prevent that outcome. This technique is normally used for 'low' or 'very low' risks where a scope for efficient means of reducing the risk is not apparent.

Risk Avoidance: This technique uses an approach that avoids the possibility of risk occurrence. Avoidance can be thought of as nullifying the risk by changing the contract parameters established between the Customer and Integrator. The following items represent ways of avoiding risks:

1. Work Scope Reduction
2. Changing the requirements and/or specifications
3. Changing the Statement of Work (SOW)
4. Changing the Technical Baseline
5. Developing and submitting Waivers and Deviations

Risk Transfer/Sharing: Transference is the process of moving something from one place to another or from one party to another. In this, the risk can be transferred to the customer or to the contractor. Typically, transference includes subcontracting to specialist suppliers who are able to reduce the overall risk exposure. This technique is best utilized during the proposal process. Transfer can also include the use of third party guaranties, such as insurance-backed performance bonds.

Risk Reduction/Mitigation: This technique is made up of actions that are to be taken, which reduce the risk likelihood or impact. Control-based actions occur at all points throughout the program's lifecycle and are typically the most common response. They typically identify an action or product that becomes part of the work plans, and which are monitored and reported as a part of the regular performance analysis and progress reporting of the Program.

Risk reduction is the active lowering of risk by a planned series of activities. Techniques include:

- Advance design models
- Reduce Dependencies
- Customer involvement
- Joint Applications development groups

Risk Investigation: This technique defers all actions until further investigation is done and/or facts are known. When no clear solution to a risk item could be identified, further action may be deferred till some solution is found by investigation/research. Investigation may include various methods/tools such as Multi-discipline involvement, Consultant/specialist reviews, Trade Studies, Team Workshops, Simulation modeling, Statistical analysis (Root cause analysis, Failure mode analysis) etc.

15.7 CONFIGURATION MANAGEMENT

Configuration Management (CM) is a field of management that focuses on establishing and maintaining consistency of a system's or product's performance and its functional and physical attributes with its requirements, design and operational information throughout its life.

CM was first developed by the United States Air Force for the Department of Defense in the 1950s as a technical management discipline to manage changes in hardware. The concepts of this discipline have been widely adopted by numerous technical management functions, such as Systems Engineering (SE), Integrated Logistics Support (ILS), Capability Maturity Model Integration (CMMI), ISO 9000, Product Lifecycle Management, Application Lifecycle Management etc. Many of these functions and models have redefined CM to suit specific needs.

SCM is the task of tracking and controlling changes in software. It is concerned with managing change in the software product during its development and future revisions/modifications.

15.7.1 Need for Configuration Management

Changes in work products during the development of software are quite common. The changes take place due to following reasons:

- The requirements of users change during development of software. This happens because the users gain knowledge about the system and the requirements by involving themselves with software development. As the users get involved with the software development, they gain more knowledge about the system. By this process they are able to identify new requirements that can make the system more effective. They may also discover flaws in their earlier requirements. Software developers are expected to incorporate these changes in the requirements while developing the software.
- For similar reasons, the change in work products may take place at the developers' end. In the software development process, the work product (deliverable) of one activity may be used as input by a number of subsequent activities. A software engineer while working on subsequent activities may find some flaws in an earlier work product and change it. Hence, this may create inconsistencies in the earlier work product and the changed one.

There is always scope for improvement in software development or for that matter in any product. Hence, if we keep modifying the work products, there will be no end to it. This is sometimes referred to as 'Gold Plating'. Continuous changes in the work product delays the project. It also escalates the project cost. Hence, it is necessary to have some control on changes.

If the work products are changed frequently, these may get mixed up. Hence, some persons may work on the old work product and some may work on the new. This is likely to create inconsistencies and problems during integration. In later stages it may become too difficult to know which work product is old and which is the revised one. Hence, it is also necessary to have a mechanism by which the changes made in a work product are formalized/documentated so that all people work on the latest work products.

15.7.2 Configuration Management Process

In the SCM process, the functional and physical attributes of the software are identified at various points in time, and systematic control of changes to these attributes are performed for the purpose of maintaining software integrity and traceability throughout the software development life cycle.

The SCM process emphasizes the need to trace changes, and to verify that the final delivered software has all of the planned enhancements that are supposed to be included in the release.

CM comprises two major processes:

- (i) Configuration identification
- (ii) Configuration control

Configuration identification

For configuration identification, the objects associated with software development can be classified into two main categories: (1) Controllable and Uncontrollable.

Controllable objects are those that can be changed and are put under configuration control. Uncontrolled objects are those that cannot be changed. These are not subjected to configuration control. There

may be some objects that are not selected for control for the time being but are likely to be brought under control at later stages. These objects may be referred to as pre-controlled objects. Controllable objects include pre-controlled objects. Some typical controllable objects are given below.

- Software Requirement Specification (SRS) document
- Design documents
- Tools used for software development, such as platforms, compilers, linkers, lexical analyzers etc.
- Source code of modules
- Test cases

The CM plan is made during project planning. It lists all controlled objects.

Change is inevitable. Change is necessary for correction of errors and for improvement. Hence, it is necessary to have a mechanism for identification of parts of the system that may require change and a mechanism for initiating the change.

Configuration control

It is the process of managing changes to controlled objects. For this it is necessary that:

- Changes to objects/work products should be done only by authorized persons.
- There should be some specified procedure for making changes and the changes should be carried out as per procedure.
- The same object should not be changed simultaneously by two or more persons.

CM practices include revision control and the establishment of baselines. The following steps may be followed for configuration control:

Identification of Change: The parts of the system that require some change are identified and a request is initiated for change.

Assessment of Change Request: Necessity of change and the effect of change on the work product and other related work products are assessed. Based on assessment change is approved by the authorized person.

Planning for Change: A plan for implementation of change is made keeping in view the aspects of traceability and control.

Implementation of Change: The changes are carried out as per plan.

Testing the Change: After implementation of change, the work product(s) is tested for correctness. This ensures that the change does not affect the system adversely.

Acceptance and Release of Change: Based on testing if the changed is found to be beneficial, it is accepted and approved by the authorized persons. Generally, a committee comprising members from development called 'Change Control Board (CCB)' approves the changes. The change is formally documented and baselined (released), so that further development work is done based on the changed work product.

Besides configuration identification and configuration control, CM involves some other processes as well.

Configuration status accounting is the process to record and report on the configuration baselines associated with each configuration item at any moment of time.

Configuration audit process ensures that the changes have been made only where these are necessary and that all procedures have been observed. Configuration audits occur either at delivery or at the moment of effecting the change. These are broken into two types: (1) Functional configuration audit and (2) Physical configuration audit. Functional configuration audit ensures that functional and performance attributes of a configuration item are achieved, while physical configuration audit ensures that a configuration item is installed in accordance with the requirements of its design documentation.

15.7.3 Software Version and Revision

Once software is installed, it is often used for many years. However, the requirements of users are never static. Modifications need to be made to the software, data files and procedures to meet the continual change in users' requirements, organization systems and the business environment. This is called software maintenance. It is an ongoing process.

Sometimes a minor bug may be noticed in the software. Hence, the bug is fixed and a new revision of software released. Software revision refers to fixing of the minor bug in the software. Sometimes a new revision is also created for minor enhancements to the functionality, usability etc.

On the other hand, a new version of software is created when there is a significant change in functionality or technology of software.

For example, let us consider that certain software works on Microsoft Window environment only. If this software is modified so that it works on LINUX environment, then it is a new version of the software. After release of the software if some minor errors are discovered and corrected or certain minor enhancements to the functionalities of the software are done then the new improved software is called software revision. The release of the software is often specified as m.n. It means the software is of version m, release n.

SUMMARY

Projects need to be performed and delivered within three main constraints, namely, scope, time and cost. The main objective of project management is to meet the scope of the project at minimum cost and time.

Software projects require many types of plans. These are: (1) Software development plan, (2) Testing and validation plan, (3) Quality assurance plan, (4) CM plan, (5) Maintenance Plan and (6) HR plan.

Software development plan forms the basis for execution of the software project. Estimation of project attributes, WBS of project, scheduling of project work component, determination of resources, identification and assessment of risks, and monitoring and control are some activities of development plan.

WBS is a hierarchical depiction of all components of work required to accomplish the entire work of the project. WBS can be of three types: (1) Activity-based WBS, (2) Product-based WBS and (3) Hybrid WBS.

There can be four types of dependency, namely, FS, SS, FF and SF. The network diagram shows the dependency relationship between activities. AON and AOA are of two types of network diagrams.

CPM and PERT are two important techniques of project planning. CPM is a deterministic model whereas PERT is a probabilistic model. PERT uses three time estimates, namely optimistic time, most likely time and pessimistic time.

Earned value analysis is an effective tool for project monitoring. It uses three parameters, namely, BCWS, ACWP and BCWP to assess the status of any project. Gantt chart is a popular tool for scheduling as well as for schedule monitoring.

A software project can be considered to have failed when it is late, over budget or fails to work as required. Incorrect time estimation, unrealistic schedules, insufficient budget, external dependencies, unexpected expansion of project scope, insufficient software testing, breakdown in communications, poor understanding of users' requirements, improper monitoring and control, lack of experience and training are some possible causes of software project failures.

Risk may be defined as the effect of uncertainty on objectives. The various categories of risks associated with software project management can be categorized into Schedule risks, Financial risks, Technical risks, Operational risks and Unavoidable risks. Risk assessment and risk control are two major categories of activities of risk management. Risk acceptance, risk avoidance, risk transfer, risk reduction and risk investigation are some strategies of risk management.

SCM is the task of tracking and controlling changes in software. Configuration identification, configuration control, configuration status accounting and configuration audit are major processes of CM.

EXERCISES

1. List the major responsibilities of a software project manager.
2. What are the necessary skills to become a successful software project manager?
3. List some important activities that are performed during software project planning.
4. List various project-related estimates required for software project management.
5. List the important items that a Software Project Management Plan (SPMP) document should discuss.
6. How does the change of project duration affect the overall project development effort and development cost?
7. List the necessary tasks that are performed to perform project scheduling.
8. Explain WBS and its usefulness to project planning.
9. What is dependency between activities? Explain how it is depicted by activity network diagram?
10. Distinguish between AON and AOA activity network diagram.
11. What are the four types of dependency between activities?
12. Explain CPM.
13. State the main difference between CPM and PERT.
14. What do mean by project crashing and resource leveling?
15. Describe Earned Value Analysis for project monitoring.
16. Write the application of Gantt charts.
17. Enumerate any three possible causes of project failure. Suggest some precautions to avoid failure.
18. Enumerate various types of risks associated with a software project.
19. Define CM. Explain why it is needed.
20. Briefly describe different processes of CM.
21. Write the steps of CM.
22. Differentiate between software version and software revision.

SOFTWARE QUALITY MANAGEMENT

This chapter gives concepts of quality in general and with respect to software. The fundamental principles of Quality Management based on thoughts of pioneers in this area are covered with reference to their applicability to software development. Some of the concepts covered are:

- *Inspection, Quality Control (QC), Quality Assurance (QA) and Total Quality Management (TQM)*
- *Random and Assignable Causes of Error*
- *Plan-Do-Check-Act (PDCA) Cycle*
- *Quality Trilogy*
- *Cost of Quality (CoQ)*
- *Factory within Factory and Zero Defects*

The Process Quality Models, Process Improvement and Six Sigma, Process Standard: ISO 9000, Process Standard: ISO 12207 and Capability Maturity Model (CMM) are some important topics of this chapter.

Product quality as perceived by the customers is a major factor that determines sales success. It could be argued that powerful media campaigns can create demand for a product. However, sustained sales are achieved on the basis of good-quality products. Hence, in a global economy, quality has been an increasingly important stated objective of organizations. Organizations buy software on the belief that the computerised system will improve their operational performance. In order to meet the above expectation of users, it is important that software should be of good quality. Hence, quality is very important for software industries. Quality is not achieved by accident. It is a result of thorough planning and concerted effort based on principles and concepts of quality management. The principles and concepts of quality management are universal in nature and are equally applicable to software development as for any other products or services.

16.1 THE CONCEPT OF QUALITY

Quality in the traditional sense means conformity to specification or the absence of defects. Some definitions of quality are given below.

Conformance to specification—Philip B Crosby

Fitness for use—J M Juran

Predictable degree of uniformity and dependability at low cost and suited to market

—W Edward Deming

Degree of excellence at an acceptable price and control of variability at an acceptable cost—Broth

The definition given by the International Organization for Standardization (ISO) is generally considered as the most acceptable definition of quality. It defines quality as:

“Totality of features and characteristics of product/service that has ability to satisfy stated and implied needs of customer”.

—*ISO 8402 vocabulary*

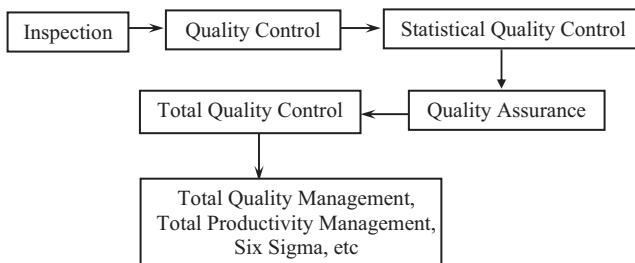
The modern marketing concept is based on this concept of identifying the specific needs of the customer and delivering products/services that meet these specific needs. The product/service should meet basic quality attributes as well as specific (special) requirements of customers. There are several attributes that determine the quality of software. These have been discussed earlier in software metrics. Some attributes of software quality are listed below.

| Attribute | Description |
|----------------------|--|
| 1. Correctness | Extent to which a program meets system specifications and user objectives |
| 2. Reliability | Degree to which the system performs its intended functions over a period of time |
| 3. Efficiency | Amount of computer resources required by a program to perform a function |
| 4. Ease of Use | Effort required to learn and operate a system |
| 5. Accuracy | Required precision in input editing, computations and output |
| 6. Portability | Ease of transporting a program from one hardware configuration to another |
| 7. Control and audit | Control of access to the system and the extent to which it permits data and processing to be audited |
| 8. Maintainability | Ease with which errors are located and corrected |
| 9. Expandability | Ease of adding or expanding the existing data base. |

16.2 EVOLUTION OF QUALITY MANAGEMENT

Quality in the traditional sense means output should conform to specifications. Earlier, for achieving quality the emphasis was on thorough testing of the final product before it was sold to the customer. Testing was done after the product was made. Its purpose was only to detect errors. Rectification of errors required extra time, effort and cost. Hence, testing was not the complete solution to quality problems. Rather instead of error detection the emphasis should have been on prevention of errors.

There has been much development in the field of Quality Management during the last six decades. Thoughts of Quality Management have evolved in the following sequence:



Inspection: It involves sorting the products that conform to specifications from the non-conforming products.

Quality Control (QC): Defect if detected early is much easier to rectify. It recognizes the need to identify and eliminate poor quality at the source. It involves continued inspection and monitoring of the processes.

Statistical QC: It is the application of statistics to QC. It uses various statistical distributions to measure the degree of conformance of inputs materials, processes and products to previously laid down specifications.

Quality Assurance (QA): In addition to QC, QA includes sufficient documentation, process analysis and administration to provide confidence that a product or service will satisfy the given requirements of quality. QA encourages implementation of procedures to comply with set standards at all stages of a process.

Total Quality Control (TQC): It is QA, including a commitment for continuous improvement involving every functional area and individual of an organization. The concept is also referred to as Company Wide Quality Control (CWQC). It recognizes that quality is everyone's responsibility. TQC also involves statistical control of quality applied to all the functional areas of an organization, including all steps of planning, design, production, service, marketing, finance and administration.

Total Quality Management (TQM): It refers Company Wide QA using system approach in terms of documented sets of procedures and an endeavor towards continuous improvement in every sphere of activity. Hence, in every step towards progress in quality management, the scope was broadened to cover wider range of activities being done in the organization. The broader view is generally referred to by organizations as TQM).

The evolution is still continuing and new concepts such as Total Business Excellence, Total Marketing Leadership, Total Productive Maintenance, Business Process Reengineering, Six Sigma, Lean Six Sigma etc. have evolved to broaden the scope of Quality Management.

The modern concept of quality management focuses not only on quality of the output but also on the inputs, the processes and environment. It advocates that if the inputs, the processes and the environment are good, then quality of output will be good. This statement, which appears to be common sense, has great implications for quality. This means that errors can be reduced by improvement of processes. If errors are reduced, there will be less need for elaborate testing and consequently overall cost is reduced. Hence, the organizations must strive to continuously improve their processes and people.

16.3 SOME THOUGHTS OF QUALITY GURUS

The modern principles and concepts of quality management have been shaped by pioneering thoughts and work of some persons. Edward W Deming, Joseph M Juran, Philip B Crosby, A V Feigenbaum,

K Ishikawa are some such pioneers in quality management. They are regarded as Quality Gurus. Their propositions are the foundation for understanding the concept of quality management. Some of these important thoughts are as follows:

Management's Responsibility: Deming observed that in most cases, the major proportions (about 75% to 85%) of failure are caused due to fault of the system controlled by the management. It is the management's responsibility to provide commitment, leadership, empowerment, encouragement and the appropriate support to technical and human processes.

ISO has described Quality Management as that aspect of the overall management function that determines and implements quality policy. As such it is the responsibility of the top management (ISO8402: 1986).

Hence, the management should foster participation of the employees in quality improvement, and develop a quality culture by changing perception and attitudes toward quality.

Random and Assignable Causes of Error: Dr. Deming W Edwards, an American quality expert, made significant contribution to the quality movement in Japan. Deming's approach to quality is aimed at understanding variations that may be caused either by assignable causes or random (un-assignable) causes. If significant variation is observed even after the assignable causes have been sorted out, quality improvement can come about only by redesigning or making changes in the system.

Plan-Do-Check-Act (PDCA) Cycle: Dr. Deming introduced the PDCA cycle for making improvements in a system. According to the PDCA cycle, to bring about improvement in a system, a company plans a change, does it, checks the result and depending upon the results, acts either to standardize the change or to begin the cycle of improvement again with a fresh approach. Dr. Deming advocated continuous improvement through this cyclic approach.

Quality Trilogy: Dr. Joseph M Juran is an American, who rose to fame during the 1950s as management consultant in Japan. Dr. Juran has authored ten books and a large number of papers. Some of his important books are 'Quality Planning and Analysis' (co-authored with F M Gryna), 'Managerial Breakthrough' and 'The Quality Control Handbook'.

Juran focuses on three major processes necessary to bring about quality, which he referred to as 'quality trilogy'. These three processes are: (1) QC, (2) Quality improvement and (3) Quality planning. In his view, the approach to managing for quality consists of two types of problems, namely, sporadic problem and chronic problem. Sporadic problems are caused at random once in a while and are acted upon by the process of QC. Chronic problems occur because of some assignable causes and these require a different process, namely, quality improvement. Chronic problems are traceable to an inadequate quality planning process. Juran defined a universal sequence of activities for the three quality processes as listed in Table 16.1.

Table 16.1 Three Processes for Managing Quality

| Quality planning | QC | Quality improvement |
|----------------------------|--|--|
| Establish quality goals | Choose control subjects | Prove the need |
| Identify customers | Choose units of measure | Identify projects |
| Discover customer needs | Set goals | Organize project teams |
| Develop product features | Create a sensor | Diagnose the causes |
| Develop process features | Measure actual performance | Provide remedies, prove its effectiveness |
| Establish process controls | Interpret the difference Take corrective action | Take action on the difference Deal with resistance to change Control to hold the gains |

Juran's assessment of most companies reveals that whereas QC is a priority, quality planning and quality improvements are generally neglected. He feels that more effort should be placed on quality planning and more so on quality improvement.

Cost of Quality (CoQ): Juran defined four broad categories of quality costs, which can be used to evaluate the firm's costs related to quality. These are listed as follows:

1. Internal failure costs (scrap, rework, failure analysis etc.) associated with defects found prior to transfer of the product to the customer
2. External failure costs (warranty charges, complaint adjustment, returned product etc.) associated with defects found after the product is transferred to the customer
3. Appraisal costs (incoming, in-process, and final inspection and testing, product quality audits, maintaining accuracy of testing equipment etc.) incurred in determining the degree of conformance to quality requirements
4. Prevention costs (quality planning, new product review, quality audits, supplier quality evaluation, training etc.) incurred in keeping failure and appraisal costs to a minimum

CoQ is an important concept in quality management and a popular area of research. Feigenbaum A V (1991) has also made significant contribution to development of this concept. It is valuable information for quality improvement.

It has been estimated that for manufactured goods, CoQ is about 10–25% of total cost. CoQ is even more in case of software development. Hence, the concept of CoQ is also relevant to software development. By making improvement in quality system, a company can reduce its CoQ.

Factory within Factory: Philip B Crosby had headed the QC function in the International Telephone and Telegraph (ITT) Corporation and had developed the 'zero defect' program, which was later adopted as a QC philosophy even in many other commercial corporations. In his book 'Quality is Free' he offers a quote from his previous CEO of ITT, Harold Geneen:

"Quality is not only right, it is free. And it is not only free; it is the most profitable product line we have."

He estimated that in a typical industry, significant amount of human and other resources are involved in reworking defective product to make it acceptable. He termed it as 'factory within factory'. He attributed lack of discipline in the way the products are conceived, designed, specified, made, tested, packaged, stored etc. (i.e. the processes) as the reason for such defective products. This observation is also applicable to the software industry. At each stage of software development, significant amount of time and effort is spent on error correction. It would cost much less to do the things right the first time and every time.

Zero Defects: According to Crosby, companies incur significant amount of total cost towards detection (testing) and rectification of defects. The occurrence of defects can be reduced by redirecting effort and resources on improvement of processes. In the context of software, quality can happen if measures are taken to prevent occurrences of errors. Errors can be prevented by understanding the process, identifying the causes of errors and eliminating those causes. However, the mindset of people often becomes a major obstacle in prevention of defects. People have dual standards towards tolerance to defect. In personal life people do not accept errors. However, in work life people are conditioned to believe that errors are inevitable. Hence, Crosby emphasized that acceptance of defects is the major cause of defects. Therefore, zero defect/error should be the only performance standard of quality.

Raising the Quality Bar: Feigenbaum prescribed following steps to manage quality effectively:

- Set quality standards
- Appraise conformance to these standards
- Act when standards are not met
- Plan for improvement in these standards

All functional activities, such as marketing, design, purchasing, manufacturing, inspection, shipping, installation and service etc., are involved in and influence the attainment of quality. Hence, a high degree of effective functional integration among people, machines and information is necessary for quality.

Quality—a Way of Life: Feigenbaum emphasized that quality is an integral part of the day-to-day work of the line, staff and operatives of a firm. There are two factors affecting product quality: (1) technological factors and (2) human factor. Of these, human factor is of greater importance by far. Ishikawa also supported this view and argued that quality management extends beyond the product and encompasses wide aspects such as after-sales service, quality of management, quality of individuals and quality of the firm itself. He claimed that the success of a firm is highly dependent on treating quality improvement as a never-ending quest. A commitment to continuous improvement can ensure that people will never stop learning.

Ishikawa has been associated with the development of seven QC tools. QC tools are statistical tools, which are easy to learn and use in work situations for process improvement. The seven QC tools are: (1) Flow charts, (2) Check sheets, (3) Histograms, (4) Cause-and-effect diagrams, (5) Pareto diagrams, (6) Scatter diagrams and (7) Control charts.

Ishikawa advocated employee participation and team work such as 'Quality Circle' as the key to quality. He emphasized the importance of education, stating that quality begins and ends with it.

16.4 PROCESS QUALITY MODELS

Modern concepts of quality management give much emphasis on quality of the process. If the process is good and properly followed, then it will reflect on the quality of the product. Based on this premise, many countries have instituted quality awards to encourage industrial houses to improve their processes. The evaluation criteria of these quality awards may be regarded as models of process quality. Some of the most important models for process evaluation are given below.

16.4.1 The Deming Prize

The Deming Prize was established by the Japanese Union of Scientists and Engineers (JUSE) in 1951. Its main purpose was to spread the concepts of TQM. The Deming prize has 10 elements. These are: (1) Policies, (2) Organization, (3) Information, (4) Standardization, (5) Human resources, (6) QA, (7) Maintenance, (8) Improvement, (9) Effects and (10) Future plans.

The primary elements of the Deming prize and some important checklists are listed in Table 16.2.

The evaluation model of Deming's prize gives the maximum importance to top executives' leadership, vision, long-term plans and their understanding of modern concepts of management.

16.4.2 Baldridge (MBNQA) Model

In response to success of the quality movement in Japan, the United States instituted a quality award in 1987, in honour of Malcolm Baldridge, a quality professional and late American secretary of commerce.

Table 16.2 Elements and Checklist of the Deming Prize

| Elements | Checklists |
|---------------------|--|
| (1) Policies | Policies relating to quality and their place in overall business management; Methods and processes for establishing policies; Clarity and communication (deployment) of policies, Top executives' leadership |
| (2) Organization | Organizational structure for quality and status of employee involvement; Clarity of authority and responsibility; Status of interdepartmental coordination and team activities; Relationships with associated companies (group companies, vendors, contractors etc.) |
| (3) Information | Appropriateness of information, its utilization and retention; Status of utilizing statistical techniques for data analysis |
| (4) Standardization | Appropriateness of the system of standards; Procedures for establishing, revising and abolishing standards; Contents of standards; Status of utilizing and adhering to standards |
| (5) Human resources | Education and training plans; Status of supporting and motivating self-development of employees; Status of understanding and utilizing statistical concepts and methods; Status of Quality Circle development |
| (6) QA | Status of inspection, quality evaluation and quality audit; Status of QC diagnosis; Status of process control, analysis and improvement; Status of new product and technology development; Status of customer satisfaction |
| (7) Maintenance | Methods for determining control items and their levels; Status of utilizing control charts and other tools; Status of taking temporary and permanent measures; Status of operating management systems for cost, quantity, delivery etc. |
| (8) Improvement | Methods of selecting themes (important activities, problems and priority issues); Linkage of analytical methods and technology; Status of utilizing statistical methods for analysis and utilization of results for maintenance/control activities; Contribution of QC circle activities |
| (9) Effects | Tangible effects such as quality, delivery, cost, profit, safety and environment; Intangible effects such as customer satisfaction, employee satisfaction; Influence on associated companies and communities |
| (10) Future plans | Status of grasping current situations; Projection of changes in social environment and customer requirements and future plans based on these projected changes; Relationships among management philosophy, vision and long-term plans |

The award was called Malcolm Baldrige National Quality Award (MBNQA). The MBNQA evaluation model consists of seven measures as given below:

1. Leadership
2. Information and Analysis
3. Strategic Quality Planning
4. Human Resource Management
5. Management of Process Quality
6. Quality and Operational Results
7. Customer Focus and Satisfaction

Table 16.3 Award criteria of MBNQA

| Quality Measures | Maximum points |
|---|----------------|
| 1. Leadership: Top executive leadership, responsibility and commitment for quality | 90 |
| 2. Information and Analysis: Scope and management of quality and performance data; Competitive comparison and benchmarks; Analysis and use of information | 80 |
| 3. Strategic Quality Planning: Strategic quality planning process; Quality and performance plans | 60 |
| 4. Human Resource Management: Employee training and development; Performance appraisal; Employee well-being, involvement and morale | 150 |
| 5. Management of Process Quality: Design and introduction of product and services, Production and delivery process; Business process and service quality; Supplier quality; Quality assessment | 140 |
| 6. Quality and Operational Results: Product and service quality results; Company operational results; Business process results; Supplier quality results | 180 |
| 7. Customer Focus and Satisfaction: Customer relationship management; Commitment to customer; Customer satisfaction determination; results and comparison; Future requirements and expectations of customers | 300 |
| Total Points | 1000 |

Each of the seven measures consists of certain award points. The total points add up to 1000 points. The breakup of the points is given in Table 16.3.

The MBNQA model provides useful guidelines for design and implementation of quality system in any organization.

16.4.3 European Foundation for Quality Management (EFQM) Business Excellence Model

The European Quality Award was officially launched in 1991. This model of the quality award is called European Business Excellence Model. It consists of two types elements: (1) Enablers and (2) Results. The enablers are: (1) Leadership, (2) People, (3) Organizational policies and strategies, (4) Resources and (5) Processes. These five aspects steer the business and facilitate the transformation of inputs to outputs. The results are people satisfaction, customer satisfaction, impact on society and business results (the measure of the output attained by the firm). In totality therefore, the European Business Excellence Model consists of nine primary elements. The EFQM Business Excellence model presents the community as a stakeholder group.

In addition to the above models a large number of Quality Management models have been developed by various researchers, associations/societies for quality and corporate houses of various countries. All the models enlarge the scope of quality management and emphasize on following core concepts.

- Focus on the customer to determine their exact requirements in order to meet those needs
- Senior management commitment and involvement for improving quality
- Total involvement and participation of all people
- Endeavor towards continuous improvement as a core ingredient of organization culture. This includes improvement of people through training and education

- Emphasis on measurement, its analysis and systematic approach for problem solving
- Participative management and empowerment of people for harnessing their potential for improvement, growth and creativity
- Emphasis on teamwork and group performance

The above concepts are the core concepts of TQM. The quality models are universally applicable to any product or service. These models serve as a useful guide for designing the quality system and processes of a firm.

16.5 QUALITY ASSURANCE

The purpose of Quality Assurance (QA) is to ensure that things are done right the first time and every time. Consistency in quality is very important. This requires a system for quality management.

16.5.1 Concepts and Definition

QA is a set of activities, action plan and review of software products and related documentation carried out to ensure that the system meets the specifications and requirements for its intended use. It emphasizes on adherence to some standards for software product, processes and procedures to ensure detection of errors, timely corrective actions and prevention of errors. For QA the products are reviewed throughout the software development life cycle to eliminate defects at critical points.

Software QA process includes activities that give assurance that standards and procedures are established and are being followed throughout the software development project.

16.5.2 Standards and Procedures

Standards and procedures provide the framework and methods for software development. Standards are the established criteria to which the software products are compared. Procedures are the established criteria to which the development and control processes are compared. Standards and procedures should be properly documented and formalized by management approval.

The Management Plan describes the processes for software development and control. The planning activities required to assure compliance of both products and processes with designated standards and procedures are described in the QA portion of the Management Plan. Some important standards for QA are given below.

1. The Design Standard specifies the form and content of the design product. It provides rules and methods for translating the software requirements into the software design and for representing it in the design documentation.
2. The Code Standard specifies the language in which the code is to be written and defines any restrictions on use of language features. It defines desirable language structures, style conventions, rules for data structures and interfaces, and internal code documentation.
3. The Documentation Standard specifies form and content for planning, control, and product documentation and provides consistency throughout a project.

Procedures are the steps that should be followed in carrying out a process. All processes should have documented procedures. There should be documented procedures for all important processes such as

configuration management (CM), nonconformance reporting and corrective action, testing, formal inspections etc.

16.5.3 Quality Assurance Activities

Product evaluation and process monitoring assure that the software development processes as described in the Project Management Plan are carried out correctly. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Product evaluation assures that clear and achievable standards exist and then evaluates compliance of the software product to the established standards. Process monitoring ensures that the processes are being carried out as per prescribed (documented) procedures. The Management Plan should ensure that appropriate QA approval points are built into the processes.

Quality Audit: It is the fundamental QA technique, which looks at a process and/or a product in depth and compares its compliance to established procedures and standards. Quality audits are used to review management, technical and assurance processes to provide an indication of the quality and status of the software product. Its purpose is to assure that proper control procedures are being followed, required documentation is maintained and status reports accurately reflect the status of the activities.

In addition to the above there are some other activities of QA also. These are listed below.

Verification and Validation Monitoring: It assures that the Verification and Validation (V&V) activities such as technical reviews, inspections and walkthroughs are properly conducted and documented. Formal software reviews should be conducted to identify problems and determine whether the interim product meets all requirements. Preliminary Design Review (PDR), Critical Design Review (CDR) and Test Readiness Review (TRR) are some examples of formal reviews.

Formal Test Monitoring: It assures that formal software testing, such as acceptance testing, are done in accordance with plans and procedures. The documents/reports relating to software testing are reviewed for completeness and adherence to standards. The documentation review includes test plans, test specifications, test procedures and test reports. QA monitors testing and provides follow-up on non-conformances. The objectives of QA in monitoring formal software testing are to assure that:

- The test procedures are testing the software requirements in accordance with test plans.
- The test procedures are verifiable.
- The correct version of the software is being tested.
- The test procedures are being followed.
- Non-conformances occurring during testing are noted and recorded.
- Test reports are accurate and complete.
- Resolution of all non-conformances is taken prior to delivery.

CM Monitoring: It assures that software CM activities are performed in accordance with CM plans, standards and procedures.

QA is a system to ensure the quality of product or service. Customers are concerned with QA. However, organizations should think beyond QA. The problem with just maintaining status quo in quality is that the competitors may move ahead. Hence, to cope up with technical advancement, competition and increasing customer expectation, it is necessary not only to maintain quality but also to improve it with time. Hence, quality improvement is now considered as an essential process and a part of the Quality Management System.

16.6 PROCESS IMPROVEMENT AND SIX SIGMA

The modern paradigm for quality management emphasizes on continuous improvement. There are various management initiatives for quality improvement. Some of these are listed below.

- (i) TQM
- (ii) Process Standards (ISO 9001:2000, Capability Maturity Model (CMM))
- (iii) Total Productive Maintenance
- (iv) Lean Systems/Manufacturing
- (v) Six Sigma

The concept of TQM originated from Japan. It improved the quality of Japanese industries to a large extent, due to which TQM became very popular throughout the world. The top management's commitment to quality, customer orientation, continuous improvement and total participation of all people for quality improvement are the core principles of TQM. TQM relies on collective strength of people to generate and implement new ideas for making continuous improvement. The success of TQM thrives on team work, organizational commitment and participative management. Hence, TQM was not very successful in other cultures as it was in Japan.

Many organizations believe that if the processes are systematized and carried out strictly as per standard procedures, it would lead to consistent quality. Accordingly, many standards have evolved. ISO 9000 is a generic process standard whereas ISO 12207 and CMM are two popular standards applicable to software companies.

The concepts of TPM and Lean systems are important concepts for quality improvement. Both these concepts emphasize on reduction of losses. 'Lean' is defined as a systematic approach to identifying and eliminating the wastes in a system. The eight types of wastes in the context of software development/management activities are listed below.

- i) **Waiting:** It refers to a situation when an activity cannot be started or continued because some other related activities are not completed or some other resources are busy in other work. The examples are: system downtime, system response time, approvals from others, information from customers etc.
- ii) **Defects:** It refers to any form of scrap, mistakes, errors or correction resulting from the work not being done correctly the first time, e.g. data input errors, design errors, engineering change orders and invoice errors.
- iii) **Extra Processing:** It refers to having to do anything more than needed, e.g. re-entering data, making extra copies, preparation of unnecessary or excessive reports, monitoring.
- iv) **Inventory:** It refers to any supply that is in excess, e.g. extra office supplies, sales literature, batch processing transactions.
- v) **Excessive Motion:** It refers to unnecessary movement of people, e.g. movement of people to do photocopy, to file documents, to send and receive fax or to do minor discussion with other co-workers.
- vi) **Transportation:** It refers to movement of work or paperwork from one step to the next step in the process, e.g. movement of documents from one desk to another, from one office to another, too much travel/tour by employees.
- vii) **Overproduction:** It refers to producing more, and sooner than is required by the next person, e.g. printing more copies than required, purchasing items before they are needed, doing a work beyond the scope of project.

- viii) **Under-utilized Employees:** It refers to not utilizing the creativity, ideas and abilities of people to their full potential, e.g. not empowering the people to take up responsibilities and to do basic tasks, idle employees, high-skilled employees engaged in low-skill jobs.

The organization should identify and try to eliminate these wastes and low value added activities through continuous improvement.

Of late the concept of ‘Six Sigma’ has become very popular for process improvement. It is now a major influence on production methods and QA. Six sigma is a data- and statistical-driven approach to eliminate defects in production. It aims to improve processes and reduce variations in quality. It necessitates planning, training and organizational change.

‘Defect’ may be defined as any product or service that does not conform to the set standards or to the satisfaction of the customer. Defects are inherent in a process. However, if the process is improved, the occurrence of defects or errors can be reduced even to near-zero level. The Greek letter for Sigma (σ) represents one standard deviation from the mean or average. The higher the sigma level of process, lesser is the percentage of defects in output and better is the quality.

| Sigma Level | Percent of defects in output |
|-------------|------------------------------|
| 1 Sigma | 69.1% |
| 2 Sigma | 30.8% |
| 3 Sigma | 6.7% |
| 4 Sigma | 0.6% |
| 5 Sigma | 0.03% |
| 6 Sigma | 0.00034% |

Six Sigma is a concept that tries to achieve a quality level where percentage of defects is less than 0.00034 or 3.4 Defects Per Million Opportunities (DPMO).

Six Sigma was pioneered in the US by Bill Smith at Motorola in 1986. It was originally used as a metric for measuring defects for improving quality. Later it was developed as a methodology to reduce defects to a level of 3.4 DPMO. Motorola has reported significant benefits in terms of quality improvement and cost reduction as of 2006.

As a **business improvement** methodology it focuses on:

- Understanding and managing customer requirements
- Aligning key business processes to achieve those requirements
- Utilizing rigorous data analysis to minimize variations in those processes
- Driving rapid and sustainable improvement to business processes

Motorola developed a five-step Six Sigma methodology for process improvement, called DMAIC. The five steps of DMAIC are: (1) Define, (2) Measure, (3) Analyze, (4) Improve and (5) Control. The steps are described below.

Define: This step involves identifying ‘who the customers are and what their requirements are for the products and services’. The expectations of customers determine the priority that should be given to different areas for improvement. It serves as a guide to define the goals, objectives and project boundaries; i.e. to define what will be the focus for improvement. It involves mapping of processes and identifying the key determinants of performance.

Measure: This step involves collection of data about various aspects of processes such as key performance matrices, types of defect and how frequently they occur, customer feedback on how processes fit

their needs, speed in responding to customer requests etc. Six Sigma methodology, stresses on elimination of guesswork and assumptions.

Analyze: Data analysis involves organization of data and looking for process problems and opportunities. The analysis of data helps to identify sources of variation and root causes of problems in the process. Hence, this step helps to identify gaps between current performance and the goal; and to prioritize opportunities for improvement.

Improve: This step involves finding solutions by correcting root causes. It requires innovation, technology and discipline.

Control: The findings and solutions determined in earlier steps are implemented and standardized. The processes are documented and monitored through statistical process control. After a 'settling in' period, the process capability is reassessed to ensure if improvement has really taken place.

These steps are carried out sequentially to bring about improvement. After one cycle is over the steps are repeated iteratively to reach desired quality level. The DMAIC methodology is depicted in Figure 16.1.

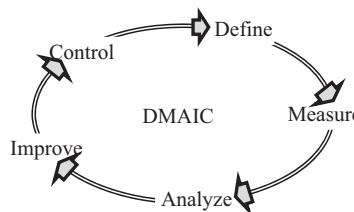


Figure 16.1 DMAIC methodology for Six Sigma

Six sigma uses various statistical tools for data analysis. Some of the important tools are given below.

1. **Quality Function Deployment (QFD):** QFD is a system and set of procedures to identify, communicate and prioritize customer requirements.
2. **Cause and Effect Matrix:** It is a tool that helps Six Sigma teams select, prioritize and analyze the data they collect over the course of a project to identify problems in that process.
3. **Failure Modes and Effects Analysis (FMEA):** FMEA helps Six Sigma teams to identify and address weaknesses in a product or process, before they occur.
4. **T-Test:** It is used to determine the statistical difference between two groups, not just a difference due to random chance.
5. **Control Charts:** The Six Sigma teams use Control Charts to assess process stability.
6. **Design of Experiments (DOE):** DOE is a planned way of doing experiments to determine the effect of various inputs and process parameters on output.

The Six Sigma methodology relies heavily on advanced statistical methods that complement and reduce process and product variations. It is a topdown approach. It is driven by professionals who have expert knowledge of processes and statistical methods. The Bank of America, Caterpillar, Honeywell International, Raytheon, Merrill Lynch and General Electric are some of the early adopters of Six Sigma methodology. The six sigma methodology has been successful in the US, Western countries and India.

16.7 PROCESS STANDARD: ISO 9000

Consistency in quality of a product requires consistency in quality of inputs and processes. Thus, it is essential that an industry follows sound procedures, methods and controls, applicable to all its operations with regards to quality. Such procedures, methods and control constitute their individual Quality Management System. To ensure consistent quality, buyers/clients generally try to assess if their suppliers have the capability to meet the quality requirement or not. Hence, a universally accepted quality standards, which could serve as a benchmark for the assessment of any supplier's quality system, was deemed necessary. To fulfil this need, the ISO issued the ISO 9000 series of standards for QA systems.

The Structure of ISO 9000 Standards: ISO 9000 series of standards is a quality-system standard. It is somewhat different from traditional 'product-quality standard'. The product-quality standard specifies various characteristics or parameters that the product must meet if it is to conform to the product standard. Product certification authorizes a manufacturer to use the prescribed certification stamp on a product. Conformity of a product to its specification is ascertained by the certification body through periodic testing of product samples.

However, the quality-system standard defines the method of managing quality in a company to ensure that products conform to the quality level of the company. A company is free to set any quality standards for its product on the basis of marketing strategies and customer requirements. The quality system of a company is designed and evaluated based on following principles:

Say what you do. A company must state its mission and quality objectives. It must prepare a detailed quality manual and state the procedures for all its processes. And then it can be seen how its processes are capable of meeting the quality objectives.

Do what you say. All the work in the company should be carried out as per procedure laid down in the quality manual.

Prove it. The relevant records must be maintained so that it can be verified whether the work in the company are being carried out as per documented procedure or not.

A company can get ISO 9000 certification if it can demonstrate that its processes are complying with all the relevant clauses of the ISO standards.

A quality-system standard helps the company to plan and achieve the requisite product quality consistently. Since ISO 9000 defines a system for managing quality in generic terms, it applies universally to all products and services. This standard is equally relevant to companies manufacturing different types of products as well as organizations in various service sectors such as hotels, hospitals, airlines, education, electricity, information technology and telecommunications services. The standards can be used by large, medium as well as small-scale companies. An ever-increasing number of companies all over the world are implementing the quality-system standard. The bigger companies are insisting that their suppliers implement certified quality systems. The ISO 9000 certification gives credibility to the quality system of companies in the international markets. Hence, ISO 9000 certification is useful to software companies.

TQM Compared to ISO 9000 Quality Standards: The concept and principles of TQM to a large extent are embodied in the ISO 9000 series of quality standard. ISO 9000 standard gives emphasis on processes. It covers a wide range of management activities and functions. In this respect, it takes a broader view for managing quality. Where there is an ISO system, many of the steps are in place for TQM. The ISO quality standard sets in place a system to deploy quality policies to achieve verifiable objectives. However, the ISO standard addresses only the system-related aspects of TQM, whereas the behavioral aspects of TQM are also equally important.

The Quality Leadership: The top management has a major role in TQM in providing leadership in quality. They must show commitment in communication and actions. Their leadership style should support improvement of processes, improvement of employees through training, use of quality tools in decision-making, employee participation, involvement and teamwork.

The Attitude: Quality results from the right attitude of people. TQM recognizes that managing quality is everyone's responsibility. It is the joint responsibility of managers and workers to create proper work environment for quality. The task for senior managers is in changing people's attitudes and culture. It is not just about introducing new concepts, techniques and methodologies but is about changing attitudes towards doing business. Customer orientation is a mindset, in recognizing that the customers set the product features, quality requirements and the standards of competitiveness. TQM is a philosophy of continuous improvement that also involves a positive attitude to changes.

The above behavioral aspects constitute the concepts of TQM. Since ISO 9000 series of quality standards do not adequately address these aspects, implementation of ISO standard cannot be equated with implantation of TQM. This may explain why in some cases implementation of ISO 9000 standards has not resulted in improvement of quality. TQM requires changing the mindset of employees, especially of the top management.

16.8 PROCESS STANDARD: ISO 12207

ISO 9000 is a generic process standard. It is applicable to any type of industries. Beside this ISO has also developed a standard for software lifecycle processes. ISO 12207 is a standard developed by ISO for the purpose. The standard aims to define all the tasks required for developing and maintaining software.

The standard establishes the architecture for the software life cycle with a set of processes and inter-relationships among these processes. The standard is based on two basic principles: (1) Modularity and (2) Responsibility.

Modularity: It means that the processes are cohesive and least coupled to the extent feasible. An individual process should be dedicated to a single function.

Responsibility: It means that for each process, responsibilities of different parties/people involved are clearly specified. This facilitates the application of standards in projects where many people can be legally involved.

The processes are classified into three types:

1. Primary Processes
2. Support Processes
3. Organizational Processes

16.8.1 Primary Processes

The standard specifies five basic or primary processes. These are:

1. Acquisition Process
2. Supply Process
3. Development Process
4. Operation Process
5. Maintenance Process

Each phase within the primary life cycle processes can be divided into different activities and each activity consists of a number of tasks.

Acquisition Process: Acquisition covers the activities involved in initiating a project. The acquisition phase can be divided into different activities and deliverables. Some important activities are given below.

| Activities | Tasks | Deliverables |
|---|--|---|
| Initiation | Description of need (i.e. why to acquire, develop or enhance software product), Approval of System requirements, Evaluation of options (i.e. develop or buy), Development of an acquisition plan, Development of acceptance criteria | Initiation documents |
| Request for proposal preparation | Description of Acquisition requirements, System requirements and technical constraints, Outlining the process for the project, Demarcation of contract milestones for progress review and audit | Request for proposal |
| Preparation of Contract | Outlining the selection procedure for suppliers, Selection of suppliers | Contract, i.e. a draft agreement, between the company and suppliers |
| Negotiation for changes and Update contract | Negotiations with the selected suppliers Updated contract with the result from the negotiations | The Final Contract |
| Supplier monitoring | Monitoring the activities of the suppliers as per agreements, Collaboration with suppliers to ensure timely delivery | Supplier Monitor Report |
| Acceptance and completion | Outlining of acceptance test procedures, Testing and acceptance of products delivered by the suppliers, CM on the delivered product | Acquisition Report |

Supply Process: During the supply phase a project management plan is developed. This plan contains information about the project such as different milestones that need to be reached. This project management plan is used in the next phase, which is the development phase.

Development Process: During the development phase the software product is designed, created and tested and will result in a software product ready to be sold to the customer. The standard provides guidelines about activities to be performed and their deliverables for the development process. The important deliverables of development process are listed below:

- Software Requirements: this is a collection of different software requirements;
- High-level design: A basic layout of the software product is created. This means the setup of different modules and how they communicate with each other. This contains the outline design and not the details about the modules.
- Module design: This specifies the details about different modules present in the high-level design.
- Program Code: This is created according to the high-level design and the module design.
- Module test report: This contains the test results of software codes of individual modules.
- Integration test report: This contains the test results after different modules are integrated with each other.
- System test report: This is the final test report for acceptance of the system.

The standard prescribes that activities should be planned before their execution. The outcome of activities should also be properly documented and evaluated. The standard prescribes the following generic acceptance criteria for the evaluation:

- Traceability to source document
- Consistency with earlier products
- Internal consistency
- Testability and test coverage
- Conformance to expected results
- Appropriate standards and methods
- Feasibility of next activity
- Operational feasibility
- Maintainability

The activities need not be performed in an exactly sequential manner and there can be overlap of activities. The standard does not prescribe any particular methodology or life cycle model. The life cycle model can be Waterfall model, Incremental model or Evolutionary model.

Operation: The operation and maintenance phases occur simultaneously, the operation phase consists of activities like assisting users in operating the software product.

Maintenance: The maintenance phase consists of maintenance tasks to keep the product up and running. Maintenance includes any general enhancements, changes and additions, which might be required by the end-users.

16.8.2 Support and Organizational Processes

A supporting process supports other processes and contributes to the success and quality of the project. ISO 12207 standard contains a set of eight supporting processes. These are: (1) Documentation Process, (2) CM Process, (3) QA Process, (4) Verification Process, (5) Validation Process, (6) Joint Review Process (7) Audit Process and (8) Problem Resolution Process.

The organization process helps in establishing, controlling and improving other processes. It operates at the organizational, corporate level, typically beyond or across projects. An organizational process may support any other process as well. The ISO 12207 standard contains a set of four organizational processes. These are: (1) Management Process, (2) Infrastructure Process, (3) Improvement Process and (4) Training Process.

The support and organizational processes must exist independently of the project being executed. The set of processes, activities and tasks can be adapted according to the software project.

ISO 12207 is an international standard that provides a complete set of processes for acquiring and supplying software products and services. These processes may be employed also to manage, engineer, use and improve software throughout its life cycle. Its architecture can accommodate traditional as well as modern software methods, techniques, tools and engineering environments.

16.9 CAPABILITY MATURITY MODEL

The Capability Maturity Model (CMM) is a framework to assess an organization's capability. This model is most applicable to software development. CMM covers practices for planning, engineering and managing software development and maintenance. It describes the key elements of effective

software processes with an objective to gradually bring about improvement in them. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality and product quality. CMM establishes a yardstick to measure the maturity of an organization's processes. It specifies five levels of maturity as given below.

Level 1 – Initial State: An organization is at CMM level-1 if its processes are ad hoc, disordered and may be even occasionally chaotic. Few processes are defined, and success depends on individual effort and chance factor. It signifies that the organization does not have any system in place to ensure successful completion of software projects.

Level 2 – Repeatable State: An organization is at CMM level-2 if its basic project management processes are established to track cost, schedule and functionality. It signifies that the organization has some system or process discipline in place to repeat earlier successes on projects with similar applications.

Level 3 – Defined State: An organization is at CMM level-3 if its processes for both management and engineering activities are documented, standardized and integrated into a system. It signifies that all activities for developing and maintaining software in the organization are carried out as per documented/approved procedures.

Level 4 – Managed State: An organization is at CMM level-4 if all its processes and work products are periodically measured and the data are recorded and analyzed to ensure quality. It signifies that attributes of both processes and products are quantitatively understood and controlled.

Level 5 – Optimized State: An organization is at CMM level-5 if continuous process improvement is enabled by quantitative feedback from the processes. It signifies that there is a system to bring about improvement in the organization through new ideas and technologies.

Process Area (PA): For each maturity level, different PAs are defined as given below.

| CMM Level | Process Area |
|-----------------------------|---|
| Level 1 – Initial State: | No PA is defined |
| Level 2 – Repeatable State: | Software CM Software QA Software subcontract management Software project tracking Software project planning Requirement management |
| Level 3 – Defined State: | Peer review, intergroup coordination Software product engineering Training programs Organization process |
| Level 4 – Managed State: | Software quality management Quantitative process management |
| Level 5 – Optimized State: | Process change management Technology change management Defect prevention |

A PA contains the goals that must be reached in order to improve a software process. A PA is said to be satisfied when procedures are in place to reach the corresponding goals. When an organization has achieved a specific maturity level, all the corresponding PAs are satisfied. For example, CMM level-5 has three PAs defined as under.

1. Process change management: To identify the causes of defects and prevent them from recurring.
2. Technology change management: To identify new technology that are beneficial and transfer them in an orderly manner.
3. Defect prevention: To continually improve the processes to improve quality, productivity and development time.

The defined standards give the organization a commitment to perform because:

- The organization follows a documented policy for implementing software process improvements.
- Senior management sponsor the organization's activities for software process improvement.

Organizations strive to improve the capability of its processes to reach CMM level-5. At the CMM level-5 (optimized level), the quantitative feedback from previous projects is used to make improvements in existing projects. The focus is on continuous process improvement. The project teams analyze defects to determine their causes. Software processes are evaluated to prevent known types of defects from recurring, and lessons learned are disseminated to other projects. The software process capability of level-5 organizations can be characterized as continuously improving. Improvement occurs both by incremental advancements in the existing process and by innovations using new technologies and methods.

SUMMARY

ISO has defined quality as 'Totality of features and characteristics of product/service that has ability to satisfy stated and implied needs of customer'. The product/service should meet basic quality attributes as well as specific requirements of customers. Correctness, reliability, efficiency, ease of use, accuracy, portability, control and audit, maintainability, and expandability are some attributes of software quality. Earlier the emphasis for achieving quality was on thorough testing of the final product. The modern concept of quality management focuses not only on quality of the output but also on the inputs, processes and environment.

Edward W Deming, Joseph M Juran, Philip B Crosby, A V Feigenbaum, K Ishikawa are some pioneers in quality management, who have contributed immensely to this field. The concepts of Random and Assignable causes of error, PDCA cycle, Quality Trilogy, CoQ, Factory within Factory and Zero Defects, and the overall concepts of TQM can be attributed to a great extent to them.

The modern concepts of quality management give much emphasis on quality of the process. The evaluation criteria of Deming Prize, MBNQA Model, EFQM Business Excellence Model and other such quality awards instituted by different countries may be regarded as good models of process quality.

QA is a system to ensure quality of the product or service. QA emphasizes not only on early detection and correction of errors but also on preventing the occurrence of errors through proper documentation, data analysis and administrative measures.

Just maintaining the status quo in quality is not enough. The improvement of quality is equally important. TQM, Process Standards (ISO 9001:2000, CMM), Total Productive Maintenance, Lean Systems/Manufacturing and Six Sigma are popular management initiatives for quality improvement.

Six Sigma is a concept that tries to achieve a quality level where percentage of defects is less than 0.00034 or 3.4 DPMO. The Six Sigma methodology (known by acronym as DMAIC) consists of five steps: (1) Define, (2) Measure, (3) Analyze, (4) Improve and (5) Control.

ISO 9000 series of standards is a quality-system standard. It defines the method of managing quality. It prescribes that a company must bring out quality manual (Say what you do). Then all the work in the

company should be carried out as per procedure laid down in the quality manual (Do what you say). The company must maintain the relevant records so that it can be verified (Prove it).

ISO 12207 is a standard for software lifecycle processes. The standard is based on two basic principles: (1) Modularity and (2) Responsibility. The processes are classified into Primary Processes, Support Processes and Organizational Processes. The standard specifies five primary processes: (1) Acquisition Process, (2) Supply Process, (3) Development Process, (4) Operation Process and (5) Maintenance Process. A supporting process supports other processes. The organization process helps in establishing, controlling and improving other processes. The support and organizational processes must exist independently of the project being executed.

CMM is a framework to assess an organization's capability. It specifies five levels of maturity: Level 1 – Initial State, Level 2 – Repeatable State, Level 3 – Defined State, Level 4 – Managed State and Level 5 – Optimized State. Organizations strive to improve the capability of their processes to reach CMM level-5. At CMM level-5 (optimized level), the quantitative feedback from previous projects is used to make improvements in existing projects.

EXERCISES

1. Define Quality. What are the various attributes of software quality?
2. Distinguish between Inspection, QC and QA.
3. Distinguish between random causes and assignable causes of error.
4. What is the PDCA cycle?
5. According to Juran, what are the three processes of 'quality trilogy'?
6. Explain the concept of CoQ.
7. According to Crosby, what is meant by the term 'factory within factory'?
8. What is process quality? How is it different from product quality?
9. What is quality standard? Distinguish between product standard and process standard.
10. List the evaluation criteria of process quality according to Baldrige model.
11. List the four elements for evaluating result (performance) in the EFQM model.
12. Define QA. Describe some activities of QA.
13. List various management initiatives/concepts for process improvement.
14. Enumerate the eight types of wastes in the context of Software development activities.
15. Describe the five steps of Six Sigma methodology.
16. Discuss the application of statistical tools in quality management.
17. Discuss the concepts of TQM. Can it be exactly equated with ISO 9000 standard? Explain.
18. What are various processes of the ISO 12207 standard?
19. What is the meaning of capability maturity levels according to CMM?

WEB ENGINEERING

During the last decade there has been phenomenal growth in Web-based systems/applications. The requirements of Web-based systems are somewhat different. Hence, to meet the special requirements of these systems, some specific processes, procedures and design principles have been developed. This has led to emergence of new sub-discipline in software engineering called 'Web Engineering'. The following aspects of Web Engineering are discussed in this chapter.

- General Web Characteristics
- Web Engineering Process
- Web Design Principle
- Web Metrics
- Mobile Web Engineering
- Security Issues

The Internet and World Wide Web (www) have developed very recently. Internet is the major technical infrastructure that allows people throughout the world to have connectivity with one another and with any other computer system. It is defined as a network of computer networks. Computer networks are interconnected by dedicated, special-purpose computers called routers. The World Wide Web or simply the Web is the collection of information that is available to the users at all times through the Internet. The information is stored on various servers that are interconnected with each other and located at different locations throughout the world. With the rapid growth of the Web, significant developments have not only taken place in hardware, software and computer networking, but have also led to emergence of a new field called 'Web Engineering'.

17.1 GENERAL WEB CHARACTERISTICS

The Web is a collection of interlinked documents that is stored in electronic form on the Internet for use of the general public or some authorized users. It can be described as a system for delivering hypermedia over networks using a client/server model. It offers many possibilities for disseminating information by

using a variety of data formats such as text, graphics, sound, movies and animation. The methodology for developing a Web-based system requires a continuous, process-oriented approach focused on meeting users' needs. The first step is to understand the characteristics of the Web as a medium for information sharing.

17.1.1 Evolution of Web Software

The history of the Web software can be linked to two important innovations:

1. Hypertext Document
2. Graphical User Interfaces (GUIs) for hypertext

Vannevar Bush, who was director of the U.S. Office of the Scientific Research and Development, wrote an article in 'The Atlantic Monthly' in 1945 about how scientists can apply the skills they have acquired during World War II in various peacetime activities. That article contained a number of futurist ideas about future uses of technology to organize information in an efficient manner and subsequent access to that information. Bush speculated that engineers would eventually build a machine, which he called the Memex, a memory extension device that would store all the contents of books, records, letters and research results on microfilm. The Memex would include indexes that would help users quickly to find out a certain article and a microfilm reader to read that information.

In the 1960s, Ted Nelson described a similar system in which text on one page links to text on other pages. Nelson called his page linking system as hypertext. Douglas Engelbart, who also invented the computer mouse, created the first experimental hypertext system on one of the big computers of the 1960s. In 1987, Nelson published *Literary Machines*, a book in which he outlined project Xanadu, a global system for online hypertext publishing. He used the term hypertext to describe a large global page linking system that would interconnect related pages of information, regardless of where in the world they were stored. This was the original version of hypertext.

In 1960, the US defence sponsored a research called 'Advanced Research Projects Agency (ARPA)' to do research on networking. The main objective of ARPA was to interconnect Local Area Networks (LANs) and Wide Area Networks (WANs). The project was successful in finding the solution to the problem of connecting different networks. This led to the creation of a big WAN called ARPANET that connected various LANs and WANs. Subsequently, the ARPANET became a key part of the Internet.

In 1989, Tim Berner-Lee was trying to improve the laboratory research document handling procedures for his laboratory on particle physics. The laboratory had been connected to the Internet for two years, but its scientists wanted to find better procedure to circulate their research papers and data. To achieve this objective, Tim Berners-Lee proposed a project on hypertext development intended to facilitate sharing and distribution of data and functionality.

In 1991, Tim Berners-Lee developed the code for a hypertext server program and made it available on the Internet. A large computer connected to the Internet contains that program and other files written in the hypertext markup language (HTML). It enables other computers connected to the Internet to read these files. Hypertext servers used on the Web are called as Web servers today. The language that Berners-Lee developed from his original hypertext server program is called HTML. This language includes a set of codes or tags attached to the text. This language is the basic and the first Web language ever produced. HTML includes tags that indicate which text is a part of the header element, which text is apart of a paragraph element, which text is a part of a numbered list element. Another important type of tag is the hypertext link tag. This hypertext link or hyperlink points to another location in the same

or another HTML document. Tim Berners-Lee called his system of hyperlinked HTML documents the ‘World Wide Web’ or simply the ‘Web’. Since program codes and tags are mostly written in lower case, the World Wide Web is denoted as ‘www’. The Web became popular in the scientific research community, but few people outside that community had the software needed to read the HTML documents. In 1993, a group of students led by Marc Andreessen at the University of Illinois wrote Mosaic, the first GUI program that could read HTML documents and use HTML hyperlinks to navigate from one page to another. Mosaic was the first Web browser for personal computers.

In 1994, Andreessen and other members of the Mosaic team in the University of Illinois joined with James Clarke of Silicon Graphics to establish Netscape communications. Its first product, the Netscape Navigator, the Web browser program based on Mosaic, was an instant success. Netscape became one of the fastest growing software companies. The success of Netscape prompted Microsoft to create its Web browser called Internet Explorer. In addition to these two, a number of other Web browser suites have also been developed by various companies. Mozilla Firefox is a free and open source Web browser. It has come from the Mozilla Application Suite, managed by Mozilla Corporation.

A Web browser presents an HTML document in its GUI. All personal computers today use the GUI of a Web browser such as Netscape Navigator or Microsoft Internet Explorer to read HTML documents. A Web browser is a software interface that lets users browse HTML documents and move from one HTML document to another using hypertext link tags. By using the Web browser, we can also move from one HTML document to another on a different computer through the Internet using such hyperlink tags.

To access an HTML page, a Universal Resource Locator (URL) needs to be written on the address bar of the browser. Every website and every page of a website has a unique address. For example, website of IIT Delhi has the address ‘<http://www.iitd.ac.in/>’. This website has many Web pages. For example, ‘<http://www.iitd.ac.in/about>’ is one of the Web pages of this website. The various Web pages are hyperlinked for easy navigation.

Since its inception, Internet technology has developed very fast. In 1996, a group of network research scientists from nearly 200 universities and a number of major corporations joined together to rebuild the Internet. They came up with an advanced research network called Internet2. Internet2 is used by universities to conduct large collaborative research projects that require several supercomputers connected at high speed. It could use multiple video feeds that was not possible earlier. It built the ground for new technologies and applications on the Internet. The Internet2 project is focused mainly on technology development.

Thus, a new standard has evolved in Web technologies. It is called Web 2.0. The Web 2.0 standard allows highly interactive websites, where one can post data and view it. This has led to the emergence of social networking websites such as flicker, facebook, youtube etc. The concepts of Asynchronous JavaScript and XML (AJAX) are derived from Web 2.0.

New technologies and Web publishing behaviors are emerging at a very fast rate. The Web pages are now more dynamic. Some of the latest Web characteristics include social networking, video sharing like youtube, blogging, Really Simple Syndication (RSS) feed etc. Further research is going on to build Intelligence into the Web.

17.1.2 Emergence of Web Engineering

During the last decade the Web has undergone great transformation. Hence, there is paradigm shift in how websites are designed. The new paradigm is shown in Table 17.1.

The above developments have led to emergence of a new sub-discipline in software engineering called ‘Web Engineering’.

Table 17.1 Paradigm Shift in Website Design

| Earlier Web-Based System | Present day Web-Based System |
|---|---|
| Easy to create, does not require a proper development process | Requires a sound development process and methodology |
| Website development is quite simple, does not require a proper project plan | Necessitates project plan and management |
| Does not require a proper management of website | Needs configuration control and management |
| Primarily textual information in noncore applications | Dynamic Web pages because information changes with time and users' needs |
| Information content fairly static | Large volume of information |
| Simple navigation | Navigation in website is quite complex |
| Infrequent access or limited usefulness | Integrated with database and other planning, scheduling and tracking systems |
| Limited interactivity and functionality | Deployed in mission-critical applications |
| Stand alone systems | Prepared for seamless evolution |
| High performance not a major requirement | High performance and continuous availability is a necessity |
| Developed by a single individual or by a very small team | May require a larger development team with expertise in diverse areas |
| Security requirements minimal (because of mainly one-way flow of information) | Calls for risk or security assessment because there is both-way flow of information |
| Feedback from users either unnecessary or not sought | User satisfaction is vital |
| Website mainly as an 'identity' for the current clientele and not as a medium for communication | Website acts as an important and main communication medium between the organization and users |

17.2 WEB ENGINEERING PROCESS

Web Engineering is a systematic approach that uses scientific engineering and management principles to develop, deploy and maintain high-quality Web-based systems and applications.

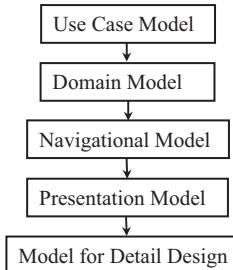
The Web Engineering process uses certain methodologies, procedures, principles, tools and techniques to develop large and complex websites, to fulfil desired requirements. Engineering is about systematic, disciplined and quantifiable approaches to create usable systems. Like software engineering, Web Engineering also follows various phases of development, e.g. requirement elicitation, design, testing etc. However, there are some differences between software engineering and Web Engineering as shown in Table 17.2.

The Object-Oriented Web Solution (OOWS), Object-Oriented Hypermedia Design Model (OOHDM), Web Modeling Language (WebML) and Open Space Framework are some methodologies of the Web Engineering process. However, these methodologies are still evolving.

Web Engineering being a relatively new field, there is some disagreement on a common Web development process. However, artifacts and activities that are followed in most of the methodologies are shown in Figure 17.1. This may be regarded as a simplified Web Engineering process. It is based on the Model-Driven Engineering paradigm.

Table 17.2 Characteristics of Software Engineering and Web Engineering

| Characteristics | Software Engineering | Web Engineering |
|---|--|--|
| User range | Small | Large |
| Number of simultaneous users | Less | More |
| User requirements | Specific | Changes fast |
| Types of users | Mostly internal | Internal as well as external |
| Growth and change | Slow | Fast |
| Hardware and software environment constraints | Specific (more clear during the course of development) | Unknown (should cater for all possible combinations) |
| Adherence to standards and protocols | Not very important | Very important |
| Security and legal issues | Not very important | Very important |
| Look and feel of the final product | Performance is important; look and feel are secondary | Look and feel are equally important |

**Figure 17.1** A simple web design process

The Web Engineering process has the following three major workflows:

1. Analysis workflow
2. Conceptual design workflow
3. Detailed design workflow

The use-case diagram results in an analysis workflow. The output of this workflow is a domain model in the form of an Entity Relation (ER) diagram or a Unified Modelling Language (UML) class diagram. The navigation and presentation model are the output of conceptual design workflow. These are expressed by UML profiles. Finally, a detailed design workflow introduces platform- and technology-specific features such as J2EE and .NET. These are depicted through the deployment model.

OOWS is one of the earlier processes used in Web Engineering. It consists of the use-case model, navigational and presentation model. The other process, OOHDM consists of domain analysis, navigation, abstract interface design and implementation. The WebML methodology is similar to OOHDM. WebML stands for Web Modeling Language. The hypertext design is a major activity of WebML. It mainly consists of navigational design, interface design and layout design. The WebML development methodology is depicted in Figure 17.2.

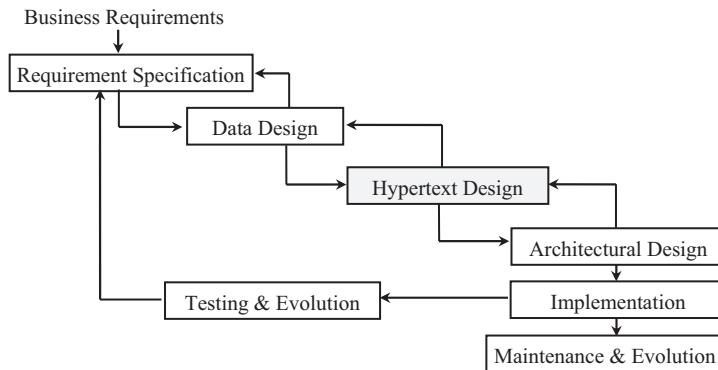


Figure 17.2 WebML development methodology

The model-driven approach can be an effective way to transform the Web requirements into Web system design. In this approach, the content model, the navigation model, the process model and the presentation model are parts of the platform-independent design models. The platform-independent design models are later adapted for platform-specific model such as .NET or J2EE.

17.3 WEB DESIGN PRINCIPLES

The website design process always starts with a detailed analysis of requirements and development of appropriate specification. A prototype is constructed from the design. The prototype usually contains a set of sample pages. These can be used for evaluating the screen layout and navigation between different pages. Based on the feedback from the stakeholders, either the design or the specifications are changed. One can iterate through this process until stakeholders are happy with the screen layout and the navigation structure. Figure 17.3 shows various inputs to the design process and the typical outcomes from this process.

The stakeholder requirements are developed at the start of the development process and requirement specification is given in detail. The Web designer should also take into account the non-technical aspects such as legal, moral and cultural issues that are relevant to the environment in which this application will

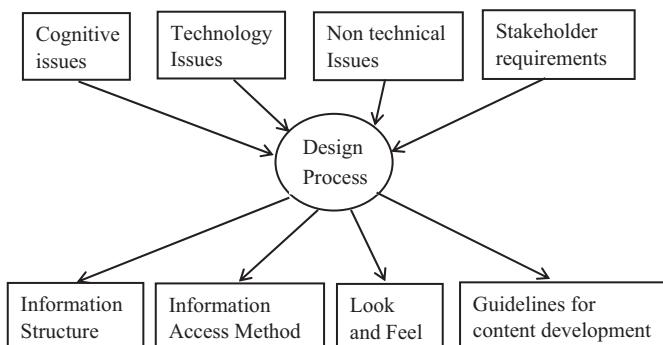


Figure 17.3 Web site design process

be used. The knowledge of users' cognitive issues is also important for developing a good design. The designer should know how users would perceive and comprehend information and how the style of fonts, color and layout can contribute toward this. Technical feasibility is also an important aspect. For example, use of very large graphics or video may not be possible due to limitations of the bandwidth. It is important for the designers to be aware of these technological issues.

The outcomes of the design process consist of information structure, information access methods, various screen layouts or look-and-feel and guidelines for new content development or processing legacy material if the content is derived from a legacy database or other systems. Irrespective of how the contents are derived, we need to create a structure to organize this content. The content structure depends on factors such as nature of the application, nature of the information and what technology will be used to store the contents. This will also determine the granularity of information that can be accessed. For example, the contents can be stored as (Extensible Markup Language) XML files or a database. Hence, the designer needs to sub-divide the contents into small sub-sections, e.g. title, sub-headings, document author, keywords etc. and provide access to the contents based on these sub-sections. Once a structure for organizing the information is developed, the navigation mechanisms need to be decided in order to access that content. Access to information can be provided by hyperlinks or search facility.

If new content needs to be developed, it is better to come up with guidelines to assist the content development process as part of the design process. If content is drawn from a legacy system we need to develop a process to convert the legacy information to the required structure and format. The output of the design process and the original specifications that were developed at the start of the website development process will form the input to the next stage, i.e. the website construction process.

As the Web contents need to be accessed through hyperlinks, the Web design phase is named as hypertext design phase as mentioned in the WebML process. However, as stated earlier, its main component is the navigational design model. The navigation model defines how the user can browse through the Web application. Nodes define atomic information units and reshape the elements in the information model to specify the actual information chunks. Clusters link together sets of nodes and define how the user can move around these elements. Clusters can be further organized into the following types of clusters:

- | | |
|---------------------------|---|
| 1. Structural Clusters | All elements come from the same entity |
| 2. Association Clusters | They render associations |
| 3. Collection Clusters | The describe the topology of a collection |
| 4. Transactional Clusters | Set of nodes that should be traversed to complete a transaction |

17.4 WEB METRICS

Software metrics are very useful for software projects. However, these software metrics may not be quite suitable for Web applications. Hence, metrics have been developed to suit the specific needs of Web application projects. Web metrics are a useful measure of the success of Web Engineering projects.

Size is often a major determinant of effort. It is usually described in terms of length, functionality and complexity. Most effort prediction models concentrate on measurement of size based on functionality. However, there are other aspects as well. Some of these are:

- Length size metrics
- Reusability metrics

- Complexity size metrics
- Effort metrics etc.

Description of length size metrics, reusability metrics, complexity size metrics, effort metrics are given in Table 17.3, Table 17.4, Table 17.5 and Table 17.6, respectively.

Table 17.3 Length Size Metrics

| Entity Type | Metric | Description |
|-------------|------------------------|--|
| Application | Page Count | Number of static HTML files |
| | Media Count | Number of media files |
| | Program Count | Number of Common Gateway Interface (CGI) scripts, JavaScript files, Java applets |
| | Total Page Allocation | Total space allocated for static pages |
| | Total Media Allocation | Total space allocated for media files in the static pages |
| | Total Code Length | Number of lines of code (LOCs) for all programs |
| Page | Page Allocation | Size of static HTML files |
| Media | Media Duration | Duration of audio, video and animation |
| | Media Allocation | Size of media files |
| Program | Code Length | Number of LOCs of the program |
| | Code Comment Length | Number of comment lines of program |

Table 17.4 Reusability Metrics

| Entity Type | Metric | Description |
|-------------|-------------------------------|---|
| Application | Reused Media Count | Number of reused or modified media files |
| | Reused Program Count | Number of reused or modified programs |
| | Total Reused Media Allocation | Total space allocated for all reused media files used in an application |
| | Total Reused Code Length | Total number of LOCs for all programs reused by an application |
| | Reused Code Length | Number of reused LOCs |
| Page | Reused Comment Length | Number of reused comment lines |

Table 17.5 Complexity and Size Metrics

| Entity Type | Metric | Description |
|-----------------------------|-----------------------|---|
| Application generated links | Connectivity | Total number of internal links, not including dynamically |
| | Connectivity Density | Connectivity divided by page count |
| | Total Page Complexity | $\frac{\sum_i \text{Page Complexity}}{\text{Page Count}}$ |

| | | |
|------|--------------------------|---|
| Page | Cyclomatic Complexity | Connectivity- page count +2 |
| | Structure | Measurement of organization of the application's main structure: sequence, hierarchy or network |
| | Page Linking Complexity | Number of links per page |
| | Page Complexity | Number of different types of media used on a page not including text |
| | Graphic Complexity | Number of graphics media used in a page |
| | Audio Complexity | Number of audio media used in a page |
| | Video Complexity | Number of video media used in a page |
| | Animation Complexity | Number of animation media used in a page |
| | Scanned Image Complexity | Number of scanned images used in a page |

Table 17.6 Effort Metrics

| Entity Type | Metric | Description |
|--|-------------------------|---|
| Application authoring and design tasks | Structuring Effort | Estimated elapsed time taken to structure the application in hours |
| | Interlinking Effort | Estimated elapsed time taken to interlink pages to build an application structure in hours |
| | Interface Planning | Estimated elapsed time taken to plan the application's interface in hours |
| | Interface Building | Estimated elapsed time taken to implement the application's interface in hours |
| | Link Testing Effort | Estimated elapsed time taken in hours to test all links in the application |
| | Media Testing Effort | Estimated elapsed time taken in hours to test all media in the application |
| | Total Effort | Structuring effort + interlinking effort + interface planning + interface building + link-testing effort + media-testing effort |
| Page authoring | Text Effort | Estimated elapsed time taken in hours to author or reuse text in a page |
| | Page-Linking Effort | Estimated elapsed time taken in hours to author links in a page |
| | Page-Structuring Effort | Estimated elapsed time taken in hours to structure a page |
| | Total Page Effort | Text effort + page-linking effort + page-structuring effort |
| Media-authoring task | Media Effort | Estimated elapsed time taken in hours to author or reuse a media file |
| | Media-Digitizing Effort | Estimated elapsed time taken in hours to digitize the media |
| | Total Media Effort | Media effort + media-digitizing effort |
| Program-authoring effort | Program Effort | Estimated elapsed time taken in hours to author or reuse a program |

Table 17.7 Web Metrics

| Entity Type | Response Variable | Predictor Variable |
|--|--------------------|---|
| Application authoring and design tasks | Total Effort | Page Count Media Count Program Count Total Page Allocation Total Media Allocation Total Code Length Reused Media Count Reused Program Count Total Reused Media Allocation Total Reused Code Length Connectivity Connectivity Density Total Page Complexity Cyclomatic Complexity |
| Page authoring | Total Page Effort | Page Allocation Page Linking Complexity Page Complexity Graphic Complexity Audio Complexity Video Complexity Animation Complexity Scanned Image Complexity |
| Media authoring | Total Media Effort | Media Duration Media Allocation |
| Program authoring | Program Effort | Code Length Code Comment Length Reused Code Length Reused Comment Length |

17.4.1 Web Metrics Based on Response and Predictor Variable

Mendes, Mosley and Counsell identified some response variables and the corresponding predictor variables for the different entity types in a Web-based system. Then they classified these predictor variables as different metrics. Some of these metrics for different types of entities are shown in Table 17.7.

17.4.2 Web Metrics Based on Web Characteristics

The size and effort requirement can also be estimated by evaluating different characteristics of Web applications. Some important characteristics are:

- **Web Graph Properties:** www can be represented as a graph structure where Web pages comprise nodes and hyperlinks denoted by directed edges. Graph-based metrics quantify structural properties of the Web on both macroscopic and microscopic scales.
- **Web Page Significance:** Significance metrics formalize the notions of ‘quality’ and ‘relevance’ of Web pages with respect to information needs of users. Significance metrics are employed to rate

candidate pages in response to a search query and have an impact on the quality of search and retrieval on the Web.

- Usage Characterization: Patterns and regularities in the way users browse Web resources can provide invaluable clues for improving the content, organization and presentation of websites. Usage characterization metrics measure user behavior for this purpose.
- Web Page Similarity: Similarity metrics quantify the extent of relatedness between Web pages. There has been considerable investigation into what ought to be regarded as indicators of a relationship between pages.
- Web Page Search and Retrieval: These are metrics for evaluating and comparing the performance of Web search and retrieval services.
- Information Theoretic: Information theoretic metrics capture properties related to information needs, production and consumption. The relationships between various regularities observed in information generation on the Web are considered here.

17.5 MOBILE WEB ENGINEERING

Web Engineering is mostly used for computers (mostly desktops and laptops). However recently, the use of handheld mobile devices has greatly increased in Web applications. E-news, e-shopping, e-banking and most other Web applications have already made a transition to mobile applications. Therefore, the standard processes for mobile Web applications have also become necessary. However, Web Engineering itself has not yet matured and the engineering process for mobile Web applications is still evolving. Particularly the issues of greater device dependency have made stiffer engineering practices.

The emphasis is on device-independent systems. Device independence refers to a single Web interface that can work satisfactorily on all devices. The mobile devices need a User Agent (UA) to access Web applications. UAs are designed to meet the minimum requirement of common Web standards. This World Wide Web Consortium (W3C) has recommended standards for Web applications and devices. Defining exactly what is the minimum common UA implementation required for a Web application is relative and is changing over time. UAs from Mozilla and Opera meet all the W3C standards.

There can be some context parameters for development of Web applications. The context requirements are not just the requirements of the stake holders, but also many other non-functional requirements. Of these, security and performance are the most important. The mobile applications need many contextual parameters as shown in Figure 17.4.

Time and location are the two major contextual parameters. The user's role and tasks are other context requirements. Due to the large varieties of mobile devices available in the market, the device also is part of the context parameter.

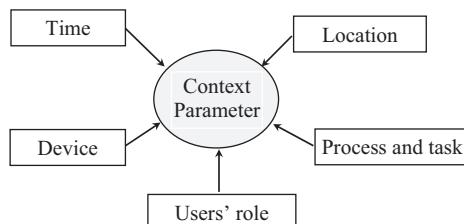


Figure 17.4 Context parameter classes

17.6 WEB ENGINEERING SECURITY

Security is an important issue in software engineering. However, it is more so in the case of Web Engineering. Because of the ubiquitous nature of these systems, it has become mandatory to develop a proper process inclusive of security. The essential security criteria of Web Engineering methodologies are as follows:

- Active organizational support for security aspects in the Web development process
- Proper controls in the development environment
- Security visibility throughout all areas of the development process
- Delivery of a cohesive system, integrating business requirements, software and security
- Prompt, rigorous testing and evaluation
- Trust and accountability

Threats to the security of Web-based systems can be mainly classified into confidentiality, integrity and availability. Confidentiality allows only authorized parties to read the protected information. For example, if the postman reads somebody's mail, this is a breach of that person's privacy. Integrity ensures that data are not tampered with as they move from the sender to the receiver. If someone added an extra bill to the envelope, which contained the customer's credit card bill, he has violated the integrity of the mail. Availability ensures that the authorized person has access to resources as and when needed. These three issues are generalized as given below:

- Access Control: Access to the system resources should only be limited to authorized persons, entities or processes.
- Confidentiality: Information is not accessed by or disclosed to unauthorized persons, entities or processes.
- Privacy: Individual rights to nondisclosure.
- Identification and Authentication: Verification and identification that the originator of a transaction is the same person, entity or process.
- Integrity: Information is not quietly altered or destroyed by an unauthorized person, entity or process.
- Availability: System resources are safeguarded from tampering and are available to the authorized persons as and when required in the required format.
- Non-repudiation: Undeniable proof of participation that the sender and/or receiver have made an online transaction.
- Accountability: Security-relevant activities on an e-commerce system can be traced back to individuals who may be held responsible for their actions.

Privacy and confidentiality are maintained by exercising control on access. Authentication at different stages ensures integrity. All these are technical solutions. However, sometimes the vulnerability comes through social factors like getting confidential information from users by tricking them. Hence, many research groups are also working on this issue.

Adoption of some policies on security can minimize the security risk to a great extent. Security policies contain guidelines on physical security, network security, access authorization, virus protection and disaster recovery. Specific elements of a security policy address the following points:

- Authentication: Verifies the person trying to access the e-commerce site. It enforces that the legal customer is the only one allowed to log on to his Internet banking account.

- Authorization: Allows only the authorized persons to manipulate their resources in specific ways. This prevents the customer from increasing the balance of his account or deleting a bill.
- Encryption: Deals with information hiding. It ensures one cannot spy on others during Internet banking transactions.
- Auditing: Keeps a record of operations. Merchants use auditing to prove that the customer bought specific merchandise.

This can be stated as a comprehensive and integrated security plan. The physical security of Web servers consists of tangible protection devices such as locks, alarms, guards, fireproof doors, security fences, secure buildings etc. As messages are transmitted over public communication channels, these are under great threat. So much attention is needed toward the network security. Proper access control mechanism should be provided to the authorized persons (including the customers) to enable proper security features. Protection against computer viruses also needs to be provided at the Web servers as well as to the client computers. In spite of all these sometimes it is difficult to guarantee a totally secure system. Hence, provisions for backup and recovery should be there, so that the Web-based system can be restored immediately in case of any eventuality or disaster. Therefore, all these security measures working together can prevent any unauthorized disclosure, destruction or modification of assets of the Web-based system.

SUMMARY

The Web is a collection of interlinked documents that is stored in electronic form on the Internet for the use of the general public or some authorized users.

Hypertext Document and GUIs for hypertext are two important innovations linked to Web-based software. HTML was developed by Tim Berners-Lee in 1991. The GUI for hypertext was first developed by a group of students led by Marc Andreessen at the University of Illinois, in 1993. It was the first Web browser called the Mosaic.

The new standard for Web technologies is called Web 2.0. It allows highly interactive websites, such as flicker, facebook, youtube etc. The concepts of AJAX are derived from Web 2.0.

During the last decade the Web has undergone great transformation and there is a paradigm shift in how websites are designed. This has led to emergence of new sub-discipline in software engineering called 'Web Engineering'. Web Engineering is a systematic approach that uses scientific, engineering and management principles to develop, deploy and maintain high-quality Web-based systems and applications.

OOWS, OOHDM, WebML and Open Space Framework are some methodologies of the Web Engineering process. The Web Engineering process has three major workflows, namely, (1) Analysis workflow, (2) Conceptual design workflow and (3) Detailed design workflow. The hypertext design is a major activity of the Web Engineering process. It consists of navigational design, interface design and layout design. The website is usually designed iteratively through prototyping. The prototype is used for evaluating the screen layout and navigation among different pages.

Web metrics have been developed to suit the specific needs of Web application projects relating to various aspects such as Length size metrics, Reusability metrics, Complexity size metrics, Effort metrics etc. Web metrics can also be based on response variables and the corresponding predictor variables. The size and effort requirement can also be estimated by evaluating different characteristics of Web applications. Web graph properties, Webpage significance, Usage characterization, Webpage similarity, Web-page search and retrieval, and Information theoretic are some such characteristics used for Web metrics.

Recently, the use of mobile devices has greatly increased in Web applications. Mobile devices are required to have an interface called UA to access Web applications. Hence, standards have been developed for UAs. W3C is the widely acceptable standard.

Confidentiality, Integrity and Availability are three major issues of Web security. Confidentiality allows only authorized parties to read the protected information. Integrity ensures that data are not tampered with as they move from the sender to the receiver. Availability ensures that the authorized person has access to resources as and when needed.

EXERCISES

1. Write the salient characteristics of Web-based applications.
2. Explain how the present day's Web-based systems are different from earlier systems.
3. Explain the necessity of having a separate sub-discipline in software engineering for development of Web applications.
4. Write the difference between OOHDM and WebML.
5. Explain the WebML process model using a suitable example.
6. Explain a website design process. Mention the various issues with respect to the design process.
7. Consider that the placement section of an educational institution wants to develop and maintain a separate website of its own. Write down the issues that may be raised with respect to the design process.
8. What are the different categories of Web metrics? Write down one metric from each category.
9. Describe the various issues that can be raised for development of Web applications for mobile devices.
10. Mention various criteria for security of Web-based systems.
11. Explain the concept of confidentiality, integrity and availability with respect to security of Web-based systems.
12. List various Web security issues and the probable solution for them.

appendix A

OBJECTIVE-TYPE QUESTIONS

MODULE – I (Introduction and Traditional Software Development)

1. Identify the advantages of using high-level languages over assembly languages.
 - A) High-level language programs are smaller in size.
 - B) High-level language reduces developmental efforts and time.
 - C) High-level language programs are faster.
 - D) All of the above.
2. Which of the following factors has contributed most to software crisis?
 - A) Large problem size
 - B) Slow progress in development of hardware
 - C) Lack of good programmers
 - D) Use of FORTRAN programming language
3. Which of the following is an essential program control structure?
 - A) Sequence
 - B) Selection
 - C) Iteration
 - D) All of the above
4. Which of these is not a part of a Software Requirement Specification (SRS) document?
 - A) Functional requirements of the system
 - B) Non-functional requirements of the system
 - C) Algorithms for software development
 - D) Goals of implementation
5. An SRS document normally contains
 - A) Functional requirements of the system ✓
 - B) Module structure
 - C) Configuration management plan
 - D) All of the above
6. Maintainability, portability, and usability issues are addressed by
 - A) Functional requirements

- B) Non-functional requirements
 - C) High level language
 - D) Low level language
7. During software life cycle, which activity generally consumes the maximum effort?
- A) Design
 - B) Maintenance
 - C) Testing
 - D) Coding
8. During which phase in the classical waterfall model is the SRS document produced?
- A) Design phase
 - B) Maintenance phase
 - C) Analysis phase
 - D) Coding phase
9. Which of the following phases in Waterfall life cycle immediately follows design phase?
- A) Coding phase
 - B) Maintenance phase
 - C) Testing phase
 - D) Analysis phase
10. Which of the following models is considered as the traditional model of software development?
- A) Waterfall Model
 - B) Prototyping Model
 - C) Spiral Model
 - D) None of the above
11. What is the Evolutionary model very useful for?
- A) Well-acquainted project that needs to be controlled
 - B) Very large problems
 - C) Projects whose user requirements are not well understood
 - D) Challenging software prone to several kinds of risks
12. What is the prototype model suitable for?
- A) Well-acquainted project that needs to be controlled
 - B) Very large problems
 - C) Projects whose user requirements are not well understood
 - D) Challenging software prone to several kinds of risks
13. What is the spiral model suitable for?
- A) Well-acquainted project that needs to be controlled
 - B) Very large problems
 - C) Projects whose user requirements are not well understood
 - D) Challenging software prone to several kinds of risks
14. The Function-Oriented (FO) approach
- A) Concentrates on functions
 - B) Is based on structured programming
 - C) Permits sharing of data by functions
 - D) All of the above
15. The Object-Oriented approach
- A) Concentrates on functions
 - B) Is based on structured programming

- C) Focuses on entities of system
 - D) Permits sharing of data by objects
16. Which of the following models is used in Structured Analysis and Design methodology of FO approach?
- A) Environmental Model
 - B) Behavioural Model
 - C) Implementation Model
 - D) All of the above
17. The purpose of Structured Analysis is to
- A) Determine the structure of system as perceived by users
 - B) Determine the structure of the solution for implementation/coding
 - C) Determine the structure chart
 - D) All of the above
18. Structured Analysis technique is based on
- A) Structured programming
 - B) Top-down decomposition approach
 - C) Principle of divide and conquer principle
 - D) All of the above
19. Data Flow Diagram (DFD) is also known as a
- A) Structure chart
 - B) Bubble chart
 - C) Program flowchart
 - D) Structured flowchart
20. The context diagram of a DFD is also known as
- A) Level 0 DFD
 - B) Level 1 DFD
 - C) Bottom level DFD
 - D) None of the above
21. Time-dependent behaviour of the system is described by:
- A) State Transition Diagram
 - B) DFD
 - C) Entity Relationship Diagram
 - D) Decision Table
22. A chart that defines a logical procedure by means of a set of conditions and related actions is called
- A) State Transition Diagram
 - B) DFD
 - C) Entity Relationship Diagram
 - D) Decision Table
23. Data dictionary that remains active during program execution and performs tasks such as transaction validation and editing is called
- A) Passive data dictionary
 - B) Active data dictionary
 - C) In-line data dictionary
 - D) Super-active data dictionary
24. Which of the following is a characteristic of modular design?
- A) It is difficult to understand
 - B) If there is one module all other modules need to be changed

- C) It makes the system difficult to maintain
 - D) It makes the system easy to modify
25. A module is said to have logical cohesion, if
- A) It is strongly linked to all other modules
 - B) All functions of the module are executed together at the same time
 - C) All elements of the module perform similar tasks
 - D) All of the above
26. If coupling among modules is high
- A) It is difficult to understand and maintain
 - B) It is difficult to integrate the modules
 - C) It is difficult to find and correct errors
 - D) All of the above
27. Which among the following is the worst type of coupling?
- A) Data Coupling
 - B) Stamp Coupling
 - C) Control Coupling
 - D) Content Coupling
28. The primary characteristic of a good design is
- A) Low cohesion and high coupling
 - B) Low cohesion and low coupling
 - C) High cohesion and low coupling
 - D) High cohesion and high coupling
29. In a structure chart, a module represented by a rectangle with double edges is called
- A) Main module
 - B) Library module
 - C) Primary module
 - D) Secondary module
30. Which of the statements about flowchart is true?
- A) It is easy to identify different modules of software from its flowchart
 - B) Data interchange between modules is clearly presented in a flowchart
 - C) Sequential ordering of tasks is clearly presented in a flowchart
 - D) All of the above
31. Input portion in DFD that transforms input data from physical to logical form is called
- A) Afferent branch
 - B) Efferent branch
 - C) Central transform
 - D) None of the above
32. During structured design, if all the input data are incident on different bubbles in the DFD, then we have to use
- A) Transform analysis
 - B) Transaction analysis
 - C) Both transform and transaction analyses
 - D) Neither transform nor transaction analysis
33. During structured design, if all the input data are incident on the same bubble in the DFD, we have to use
- A) Transform analysis
 - B) Transaction analysis

- C) Both transform and transaction analyses
 - D) Neither transform nor transaction analysis
34. Which of the following process for withdrawal of money from bank belongs to central transform?
- A) Get input data of customer and money to withdraw
 - B) Validate input data
 - C) Update customer account
 - D) Print transaction
35. Which of the following activities are done in detailed design?
- A) Development of pseudo-code for different modules in the form of MSPECs
 - B) Transaction and transform analyses of DFD
 - C) Design of module structure
 - D) All of the above

MODULE – II

(Object Oriented Software Development)

1. An object acquires the properties of objects of other classes by
 - A) Abstraction
 - B) Inheritance
 - C) Encapsulation
 - D) None of the above
2. The process of determining the appropriate function that should be invoked during the execution of program is termed as
 - A) Static binding
 - B) Dynamic binding
 - C) Method overloading
 - D) None of the above
3. Has-A relationship is termed as
 - A) Inheritance
 - B) Association
 - C) Aggregation
 - D) None of these
4. Object Modeling Technique (OMT) was developed by:
 - A) Rumbaugh
 - B) booch
 - C) Jacobson
 - D) Blob
5. Which of the following symbols was used by Booch to represent a Class?
 - A) Square
 - B) Rectangle
 - C) Cloud
 - D) One side open rectangle
6. Use-cases that do not have any actor are referred as
 - A) Abstract Use-case
 - B) Non-Use-case

- C) Zero Use-case
 - D) None of these
7. The structural view is depicted through
- A) Object Diagram
 - B) Class Diagram
 - C) Both of these
 - D) None of these
8. The property of a relationship that specifies how many instances of a class relates to a single instance of an associated class is called
- A) Cardinality of relationship
 - B) Degree of relationship
 - C) Both of these
 - D) None of these
9. LoanAdvisor creates an object Loan for a new loan. It is an example of
- A) Is-A relation
 - B) Has-A relation
 - C) Uses-A relation
 - D) None of these
10. In the context of Use-case diagram, the stick person icon is used to represent
- A) Human users
 - B) External systems
 - C) Internal systems
 - D) None of the above
11. The objects with which the actors interact are known as
- A) Controller objects
 - B) Boundary/interface objects
 - C) Entity objects
 - D) All of the above
12. The verbs in textual description described in the problem are considered to be
- A) Class
 - B) Attributes of class
 - C) Methods of class
 - D) None of these
13. The workflow behaviour that a system represents can be represented by
- A) Sequence diagram
 - B) Collaboration diagram
 - C) Class diagram
 - D) Activity diagram
14. The final state in an activity diagram can be represented by
- A) Diamond shape
 - B) Black circle
 - C) Encircled black circle
 - D) None of these

15. Synchronization bars can be put in
- Activity diagram
 - State diagram
 - Both of these
 - None of these
16. The vertical line that represents the role over time through the entire interaction in a sequence diagram is known as
- Synchronization bar
 - Lifeline
 - Both of these
 - None of these
17. Scenarios are shown graphically by using
- Interaction diagram
 - Workflow graphs
 - Both of these
 - None of these
18. The most important course of events (i.e. what happens most of the time) is represented in a Use-case document through
- Basic Flow
 - Alternate Flow
 - Use case description
 - None of these
19. The data of a system that are either used or stored are represented by
- Interface Class
 - Entity Class
 - Control Class
 - All of the above
20. The coupling between super-classes and sub-classes is called
- Interaction Coupling
 - Identity Coupling
 - Inheritance Coupling
 - None of the above
21. The number of message types an object must send to another object and the number of parameters passed with those messages is denoted by
- Interaction Coupling
 - Identity Coupling
 - Inheritance Coupling
 - None of the above
22. The run-time architecture of a system is depicted through
- Component Diagram
 - Deployment Diagram
 - Both of these
 - None of these
23. An anti-pattern having symptoms of single class with many attributes and methods is called
- Spaghetti code
 - Stovepipe system

- C) Vendor lock-in
 - D) Blob
24. A reusable design expressed as a set of abstract classes and the way their instances collaborate is called
- A) Pattern
 - B) Framework
 - C) Both of these
 - D) None of these
25. Which of the following layer usually discusses about the user interfaces?
- A) Presentation Layer
 - B) Business Logic Layer
 - C) Data Access Layer
 - D) None of these
26. The implementation view is depicted through
- A) Component Diagram
 - B) Deployment Diagram
 - C) Both of these
 - D) None of these
27. The business and database code can be represented using
- A) View
 - B) Controller
 - C) Model
 - D) None of these
28. A three-part rule that expresses a relation between a certain context, a problem, and a solution is called:
- A) Pattern
 - B) Framework
 - C) Both of these
 - D) None of these

MODULE – III

(User Interface, Coding, Testing, and Metrics)

1. Which type of user interface is suitable for supporting a large number of commands?
 - A) Mode-based interface
 - B) Modeless interface
 - C) Modeless GUI
 - D) None of the above
2. The facility to compose commands by adding optional parameters to the primitive command is a feature usually provided in
 - A) Menu-based interface
 - B) Command language-based interface
 - C) Direct manipulation interface
 - D) None of the above
3. The term “iconic interface” is applicable to
 - A) Menu-based interface
 - B) Command language-based interface

- C) Direct manipulation interface
 - D) None of the above
4. A development style based on widgets is called
- A) Command language-based GUI development style
 - B) Component-based GUI development style
 - C) Menu-based GUI development style
 - D) Direct manipulation-based GUI development style
5. Which of the following window objects are used for entering data into the system?
- A) Option button
 - B) Text box
 - C) List box
 - D) All of the above
6. The window object that allows users to make multiple choices among various options is
- A) Option button
 - B) Check box
 - C) List box
 - D) All of the above
7. Tables and Grids allow users to
- A) Enter large amount of data at a time
 - B) View large amount of data at a time
 - C) Enter or view large amount of data at a time
 - D) Used only for very small amount of data at a time
8. Which of the following statements about Window Management System (WMS) is true?
- A) WMS monitors of screen area resources and allocates it to different windows
 - B) WMS can be considered as a user interface management system (UIMS)
 - C) WMS provides several utilities for the development of user interface
 - D) All of the above
9. The Coding Standard that has general applicability for program codes across all projects is a type of
- A) General Coding Standard
 - B) Language-specific Coding Standards
 - C) Project-specific Coding Standard
 - D) Coding Guidelines
10. The Coding Standard prescribes some guidelines relating to
- A) Formatting features
 - B) Naming convention for source file, variable, classes, functions, and methods
 - C) Norms for desirable size of classes, subroutines, functions, and methods
 - D) All of the above
11. A set of desirable but non-mandatory practices that make the program codes easy to read and maintain is called
- A) General Coding Standard
 - B) Language-specific Coding Standards
 - C) Project-specific Coding Standard
 - D) Coding Guidelines
12. Written description of a software product is a type of
- A) Requirements documentation
 - B) Design documentation

- C) User documentation
 - D) Marketing documentation
13. Documentation to create interest for the software among potential users is a type of
- A) Requirements documentation
 - B) Design documentation
 - C) User documentation
 - D) Marketing documentation
14. As per 'Naming Convention' the names of the classes, subroutines, functions, and methods
- A) Should be a verb
 - B) Should be a noun
 - C) Should start with a verb followed by a noun
 - D) Should start with a noun followed by a verb
15. Which of the following coding guidelines is desirable?
- A) An identifier should not be used for multiple purposes
 - B) The use 'go to' statements should be avoided
 - C) Coding style that is too clever or cryptic should be avoided
 - D) All of the above
16. Usage mode documentation in which information is arranged in an alphabetic order is called
- A) Reference mode documentation
 - B) Information-oriented instructional mode documentation
 - C) Task-oriented instructional mode documentation
 - D) None of the above
17. Lower (Back-End) CASE Tools are most often used
- A) For preliminary investigation
 - B) For software analysis and design
 - C) As an aid to generate program codes
 - D) All of the above
18. Which of the following statements is FALSE?
- A) Workbenches integrate several CASE tools into one application
 - B) CASE tools is a collection of CASE environment and workbenches
 - C) CASE environment is a collection of CASE tools and workbenches
 - D) None of the above
19. Which type of testing uses the 'equivalence partitioning' method?
- A) Black box testing
 - B) White box testing
 - C) Can be part of both of these
 - D) None of the above
20. If-then-else can be best tested by using
- A) Statement coverage testing
 - B) Branch coverage testing
 - C) Both of the above
 - D) None of the above
21. Which process is used to check the conformance of software to its specification?
- A) Verification
 - B) Validation

- C) Both of these
 - D) None of these
22. Which testing considers interactions between groups of cooperating classes?
- A) Cluster-level
 - B) System-level
 - C) Thread
 - D) None of these
23. In which integration testing are all the modules of the system put together and tested?
- A) Incremental integration
 - B) Big-bang integration
 - C) Both of these
 - D) None of these
24. A test driver is used in
- A) Bottom-up Testing
 - B) Top-down Testing
 - C) Both of these
 - D) None of these
25. A stub is used in
- A) Bottom-up Testing
 - B) Top-down Testing
 - C) Both of these
 - D) None of these
26. The system testing that is carried out by the test team within the organization is called
- A) Alpha Testing
 - B) Beta Testing
 - C) Acceptance Testing
 - D) None of these
27. The Cyclomatic Complexity $V(G)$ can be computed as
- A) $V(G) = E - N + 2$ where N is number of nodes and E is number of edges.
 - B) $V(G) = \text{Total number of bounded areas} + 1$
 - C) $V(G) = \text{Total number of decision points} + 1$
 - D) All of the above
28. According to IFPUG Function Point Metrics, the functions of an application are categorized into
- A) 3 types
 - B) 5 types
 - C) 8 types
 - D) 14 types
29. How many 'General System Characteristics (GSC)' are there in Function point metrics for adjusting Unadjusted Function Point (UFP)?
- A) 3 GSCs
 - B) 5 GSCs
 - C) 8 GSCs
 - D) 14 GSCs
30. The probability of repairing the software during a particular time interval is called
- A) Repairability
 - B) Defect density

- C) Mean time to repair
 - D) Mean time to failure
31. The number of modules subordinate to a module is known as
- A) Module Complexity
 - B) Module Coupling
 - C) Fan-in
 - D) Fan-out
32. Which coupling is a function of fan-in and fan-out?
- A) Data and control flow coupling
 - B) Global coupling
 - C) Environmental coupling
 - D) None of these
33. For determination of Function Points, shared databases and shared mathematical routines are categorized as
- A) External inputs
 - B) External outputs
 - C) Internal files
 - D) External interfaces
34. The ratio of LOC/FP for a C language is
- A) 320
 - B) 128
 - C) 106
 - D) None of these
35. The total number of operators and operands is defined as
- A) Program length
 - B) Program vocabulary
 - C) Volume
 - D) None of these
36. Which of the following metrics is an external quality metrics?
- A) Maintainability
 - B) Testability
 - C) Reliability
 - D) Flexibility
37. Which of the following metrics is an internal quality metrics?
- A) Integrity
 - B) Accuracy
 - C) Usability
 - D) Reusability
38. Which of the following metrics is an external process quality metrics?
- A) Cost effectiveness
 - B) Time and effort
 - C) Number of design specifications
 - D) Number of coding faults
39. Which of the following metrics is a component level design metrics?
- A) Structural complexity
 - B) Data complexity

- C) System complexity
 - D) Cohesion and coupling
40. Which of the following metrics is based on the sum of inherited methods in all classes of systems under consideration?
- A) Method inheritance factor
 - B) Method hiding factor
 - C) Coupling factor
 - D) None of these
41. Which of the following is an Object-Oriented metrics?
- A) Polymorphism Factor
 - B) Method Inheritance Factor
 - C) Method Hiding Factor
 - D) All of the above

MODULE – IV

(Software Project Estimation, Management, and Quality)

1. Which of the following is carried out first for estimation of a software project?
 - A) Estimation of cost of the project
 - B) Estimation of duration of the project
 - C) Estimation of size of the project
 - D) Estimation of development effort
2. In estimation by Parametric Modelling
 - A) Size of software is determined based on estimation of developmental effort
 - B) Effort required to develop the software is determined based on estimation of size
 - C) Size of software is determined based on estimation of cost
 - D) Effort required to develop the software is determined based on estimation of cost
3. The Delphi method is used in
 - A) Estimation by Expert Judgement
 - B) Estimation by Analogy
 - C) Estimation by Available Resources
 - D) Estimation by Parametric Modelling
4. Operating systems and real-time system programs can be considered as
 - A) Organic software
 - B) Semidetached software
 - C) Embedded software
 - D) None of the above
5. A statistical and simulation package can be considered as
 - A) Organic software
 - B) Semidetached software
 - C) Embedded software
 - D) None of the above
6. Data processing programs are considered as
 - A) Organic software
 - B) Semidetached software

- C) Embedded software
 - D) None of the above
7. In which COCOMO model is effort adjusted from rating of cost drivers?
- A) Basic COCOMO
 - B) Intermediate COCOMO
 - C) Complete COCOMO
 - D) COCOMO-II
8. In which COCOMO model are the subsystems estimated separately and then added up to give total estimates of the project?
- A) Basic COCOMO
 - B) Intermediate COCOMO
 - C) Complete COCOMO
 - D) COCOMO-II
9. Which COCOMO model uses Scaling Factors (SF) for estimation of software project?
- A) Basic COCOMO
 - B) Intermediate COCOMO
 - C) Complete COCOMO
 - D) COCOMO-II
10. The COCOMO-II model comprises
- A) Early design COCOMO model and Post-architecture COCOMO model
 - B) Basic COCOMO model and Intermediate COCOMO model
 - C) Basic COCOMO, Intermediate COCOMO, and Complete COCOMO
 - D) All of the above
11. The work components of a project arranged in a tree-like hierarchical structure is called
- A) Project Phases
 - B) Work Breakdown Structure
 - C) Project Milestones
 - D) Project Network
12. Dummy activities are sometimes used to show relationship between activities in
- A) Activity on Arrow network
 - B) Activity on Node network
 - C) Both activity on Arrow and activity on Node network
 - D) Not shown in any network diagram
13. Which of the following statements about Critical Path Method (CPM) is not true?
- A) CPM helps to analyze activities of a project, for monitoring and resource allocation.
 - B) It is assumed that activity duration is certain.
 - C) Among two activities the one that has greater float is more critical than the other.
 - D) In CPM, the float of an activity is the difference between LST and EST.
14. If t_o is optimistic time, t_m is most likely time, and t_p is pessimistic time, then in PERT, which of the following expressions is used to determine expected time of an activity?
- A) $(t_o + t_m + t_p)/3$
 - B) $(t_o + 2t_m + t_p)/4$
 - C) $(t_o + 4t_m + t_p)/6$
 - D) $(3t_m - t_o - t_p)$

15. The difference between Budgeted Cost of Work Performed (BCWP) and Budgeted Cost of Work Scheduled (BCWS) is called
A) Schedule Variance
B) Cost Variance
C) Accounting Variance
D) Total Variance
16. The difference between BCWS and Actual Cost of Work Performed (ACWP) is called
A) Schedule Variance
B) Cost Variance
C) Accounting Variance
D) Total Variance
17. The difference between BCWP and ACWP is called
A) Schedule Variance
B) Cost Variance
C) Accounting Variance
D) Total Variance
18. The project schedule can be graphically depicted by
A) PERT chart
B) Project Network Diagram
C) Work Breakdown Structure chart
D) Gantt chart
19. 'Likely errors made due to complexity of software project and lack of adequate expertise/experience' is a type of:
A) Schedule-related Risk
B) Technical Risk
C) Operational Risk
D) Financial Risk
20. 'Likely events that can adversely affect administrative/execution activities of a project' is a type of
A) Schedule-related Risk
B) Technical Risk
C) Operational Risk
D) Financial Risk
21. 'Identification of major risk items from the list containing a large number of possible risk items' is called
A) Risk Assessment
B) Risk Control
C) Risk Acceptance
D) Risk Transfer
22. Which among the following is a Risk Management Strategy?
A) Risk Acceptance/Retention
B) Risk Avoidance
C) Risk Reduction/Mitigation
D) All of the above
23. Which of the following statements about Configuration control is not true?
A) Changes to work products should be done only by authorized persons.
B) There should be some specified procedure for making changes.

- C) The same work product should be changed separately by two or more persons.
 - D) None of the above.
24. A revision of software refers to
- A) Fixing of a minor bug
 - B) Minor enhancements to the functionality, usability etc.
 - C) Fixing of a minor bug or minor enhancements to the functionality, usability etc.
 - D) Significant change in functionality, technology, or the hardware of the software
25. A new version of software refers to
- A) Fixing of a minor bug
 - B) Minor enhancements to the functionality, usability etc.
 - C) Fixing of a minor bug or minor enhancements to the functionality, usability etc.
 - D) Significant change in functionality, technology, or the hardware of the software
26. According to Juran, which are the three processes necessary for quality?
- A) Quality Planning, Quality Control, and Quality Improvement
 - B) Testing, Inspection, and Quality Control
 - C) Pre-inspection, Post-inspection, and Quality Control
 - D) Inspection, Quality Control, and Statistical Quality Control
27. Which of the following statements is incorrect?
- A) Sporadic problems are caused due to random or un-assignable causes.
 - B) Sporadic problems are acted upon by the process of quality control.
 - C) Chronic problems occur due to assignable causes.
 - D) Chronic problems are acted upon by the process of quality control.
28. Which types of problems are acted upon by the process of quality control?
- A) Problems caused due to random causes
 - B) Problems caused due to assignable causes
 - C) Chronic problems
 - D) Both sporadic and chronic problems
29. Plan-Do-Check-Act (PDCA) Cycle is a systematic approach for doing
- A) Quality Control
 - B) Continuous Improvement
 - C) Quality Audit
 - D) Six Sigma
30. Cost of Quality comprises
- A) Appraisal costs and Failure costs
 - B) Appraisal costs and Prevention costs
 - C) Appraisal costs, Prevention costs, and Failure costs
 - D) Only Failure costs
31. The term 'factory within factory' refers to
- E) Reworking of defective products
 - F) Making multiple products in one premise
 - G) A factory having a number of workshops
 - H) A factory working in multiple shifts
32. In Malcolm Baldrige National Quality Award (MBNQA), maximum weightage is given to
- A) Human Resource Management
 - B) Management of Process Quality

- C) Quality and Operational Results
 - D) Customer Focus and Satisfaction
33. Which of these objectives does not come under the purview of Quality Audit?
- A) To check if processes are carried out as per established procedures
 - B) To do thorough testing and debugging of software
 - C) To check if proper control procedures are being followed
 - D) To check if the required documentation is being maintained
34. Six Sigma is a concept that tries to achieve a quality level where percentage of defects is
- A) Zero defect
 - B) Less than 0.006%
 - C) Less than 0.034%
 - D) Less than 0.334%
35. In Six Sigma methodology, DMAIC developed by Motorola stands for
- A) Define, Measure, Analyze, Improve, and Control
 - B) Define, Monitor, Analyze, Improve, and Control
 - C) Document, Measure, Analyze, Inspect, and Control
 - D) Document, Monitor, Analyze, Inspect, and Control
36. Which of these Six Sigma tools is used to identify, communicate, and prioritize customer requirements?
- A) Cause and Effect Matrix
 - B) Failure Modes and Effects Analysis (FMEA)
 - C) Quality Function Deployment (QFD)
 - D) Design of Experiments (DOE)
37. The generic ISO 9000 standard is based on
- A) Two basic principles; Modularity and Responsibility
 - B) Say what you do, do what you say, and prove it
 - C) DMAIC
 - D) Improving the capability of its processes
38. ISO 12207 standard for software lifecycle processes is based on
- A) Two basic principles; Modularity and Responsibility
 - B) Say what you do, do what you say, and prove it
 - C) DMAIC
 - D) Improving the capability of its processes
39. The Capability Maturity Model (CMM) framework specifies:
- A) Three levels of maturity
 - B) Four levels of maturity
 - C) Five levels of maturity
 - D) Six levels of maturity

ANSWERS

Module – I

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1B | 2A | 3D | 4C | 5A | 6B | 7B | 8C | 9A | 10A |
| 11B | 12C | 13D | 14D | 15C | 16D | 17A | 18D | 19B | 20A |
| 21A | 22D | 23C | 24D | 25C | 26D | 27D | 28C | 29B | 30C |
| 31A | 32B | 33A | 34C | 35A | | | | | |

Module – II

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1B | 2B | 3C | 4A | 5C | 6A | 7C | 8A | 9C | 10A |
| 11B | 12C | 13D | 14C | 15C | 16B | 17A | 18A | 19B | 20C |
| 21A | 22B | 23D | 24B | 25A | 26A | 27C | 28A | | |

Module – III

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1A | 2B | 3C | 4B | 5D | 6B | 7C | 8D | 9A | 10D |
| 11D | 12B | 13D | 14C | 15D | 16A | 17C | 18B | 19A | 20B |
| 21A | 22A | 23B | 24A | 25B | 26A | 27D | 28B | 29D | 30C |
| 31D | 32C | 33D | 34B | 35A | 36C | 37D | 38A | 39D | 40A |
| 41D | | | | | | | | | |

Module – IV

| | | | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1C | 2B | 3A | 4C | 5B | 6A | 7B | 8C | 9D | 10A |
| 11B | 12A | 13C | 14C | 15A | 16C | 17B | 18D | 19B | 20C |
| 21A | 22D | 23C | 24C | 25D | 26A | 27D | 28A | 29B | 30C |
| 31A | 32D | 33B | 34C | 35A | 36C | 37B | 38A | 39C | |

FREQUENTLY ASKED QUESTIONS WITH SHORT ANSWERS

MODULE – I **(Introduction and Traditional Software Development)**

1. What is Software Engineering?

Answer: Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software. It is the application of engineering methods to software development.

2. Write two main characteristics that differentiate software from other physical products.

Answer: The two main characteristics that differentiate software from other physical products are:

- i. The costs of software development are concentrated in engineering and not in production. Also, the cost of software is not dependent on volume of production.
- ii. Software does not wear out and it does not require any spare parts.

3. What are different generations of computers?

Answer: The development of computer technology has taken place in distinct stages. The first generation of computers was built on vacuum tubes. The second-generation machines were built by using transistors. The third generation made use of Integrated Circuits (ICs). The present day's fourth-generation computers use Very Large Scale Integration (VLSI) circuits.

4. How has the development of DBMS helped the software industry?

Answer: Development of DBMS software helped in use of computers for data storage. This permitted the development of more sophisticated software for business applications.

5. What is software crisis?

Answer: In the early days of software industry, programmers followed their own personal style for writing programs. The design was implicitly performed in one's mind, and documentation was often non-existent. Due to developments in the field of information technology, computer programs also became very large and complex. With no tools or methods for managing this increased complexity, a large proportion of software projects failed during 1970s. It was referred to as 'software crisis'.

6. What is 'principle of abstraction' in relation to software engineering?

Answer: The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. Once simpler, the problem is solved and then the omitted details can be taken into consideration in stages.

7. What is ‘principle of decomposition’ in relation to software engineering?

Answer: Decomposition is a technique to solve a complex problem by dividing it into several smaller problems. The problem is decomposed in such a way that each component can be solved independently and then the solution of the different components can be combined to get the full solution.

8. What are the four core aspects of software development?

Answer: Product, Process, People and Project are four important core aspects of software development. Software development involves both technical and managerial aspects.

9. What do you mean by software process?

Answer: The term software process refers to the methods of developing software. It generally involves a number of steps or operations.

10. What are different types of software processes?

Answer: There are four major types of software processes. These are: (1) Software Development Process, (2) Software Project Management Process, (3) Software Configuration Management Process and (4) Software Process Management.

11. What are the different phases of a classical waterfall model?

Answer: Different phases of a classical waterfall model are: (1) System Engineering, (2) Requirement Analysis, (3) Design, (4) Implementation or Coding, (5) Verification or testing, (6) Maintenance

12. What is a software prototype?

Answer: A prototype is a working representation of the system. By seeing and interacting with the prototype model, the users are better able to realize what the real system will be like. The prototype is refined till the users are reasonably satisfied. From this the software developers are able to develop software that fulfils the real needs of users.

13. Which is the model suitable for development of technically challenging software products?

Answer: Risk handling is inherently built into the spiral model. The spiral model is suitable for development of technically challenging software products because these are prone to several kinds of risks.

14. Name some process standards applicable to software industry.

Answer: The ISO/IEC 12207 standard, Capability Maturity Model (CMM), ISO 9000 standards, and SPICE (Software Process Improvement Capability Determination) are some popular process standards applicable to the software industry.

15. What is Requirement Engineering?

Answer: Requirement Engineering is a systematic approach for determining software requirement specifications.

16. Name six types of software requirement specifications as per IEEE standard.

Answer: The six types of software requirement specifications are: (1) Design requirement, (2) Functional requirement, (3) Implementation requirement, (4) Interface requirement, (5) Performance requirement and (6) Physical requirement.

17. What are the important activities of the requirements engineering process?

Answer: Inception, Elicitation, Elaboration, Negotiation, Specification, Validation and Management are the important activities of the requirements engineering process.

18. Who are the stakeholders of a typical software system?

Answer: A stakeholder can be defined as 'anyone who benefits in a direct or indirect way from the software system being developed.' The usual stakeholders are management, marketing people, consultants, system analysts, software engineers and support/maintenance engineers of the software firm, and also buyers and end-users of the software.

19. Name some standard techniques used for elicitation of users' requirements.

Answer: Requirement Elicitation through Interview, Questionnaire, Record Review, Direct Observation and Collaborative Requirement Gathering are some standard techniques used for elicitation of users' requirements.

20. What is SRS?

Answer: SRS stands for Software Requirement Specification. It describes the functions and performance requirements of software. It also lists the constraints that will affect software development. It serves as the basis for subsequent software engineering activities.

21. List the important characteristics of a structured program.

Answer: A structured program avoids unstructured control flows by restricting the use of GOTO statements. It mainly uses three types of program constructs, i.e. Selection, Sequence and Iteration.

A) A structured program consists of a well-partitioned set of modules. It uses single entry, single-exit program constructs such as if-then-else, do-while etc. Thus, the structured programming principle emphasizes designing control structures in an orderly manner.

22. Name the three essential models of 'Structured System Analysis and Design (SSAD)'.

Answer: Environmental Model, Behavioral Model and Implementation Model are three essential models of SSAD.

23. Differentiate between Static and Dynamic Modeling.

Answer: In OOAD, the objects and their relationships are designed by using some model and are expressed as some data-type in a programming language. This is the static or structural aspects of design and is called 'Static Modeling'.

B) The objects interact with each other and also perform some action. This dynamic aspect of object is called object behavior. The behavior of objects is defined through methods. This is called 'Dynamic Modeling'.

24. Name the four major components of DFD and how they are depicted?

Answer: The four major components of DFD are given below.

- i. External entity: It represents the external source of input data or recipient of output. It is depicted by a Square or Rectangle.
- ii. Process: It represents the function or procedure that causes some transformation to data. It is depicted by a Circle or Ellipse.
- iii. Data Flow: It represents either input to or output of a process. It is data structure in motion. It is depicted by an Arrow.
- iv. Data Store: It represents data file stored for future processing. It is data structure in motion. It is depicted by an Open-end box.

25. Differentiate between Physical DFD and Logical DFD.

Answer: A logical DFD specifies various logical processes performed on data. It does not specify information such as: who does the processing, where the processing is done or on which physical device data are stored. The above facts are specified by physical DFD.

26. Differentiate between decision table and decision tree.

Answer: The decision table is a chart that defines a logical procedure by means of a set of conditions and related actions. It consists of four sections, namely, Condition stub, Condition entry, Action stub and Action entry. A decision tree is a graphic representation of a decision process indicating decision alternatives.

27. What is data dictionary?

Answer: Data dictionary is an electronic glossary of items, which specifies the characteristics of all elements of a system, namely, data flow, data stores, processes and entities.

28. Write the application of Entity-Relationship (ER) diagram.

Answer: The details about the data stores and the relationships between them are shown by the ER diagram. The ER diagram is used for modeling of stored data.

29. What are the four components to software design?

Answer: Architectural or Structural design, Detail design, Data design and Interface design are the four components to software design.

30. Define the concept of modularization.

Answer: The process of decomposition of top-level modules of a system successively into further smaller sub-modules that are organized into a hierarchical structure until the lower level modules are of manageable size and complexity is called modularization.

31. Distinguish between coupling and cohesion.

Answer: The degree of interdependence or interaction between modules is called coupling, whereas the extent to which the instructions of a module contribute to performing a single unified task is called cohesion. A good design should have low coupling and high cohesion.

32. Write the application of the Structure chart.

Answer: The Structure chart is a graphical tool to depict the modular structure of software. It shows how the program has been partitioned into smaller modules, the hierarchy and the communication interfaces between modules.

33. How is Detail design different from Architectural design?

Answer: The Architectural design described by structure charts do not show the internal data and the procedures performed by the modules. The Detail design gives the complete description and the algorithm of each module through ‘module specification’.

34. What is MSpec?

Answer: Module specification is also called ‘MSpec’. It depicts the algorithm of modules through Flowchart and/or Pseudocode.

MODULE – II

(Object-Oriented (OO) Software Development)

1. What is an object?

Answer: An object is anything that exists in the real world, such as a person, a place or a thing. It can be any noun or noun phrase, either physical or conceptual. An object contains data and functions as its integral parts. The data integral to an object are called attributes of the object and the functions are called methods.

2. Differentiate between class and object.

Answer: A class is a group of similar objects. It is a blueprint that is used to create objects. A single object is simply an instance of a class.

3. What is Inheritance?

Answer: Inheritance is the process by which objects acquire the properties of objects of other class. When two classes have parent child relationship, the child class (subclass) inherits the properties of the parent class (super class). Inheritance is specified by 'Is-A Relationship'.

4. What is meant by Encapsulation in OOSD?

Answer: Encapsulation means containing or packaging data within an object. Here data and procedures are tied together to prevent them from external misuse. Encapsulation also means data hiding, which is the process of hiding all aspects of an object that do not contribute to its essential characteristics.

5. What is meant by Encapsulation in the context of object technology?

Answer: *Polymorphism* means the ability to take more than one form. It means the same operation behaving differently on different classes. Polymorphism is achieved by method overloading and method overriding.

6. What are different types of relationships?

Answer: There are three main types of relationships, as given below.

C) Is-A Relationship: This relationship specifies inheritance.

D) Has-A Relationship: This relationship is used for Aggregation. Here, a class contains another class as its member.

E) Uses-A Relationship: This relationship represents Association.

7. What is Object Identifier (OID)?

Answer: Whenever an object is created, it is uniquely identified in the system by a unique code called OID or sometimes as unique identifier (UID). OID is never changed or deleted even if the objects name or its location is changed or the object is deleted.

8. Write about the four phases in OMT?

Answer: OMT consists of four phases. These are:

- i. Analysis: In this phase an abstraction of what the desired system must do is made in terms of attributes and operations.
- ii. System design: The overall architecture and high-level decisions about the system are made.
- iii. Object design: It produces a design document, consisting of detailed objects static, dynamic and functional models.
- iv. Implementation: This activity produces reusable, extendible, robust code.

9. What is object model of OMT?

Answer: The object model describes the static structure of the objects in the system and their relationships. It represents the data aspects of the system.

10. What are different diagrams used in Booch methodology?

Answer: Booch's methodology uses six types of diagrams to depict the underlying design of a system. These are: (1) Class Diagram, (2) Object Diagram, (3) State Transition Diagram, (4) Module Diagram, (5) Process Diagram and (5) Interaction Diagram.

11. What are Use-cases?

Answer: Use-cases are scenarios for understanding the user requirements. A use-case is depicted through the use-case diagram. The use-case diagram has mainly two types of components, namely, 'actors' and 'use-cases'. Actors in the use-case diagram are represented by stick men and use-cases by ovals.

12. Write a short note on Objectory.

Answer: Jacobson along with others developed a complete set of tools required for OO software engineering. This was called object factory for software development and is often referred to as *Objectory* or '*Objectory*'. It is built around several models. These are: Use-case model, Requirement model, Domain object model, Analysis model, Design model, Implementation model and Test model.

13. Why do we go for the unified approach?

Answer: The unified approach combines the best practices, processes, methodologies and guidelines of different OO methodologies. It uses UML (Unified Modeling Language) notations and diagrams that are considered as universal standard in OO system development.

14. What is UML?

Answer: UML stands for Unified Modeling Language. It is a language for specifying, constructing, visualizing and documenting the software system and its components. It is accepted by Object Management Group (OMG) as standard for modeling in OOSD.

15. Write the primary goals of UML.

Answer: The primary goals of UML in the design are as follows:

- Provide users a ready-to-use, expressive visual modeling language.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of any particular programming language or development process.
- Encourage the use of OO tools and their demand in the market.
- Support higher-level development concepts.
- Integrate best practices and methodologies.

16. Enumerate different diagrams of UML.

Answer: Different diagrams used in UML are: (1) Use-case diagram, (2) Class diagram, (3) Object diagram, (4) Activity diagram, (5) State diagram, (6) Sequence diagram, (7) Collaboration diagram, (8) Component diagram and (9) Deployment diagram.

17. What are various approaches for identifying classes?

Answer: There are different alternative approaches for identifying classes. These are: (1) Noun phrase approach, (2) Classical approach, (3) Function point (FP) approach, (4) Domain analysis approach, (5) Structural approach, (6) CRC approach and (7) Use-case-driven approach.

18. What do you mean by relevant, fuzzy and irrelevant classes?

Answer: In the noun phrase approach the list of nouns is divided into three categories:

- **Relevant classes:** These classes have a purpose. Each class must have a purpose and should be clearly defined and necessary.
- **Irrelevant classes:** These classes have no purpose and are unnecessary. It is safe to scrap these irrelevant classes.
- **Fuzzy classes:** Fuzzy classes are those classes that we are not sure about. These classes cannot be determined as either relevant or irrelevant and thus considered as fuzzy.

19. What are the sources used for finding the candidate classes?

Answer: Some sources used for finding candidate class and object are: (1) People - who carry out some functions, (2) Places - set aside for people or things, (3) Devices - with which the application interacts, (4) Organization, (5) Concepts, principles or ideas, (6) Events - that happen, (7) Other external systems with which the application interacts and (8) Roles played by different roles, the users.

20. What is CRC?

Answer: CRC stands for class, responsibility and collaborator. It is a technique used for identifying the attributes and methods of classes.

21. Write the three steps of the CRC process?

Answer: The CRC process consists of three steps as given below.

- i. Identify classes' responsibilities (and identify classes)
- ii. Assign responsibilities
- iii. Identify collaborators

22. How can we identify the methods?

Answer: Methods usually correspond to queries about attributes and sometimes association of objects. Methods determine the value of the attributes. For example, if 'cost' is an attribute of a product, then `getCost()` can be a corresponding method.

23. List the main activities of the OO design process?

Answer: The main activities of the OO design process are listed as under.

- Designing classes (their attributes, methods, associations, structures etc.)
- Designing the access layer
- Designing the user interface
- Testing user satisfaction and usability
- Iteratively refining the design

24. Define axiom? What are the two design axioms applicable to the OO design?

Answer: An axiom is a fundamental truth that always is observed to be valid. The axioms may not be proven or derived but they cannot be invalidated by counterexamples or exceptions. There are two design axioms applied to the OO design.

- Axiom 1: The independence axiom. Maintain the independence of components.
- Axiom 2: The information axiom. Minimize the information content of the design.

F) Axiom 1 deals with relationships between system components and Axiom 2 deals with the complexity of design.

25. Define corollary? Give an example of a corollary derived from design axioms.

Answer: A corollary is a proposition that follows from an axiom or another proposition that has been proven. 'Uncoupled design with less information content' is an example of a corollary derived from design axioms.

26. What are different types of coupling in the OO design?

Answer: The OO design has three types of coupling. These are:

- i. **Interaction Coupling:** It is a measure of the number of message types an object must send to another object and the number of parameters passed with those messages.

- ii. **Identity Coupling:** It refers to the level of connectivity in a design. For example, in case of a reference from one object to another, the object must know the identity of the other object.
 - iii. **Inheritance Coupling:** It refers to the coupling between super-classes and sub-classes. A sub-class is coupled to its super-class in terms of attributes and methods.
27. What do you mean by cohesion in the OO design?

Answer: Cohesion can be defined as the interactions within a single object or software component. It reflects the ‘single-purpose-ness’ of an object.

28. Define anti-patterns.

Answer: An anti-pattern represents a worst practice while a pattern represents a best practice. Anti-patterns come in two types:

- i. Those that describe a bad solution to a problem resulting in a bad situation
- ii. Those that describe how to get out of a bad situation

29. Write the differences between design patterns and frameworks.

Answer: Design patterns are more abstract, are smaller architectural elements and are less specialized than frameworks. A framework is executable software whereas design patterns represent knowledge and experience about the software.

MODULE – III

(User Interface, Coding, Testing and Metrics)

1. What are the major design issues in command language-based interface?

Answer: To reduce the number of primitive commands and to minimize the total typing required while issuing commands, are two major design issues in command language-based interface. To meet these two objectives, facility to compose commands by adding optional parameters to the primitive command is usually provided in command language-based interface.

2. Distinguish between Modeless interface and Mode-based interface.

Answer: A mode is a state of the system, which permits only some of the commands to be performed. Hence, in a mode-based interface, different set of commands can be invoked depending on the mode of the system. On the other hand, in a modeless interface, there is only one set of commands that can be invoked at all times during the running of the software. Thus, a modeless interface has only a single mode. The mode-based interface is useful for software that has large number of commands.

3. What are the three types of menus in WIMP user interface?

Answer: WIMP stands for ‘Windows Icon Menu and Pointer system’. It is the most popular form of GUI. It has mainly three types of menus: (1) Scrolling menu, (2) Walking menu and (3) Hierarchical menu.

4. List the functions of Window Management System (WMS).

Answer: WMS can be considered as the management system of user interface. WMS keeps track of the screen area resources and allocates it to the different windows. It also provides utilities for the development of the user interface.

5. Name some window objects (widgets) that are used for entering data into the system.

Answer: The Option Button, Text Box and List Box are window objects that are widely used for entering data into the system.

6. What is coding standard?

Answer: For consistency and ease of understanding, it is desirable that all programmers should follow a well-defined and common style of coding called 'Coding Standard'. A coding standard prescribes rules and guidelines about declaration and naming of variables, formatting features of code, error return conventions etc.

7. What are the three types of Coding Standard?

Answer: General coding standard, Language-specific coding standard and Project-specific coding standard are the three types of Coding Standard.

8. What is the difference between Coding Standard and coding guidelines?

Answer: Coding guidelines provide the programmer with a set of best practices that can be used to make program codes easier to read and maintain. The programmers are encouraged to incorporate these into their coding style. However, unlike the Coding Standards, the use of these guidelines is not mandatory.

9. What are different types of software documentations?

Answer: The software documentations are usually in the form of paper books or computer readable files. There are different types of software documentations. These are: (1) Requirements documentation, (2) Design documentation, (3) Program documentation, (4) User documentation and (5) Marketing documentation.

10. What are CASE tools?

Answer: CASE tool stands for 'Computer-Assisted Software Engineering' or sometimes 'Computer-Aided Software Engineering'. CASE tools are software to automate methods for designing, documenting and producing structured computer code in the desired programming language.

11. Differentiate between CASE tools, Workbenches and CASE Environments

Answer: A set of CASE tools are generally designed to be compatible with each other, so that information generated from one tool can be passed to other tools. Workbenches integrate several CASE tools into one application to support specific software-process activities. A CASE environment is a collection of CASE tools and workbenches that supports the software process.

12. Differentiate between Verification and Validation.

Answer: Verification and Validation are the two important processes of software testing. Verification is concerned with whether the software is being built right. This is done to ensure that software conforms to its specification. Validation is concerned with whether the right software is being built. This is done to ensure that software has functionalities the users really require.

13. What are different methodologies used in black box testing?

Answer: Decision Table-Based Testing, Cause–Effect Graphs in Functional Testing, Equivalence Partitioning and Boundary Value Analysis are some methodologies used in black box testing.

14. Differentiate between black box testing and white box testing.

Answer: Internal logic of the software is not looked into in black box testing. The software or software component to be tested is considered as a 'black box'. The testing is done by entering input data to check whether the output obtained is per requirement or not. The test cases are designed from user requirements.

G) In contrast, the inner structural and logical properties of the program are looked into for verification in white box testing. It is also called Clear Box Testing, Glass Box Testing and Structural Testing.

15. What are different methodologies used in white box testing?

Answer: Statement Coverage, Branch/Edge Coverage, Path Coverage, Data Flow Testing, Mutation Testing are some testing methods used in white box testing.

16. What is unit testing?

Answer: The software units are the independent executable components of an application. In unit testing, these software units are tested in isolation. It is usually done by programmers who write the code.

17. What is integration testing?

Answer: Integration testing is done to check whether the modules when integrated together as a system are able to function as required or not. Various types of integration testing are: (1) Big-Bang Integration Testing, (2) Incremental Integration Testing, (3) Bottom-Up Integration Testing, (4) Top-Down Integration Testing, (5) Sandwiched Testing, (6) Backbone Integration Testing and (7) Thread Integration Testing.

18. What are different levels of testing in OO methodology?

Answer: In OO methodology testing is done at various levels as given below.

- i. The algorithm level testing is done to test method on some data.
- ii. The class level testing is concerned with the interaction between the attributes and methods for a particular class.
- iii. The cluster level testing considers interactions between groups of cooperating classes.
- iv. The system level testing is done to test the system as a whole.

19. Differentiate between Alpha testing, Beta testing and Acceptance testing.

Answer: Alpha testing, Beta testing and Acceptance testing are three major types of system testing. Alpha testing refers to the system testing carried out by the test team within the developing organization whereas Beta testing is the system testing done by a select group of users. These select group of users act as a third party and give their constructive feedback to the developers. Acceptance testing is the system testing done by the customer(s) to determine whether they should accept the delivery of the system or not.

20. How does Product metrics differ from Process metrics?

Answer: Product Metrics are the measures of different characteristics of software product such as size, complexity, quality and reliability of software. Process Metrics are the measures of different characteristics of software development process such as efficiency of fault detection.

21. Differentiate between Internal Metrics and External Metrics.

Answer: The metrics used for measuring properties of software that are viewed as of greater importance to the software developer are called internal metrics, e.g. Line of Code (LOC). The metrics used for measuring properties of software that are viewed as of greater importance to the user are called external metrics, e.g. reliability, functionality, usability, performance etc.

22. What are different categories of functions of an application according to IFPUG FP Metrics?

Answer: According to IFPUG FP Metrics, the functions used in an application can be categorized into five types. These are: (1) External Inputs, (2) External Outputs, (3) Queries, (4) External Interfaces and (5) Internal Files.

23. Make a comparison between FP Metrics and LOC.

Answer: The main difference between FP metrics and LOC are as under.

- i. FP metrics is based on specification whereas LOC is based on analogy.
 - ii. FP metrics is independent of programming language whereas LOC is dependent on programming language being used.
 - iii. FP metrics is user oriented whereas LOC is design oriented.
- H) The LOC of an application can be estimated from FP and vice versa.

24. State the usefulness of process metrics.

Answer: The modern management approach emphasizes that if the process is right and it is rightly managed, the output (product) will be right. The process metrics indicates whether the process being followed for development of software is right and whether the process is being managed well. Hence, Process metrics helps in the management of software development project.

25. Name and define three measures of software design complexity.

Answer: The three measures of software design complexity are: (1) Structural complexity, (2) Data complexity and (3) System complexity.

I) The structural complexity of a module is determined in terms of number of subordinate modules that are immediately followed and directly invoked by the module (Fan out). Data complexity provides an indication of the complexity of the internal interface of a module in terms of number of inputs and outputs passed to and from it. System complexity is the sum of structural and data complexity.

MODULE – IV

(Software Project Estimation, Management and Quality)

1. What are the important parameters of software projects?

Answer: The cost, time, effort and resources required for development are the important parameters of software projects.

2. List different techniques of software project estimation.

Answer: There are five major techniques of software project estimation. These are:

- i. Estimation by Expert Judgement
- ii. Estimation by Analogy
- iii. Estimation by Available Resources
- iv. Estimation by Software Price
- v. Estimation by Parametric Modeling

3. Write the application of Delphi method for software estimation.

Answer: Delphi method is a systemic procedure to obtain the best solution from the collective wisdom of a group of experts. In this method the views of group members are collected, exchanged, organized and refined in a planned manner for estimation of a software project. The activities of the group are well organized and usually any one member from the group acts as the coordinator for this purpose.

4. Name the three categories into which software development projects were classified by Boehm.

Answer: Boehm classified software development projects into three categories based on the development complexity. These are: (1) Organic software, (2) Semidetached software and (3) Embedded software.

5. Name the three models/stages of COCOMO-81.

Answer: In COCOMO 81, the estimation of software projects is done in three stages/models. These are: (1) Basic COCOMO, (2) Intermediate COCOMO and (3) Complete COCOMO.

6. How do Basic COCOMO, Intermediate COCOMO and Complete COCOMO models differ from one another?

Answer: The Basic COCOMO model determines a rough estimate of effort and time required for software development based on software size. However, besides the software size, there are various other factors that affect the software project. The Intermediate COCOMO model recognizes 15 factors that affect the software

project. These 15 factors are called cost drivers. The rough estimate made by Basic COCOMO mode is refined by Intermediate COCOMO model based on ratings for each of the 15 cost drivers.

J) The Basic and Intermediate COCOMO models consider a software product as a single homogeneous entity. However, Complete COCOMO considers that a computerized system consists of a number of subsystems, and the attributes of each subsystem is different from the other. Complete COCOMO uses different Effort Multipliers for each subsystem as well as phase of project.

7. How does COCOMO-II differ from COCOMO-81?

Answer: The COCOMO 81 model was developed in 1981. It was revised in 2000 to suit the requirement of modern software development projects. This new model is called COCOMO II. COCOMO-II has only two main models for estimation of effort. These are: (1) Early design model, and (2) Post-architecture model

K) COCOMO-81 categorizes the software project as organic, semidetached or embedded type, but the COCOMO-II uses the 'Scaling Factors (SF)'. The Early design COCOMO model uses 7 cost drivers and Post-architecture COCOMO model uses 17 cost drivers against 15 cost drivers used in COCOMO-81 model.

8. What are different types of plans used in software projects?

Answer: Software development plan, Testing and Validation plan, Quality assurance plan, Configuration management plan, Maintenance plan and Human resource plan are different types of plans used in software projects.

9. Name some important attributes of software project that need to be correctly estimated for effective planning.

Answer: Size, Cost, Effort and Duration are important attributes of any software project that need to be correctly estimated for effective planning.

10. What is Work Breakdown Structure (WBS)?

Answer: A large project is complex and difficult to manage. A complex project is made manageable by breaking it down into individual components and organizing the work components in a hierarchical structure. This is called WBS of the project. WBS is a hierarchical depiction of all components of work required to accomplish the entire work of the project.

11. What is the purpose of network diagram in project management?

Answer: Different activities making up a project, their estimated durations, and interdependencies between activities can be effectively represented by activity network diagram. This helps in better understanding of the project.

12. What are two types of network diagram? How do they differ from each other?

Answer: There are two types of network diagram for showing activities and their relationships. These are: (1) Activity on Nodes (AON) and (2) Activity on Arrows (AOA). In AON network diagram, activities are represented by circles or ovals, and the 'arrows' represent the sequence in which the activities have to be carried out. In AOA network diagram, activities are represented by arrows. The nodes represent the events that are either the beginning or end of activities. Sometimes, dummy activities are required to be included in AOA network diagram to represent precedence relationship between activities. But dummy activities are not necessary for the AON network diagram.

13. What are different types of dependency relationships between activities that can be shown in network diagram?

Answer: There can be four types of dependencies between activities. These are: (1) Finish-to-start (FS), (2) Start-to-start (SS), (3) Finish-to-Finish (FF) and (4) Start-to-finish (SF).

14. What is the main difference between Critical Path Method (CPM) and Program Evaluation and Review Technique (PERT)?

Answer: CPM is a deterministic model. It assumes that activity duration is certain and is estimated correctly. Hence, CPM is a suitable method for analyzing the project where there is some amount of certainty in execution of activities. In contrast, PERT is based on probability concepts. It uses three time estimates, namely, optimistic time, most likely time and pessimistic time to determine expected time of an activity.

- 15.** What are the three measures of Earned Value approach to assess the status of a project?

Answer: Budgeted Cost of Work Scheduled (BCWS), Actual Cost of Work Performed (ACWP) and Budgeted Cost of Work Performed (BCWP) are the three measures of Earned Value approach to assess the status of a project.

- 16.** What is Gantt Chart?

Answer: A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of bar of an activity is proportional to its duration. The chart depicts the schedule of various activities of a project. It is also used for depicting the progress of the project.

- 17.** What are the various categories of risks associated with software project management?

Answer: The various risks associated with software project management can be grouped into different categories. These are: (1) Schedule-Related Risks, (2) Financial Risks, (3) Technical Risks, (4) Operational Risks and (5) Other Unavoidable Risks

- 18.** Differentiate between Risk Assessment and Risk Control.

Answer: Risk Assessment is concerned with identifying the possible causes that can impede the software development, estimating the adverse effect of these causes and the probability of their occurrence.

L) Risk Control is concerned with coming up with techniques and strategies to mitigate the major risks that are identified through risk assessment.

- 19.** What is Software Configuration Management (SCM)?

Answer: SCM is the task of tracking and controlling changes in the software. It is concerned with managing change in the software product during its development and future revisions/modifications.

- 20.** Differentiate between Software Version and Software Revision.

Answer: Sometimes a minor bug may be noticed in the software. Hence, the bug is fixed and a new revision of software released. Sometimes a new revision is also created for minor enhancements to the functionality, usability etc. On the other hand, a new version of software is created when there is a significant change in functionality or technology of the software. The release of software is often specified as 'm.n'. It means the software is of version 'm', release 'n'.

- 21.** Define quality.

Answer: Quality is defined by ISO as 'Totality of features and characteristics of product/service that has ability to satisfy stated and implied needs of a customer'.

- 22.** According to Juran, what are the three major processes referred to as 'quality trilogy' that are necessary to bring about quality?

Answer: Juran focused on three major processes to bring about quality, which he referred to as 'quality trilogy'. These three processes are: (1) Quality control, (2) Quality improvement and (3) Quality planning.

- 23.** What are different categories of costs associated with Cost of Quality?

Answer: There are four broad categories of costs associated with Cost of Quality. These costs are: (1) Internal failure costs, (2) External failure costs, (3) Appraisal costs and (4) Prevention costs.

24. Write about the concept of ‘Zero Defect’.

Answer: Crosby observed that the mindset of people is a major obstacle in prevention of defects. He further observed that in work life people are conditioned to believe that errors are inevitable. Hence, Crosby emphasized that acceptance of defects is the major cause of defects. Therefore, zero defect should be the only performance standard of quality.

25. What is ‘Quality Assurance’?

Answer: Quality assurance is a set of activities, action plan and review of software products and related documentation carried out to ensure that the system meets the specifications and requirements for its intended use.

26. What is ‘Quality Audit’?

Answer: Quality Audit is a fundamental Quality Assurance technique. It examines a process and/or a product to compare its compliance to established procedures and standards. Its purpose is to assure that proper control procedures are being followed, the required documentation is being maintained and the status reports accurately reflect the status of the activities.

27. Name some management initiatives for quality improvement.

Answer: There are various management initiatives for quality improvement. Some of these are: (1) Total Quality Management (TQM), (2) Process Standards such as ISO 9001:2000, ISO 12207, CMM, (3) Total Productive Maintenance, (4) Lean Systems and (5) Six Sigma.

28. What are the various levels of maturity in CMM?

Answer: CMM specifies five levels of maturity. These are:

- M) Level 1 – Initial State
- N) Level 2 – Repeatable State
- O) Level 3 – Defined State
- P) Level 4 – Managed State
- Q) Level 5 – Optimized State

SOFTWARE MAINTENANCE

The computer is a machine to boost the efficiency of mental work. The computer has two aspects: (1) hardware and (2) software. The term hardware describes the physical aspects of computers and related devices. Detailed instructions in a form understandable to computer hardware are called software. Once installed, software applications are often used for many years. Unlike physical objects (hardware), software does not get damaged or deteriorate with use and passing of time. Hence, maintenance of software is different from maintenance of physical objects. Software maintenance is the periodic modification of software to suit the changing needs of users, organizations or environment. The definition of software maintenance by IEEE is as follows:

The software maintenance is modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

A substantial proportion of the resources expended within the Information Technology industry go towards the maintenance of software systems. In various recent studies, it is observed that maintenance consumes about 50% to 80% of total life cycle costs, and typically consumes about 65% of total life cycle costs.

C.1 NEED FOR SOFTWARE MAINTENANCE

Our environment is never static. Social, political, economic and technological changes are continuously taking place in our environment. Hence, software needs to be modified from time to time to keep it up to date with environment changes and changing user requirements. Maintenance is the last stage of the software life cycle. The software should keep pace with organizational developments. In this sense, maintenance is an ongoing process. When the hardware platform is changed, the software needs to be modified to make it compatible with hardware. The software also needs modification, whenever the support environment of software is changed. For instance, when the operating system is changed, the software may also need certain modifications. Usually, every software product continues to evolve after its development through maintenance efforts.

C.2 TYPES OF SOFTWARE MAINTENANCE

There are basically four types of software maintenance:

- i. Corrective maintenance
- ii. Adaptive maintenance
- iii. Perfective maintenance
- iv. Preventive maintenance

Corrective maintenance deals with the repair of faults or defects that are not detected during testing but found later when the software is in actual use. A defect can result from design errors, logic errors and coding errors. These errors, sometimes called 'residual errors' or 'bugs', undermine the performance of the software. A software application may perform correctly in most cases but may fail in some rare exceptional conditions. The need for corrective maintenance is usually initiated by bug reports drawn up by the end users. Corrective maintenance deals with fixing bugs in the code.

Adaptive maintenance consists of adapting software to changes in the environment, such as the hardware or the operating system. The term environment refers to the totality of all conditions that act from outside upon the system. For example, accounting software may need some modification if there is some change in business rules, government policies, taxation system, work patterns, hardware, operating platforms etc. Adaptive maintenance deals with adapting the software to new environments.

Perfective maintenance is concerned with enhancement of system by adding some new features. Examples of perfective maintenance include software enhancement such as modifying the payroll program to allow deduction of insurance premium from salary, adding a new report in the sales analysis system, improving a user interface to make it more user-friendly, adding an online HELP command etc.

Preventive maintenance deals with updating documentation and improving the modular structure of the software to make it more maintainable. The corrective maintenance is the 'traditional maintenance' while the other types can be considered as 'software evolution.'

C.3 MAINTENANCE PROCESS MODELS

There are several approaches to software maintenance. These approaches have different features. The selection of software maintenance approach is done based on specific needs of the maintenance problem.

C.3.1 Quick-fix Model

A typical approach to software maintenance is to work on code first, and then make the necessary changes to the accompanying documentation. This approach is captured by the quick-fix model. The model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. This is usually done by associating a few members of the original development team. The availability of a working old system facilitates the task of the maintenance team as they get a good insight into the working of the old system. They can also compare the working of their modified system with the old system. Also, debugging of error becomes easy as the program traces of both the systems can be compared to localize the bugs.

The quick-fix model is an ad hoc approach. Its goal is to identify the problem and then fix it as quickly as possible without paying any attention to the long-term side effects of the fixes. Ideally, after the code has been changed the requirement, design, testing and any other form of available documents impacted by the modification should be updated. However, these necessary updatings are often neglected. The changes are often made without proper planning, design, impact analysis and regression testing. This is due to time and budget pressure from the customers/users who expect software to be modified quickly and cost-effectively. Repeated modifications to software through quick-fix approach may demolish the original design, and thus make future modifications more difficult and costly.

C.3.2 Iterative-enhancement Model

Evolutionary life cycle models suggest an alternative approach to software maintenance. These models are based on the idea that the many requirements of a system are gathered and understood gradually while the system is operative

for some period of time. Iterative-enhancement model is an evolutionary model. According to this model, systems should be developed and then enhanced iteratively based on the feedback of users. This model was adapted from development to maintenance. The model has three stages. First, the system has to be analyzed. Next, proposed modifications are classified. Finally, the changes are implemented. A key advantage of the iterative-enhancement model is that documentation is kept updated as the code changes. This model is not effective when the documentation of the system is not complete, as the model assumes that a full documentation of the system exists.

C.3.3 Reuse-oriented Model

The reuse-oriented model is another popular approach to software maintenance. It views maintenance as a particular case of reuse-oriented software development. The reuse-oriented model assumes that existing program components could be reused. The steps for the reuse model are identifying the parts of the old system, which have the potential for reuse, fully understanding the system parts, modifying the old system parts according to the new requirements and integrating the modified parts into the new system. The reuse-oriented model is effective when there is a repository of documents and components of existing and earlier versions of the system to be modified and also of other systems in the same application domain. This makes reuse well documented. It also promotes the development of more reusable components.

C.3.4 Software Reengineering Model

An approach called 'Software Reengineering' is a much preferred approach for software maintenance projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. During the reverse engineering, the old code is analyzed to extract the module specifications. The module specifications are then analyzed to produce the design. The design is analyzed to obtain the original requirement specification. The change requests are then applied to this requirement specification to arrive at the new requirement specification. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. This approach produces a better structured design compared to the original product and produces good documentation. However, this approach is more costly than the quick-fix approach.

All of these models have their strengths and weaknesses. Therefore, usually more than one model is necessary for all maintenance activities. The best approach is to combine the models when required.

C.4 SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirement specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

A legacy system may be defined as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured code, use of obsolete technology and lack of knowledge about the product. Most of the legacy systems were developed a long time back. However, a recently developed system can also be considered a legacy system if it has poor design and documentation. Even well-designed software becomes legacy software as its structure degrades due to a series of changes made through maintenance efforts.

The first stage of reverse engineering usually focuses on formatting the code to improve its readability, structure and understandability. Many legacy software products have complex control structure and indiscreet variable names. These codes are difficult to comprehend. Assigning meaningful variable names makes the code easier to comprehend. Complex nested conditionals in the program can be replaced by simpler conditional statements or case statements.

After formatting of code, the process of extracting the code, design and the requirement specification can begin. Full understanding of the code is needed to extract the design. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

C.5 SOFTWARE REENGINEERING

Software reengineering is the process of reorganizing and modifying the existing software systems to make them more maintainable. Software reengineering process is shown in a block diagram in the figure below.

C.6 PROBLEMS OF SOFTWARE MAINTENANCE

In many organizations, software maintenance work is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies, software maintenance is still mostly carried out as fire-fighting operations, rather than in a systematic and planned manner.

Software professionals generally consider maintenance work as a routine job that does not require much creativity and scope of learning. Thus, software professionals usually prefer to work on the development of new applications to doing maintenance of existing applications. In many organizations, maintenance is an entry-level programming job. Programmers must advance to higher positions to be given assignments in new system development.

Therefore, organizations often do not employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, often the work involved is not less challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

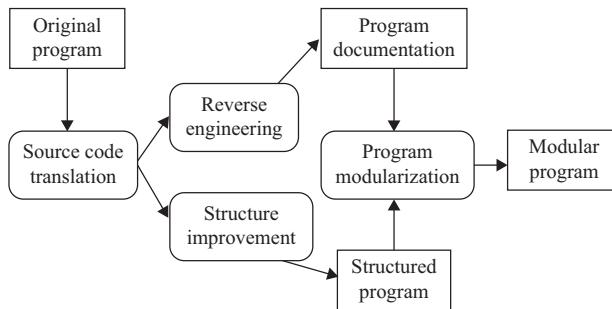


Figure C.1 The software reengineering process

C.7 SUCCESS FACTORS OF SOFTWARE MAINTENANCE

Some software applications are easier to maintain than the others. The maintainability of software is an important quality characteristic. Characteristics of software affect software maintenance effort. Larger systems require more

maintenance effort than smaller systems. This is because larger systems are more complex in terms of the variety of functions they perform. The length of the source code is the main determinant of software development cost as well as maintenance cost. Older systems require more maintenance effort than new systems, because software systems tend to grow larger and become less organized with changes. Some important factors that are found to affect software maintainability are as under.

- Type of application, i.e. Organic, Semidetached or Embedded software
- Programming language used
- Degree to which the program is structured
- Number of input/output data items
- Size of the software application
- Age of the software application

The success of software maintenance activities depend on several factors. Some of the factors are given below:

- Extent of modification required in the software
- Conditions of the existing product (i.e. how structured or well documented it is)
- Resources available for maintenance
- Risks involved in the project
- Use of structured techniques
- Use of automated tools
- Technical competence and experienced maintenance personnel

When only some minor changes are needed to a software product, the code can be directly modified and the changes appropriately reflected in all the documents. But for major modifications, elaborate activities are required. Usually, for maintenance projects for legacy systems, software reengineering process with an emphasis on reuse can be followed.

This page is intentionally left blank.

COMPONENT-BASED SOFTWARE ENGINEERING

Time and cost are two very important aspects of software engineering. Consequently, there is a growing demand for a new, efficient, and cost-effective software development paradigm. The component-based software development (CBSD) approach is one of the most promising solutions for software development. 'CBSD' is a relatively new Software Engineering methodology. It is based on building software systems from reusable components. A software component is defined as a unit of composition that can be independently exchanged in the form of an object. The use of reusable components reduces development time and cost. Hence, this concept has attracted much attention among software developers and its popularity is growing very fast.

D.1 WHAT IS A SOFTWARE COMPONENT?

A software component may be defined as a unit of composition with contractually specified interfaces and explicit context dependencies only. It is an independent software package that provides functionality via well-defined interface. A software component has the following features.

1. It carries out a set of related services or functions
2. It offers well-defined interfaces through which it can be integrated with other components or systems
3. Its functionalities or services are accessible through its interface only. Hence, it is much like an object
4. It is reusable
5. It is independent – it is possible to deploy the component without having to use any other specific component.

D.2 CHARACTERISTICS OF CBSD

The main characteristics of CBSD are as follows:

- Black-box reuse: The internal structure of the component is not visible, but it is reusable.
- Reactive-control and component's granularity: Components consist of object-oriented classes that are reusable at the instance level. It is in a physical package of executable software with published interfaces that makes it reactive-control.
- Uses rapid application development (RAD) tools such as JRun, JDesigner Pro etc. JDesigner Pro creates Java Applets with minimal coding. JRun is a Java Servlet and Java ServerPages (JSP) engine
- Contractually specified interfaces: These interfaces are very specific and help in quick building of the component system
- There are large varieties of software components available in the market, which can be procured based

on needs of the software being developed. These commercially off-the-shelf components are popularly called COTS.

CBSD incorporates many of the characteristics of the spiral model. It is also evolutionary in nature, demanding an iterative approach to the creation of software. Here, the modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented (OO) classes.

D.3 CBSD PROCESS

Regardless of the technology that is used to create the components, the components-based development model incorporates the following steps (implemented using an evolutionary approach).

1. Component Identification from Repository
2. Component Selection
3. Component Integration
4. Component Evolution
5. Putting in Component Repository

CBSD has two parallel activities

(1) Domain Engineering and (2) Component-based development

In domain engineering, the applications are explored to identify functions, features, and data components for developing reusable components. Once these components are tested for certain application and found to be OK, these are put in a component repository or library. The components from the library are suitable for reuse. Based on the requirements and architecture of the system being developed, the components can be picked up from the component library.

Domain engineering consists of domain analysis, domain model, and model analysis. This is followed by architecture design. After testing is successful, the components are kept in the Repository of Reusable Components for future reuse at a later stage.

CBSD can use either COTS components or internally developed components for development purpose.

The requirement stage requires a complete set of requirements so that as many components as possible can be identified for reuse. Requirements are refined and modified early in the process depending on the components available. If the user requirements cannot be satisfied from available components, the related requirements that can be supported are considered. The users are often willing to make changes in their requirements if this means cheaper or quicker system delivery. There is a further component search and design refinement activity after the system architecture has been designed. In fact, some usable components may turn out to be unsuitable or do not work properly with other chosen components. Finally, the development involves a composition process where the discovered components are integrated. It integrates the components with the component model infrastructure. This is known as developing a 'glue code'. It is done to reconcile the interfaces to make the components compatible. During the architectural design, the software development team may decide on a component model, although, for many systems, this decision may be made before the search for components begins. Component identification is an important activity of the CBSD process. It involves a number of sub-activities, as shown in the figure below.

The first stage in identifying components is to look for components that are available locally or from trusted suppliers. Component markets have not yet fully developed. Components are mostly available off-the-self only in some popular application domains. For specific requirements, the software development companies can also build their own database of reusable components.

Once the component search process has identified the candidate components, specific components from this list can be selected. Usually, it is possible that requirements will require different groups of components. Hence, the software development team has to decide which component compositions provide the best coverage of the requirements. Once the components are selected for possible inclusion in a system, these are validated. Component validation is done to check whether the components behave as desired. It involves developing a set of test cases for

the component. Sometimes test cases are supplied with the component for checking purpose. After component validation is done, it can be used for development.

Component identification process should be followed by customization and integration as part of the complete life cycle of CBSD. It includes two main parts:

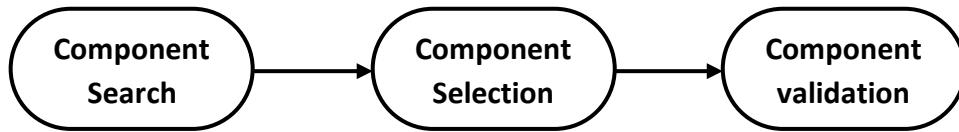


Figure D.1 Component identification process

1. Evaluation of each candidate COTS component based on the functional and quality requirements, which will be used to assess the component.
2. Customization of those candidate COTS components that should be modified before being integrated into new component-based software systems.

Component Integration involves decision-making on how to provide communication and coordination among various components of a target software system.

Continuous evolutions of components happen with time. As more and more components are put into the Component Repository, it goes on expanding over time.

D.4 SOME POPULAR COMPONENT TECHNOLOGIES

There are many component systems currently available in the market. Some of these are described below

D.4.1 Component Object Model

Microsoft Component Object Model (COM) is a general architecture for component software. It is based on Windows and Windows NT platform. However, it is a language-independent component-based application. COM objects can be created with a variety of programming languages. OO languages, such as C++, provide programming mechanisms to simplify the implementation of COM objects. The family of COM technologies includes COM+, Distributed COM (DCOM), and ActiveX Controls. Microsoft provides COM interfaces for many Windows application programming interfaces such as Direct Show, Media Foundation, Packaging API, Windows Animation Manager, Windows Portable Devices, and Microsoft Active Directory. COM is used in applications such as the Microsoft Office Family of products. For example, COM OLE technology allows Word documents to dynamically link to data in Excel spreadsheets.

D.4.2 Enterprise JavaBeans

Enterprise JavaBeans (EJB) technology is the server-side component architecture for Java Enterprise Edition (J2EE) Platform. The Java platform offers an efficient solution to portability and security problems. Java provides a universal integration and enabling technology for enterprise application development. The portability, security, and reliability of Java are well suited for developing robust and secure web-enabled applications that are independent of operating systems. EJB extend all native strengths of Java including portability and security into the area of component-based development. EJB technology enables rapid and simplified development of distributed, transactional, secure, and portable applications based on Java technology.

D.4.3 Common Object Request Broker Architecture

Common Object Request Broker Architecture (CORBA) is a vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol, a CORBA-based program from any vendor can interoperate with another CORBA-based program regardless of any other operating system, programming language, and network. CORBA is an open standard for application interoperability. It is supported by the Object Management Group (OMG), an organization of over 400 software vendor and object technology user companies. The most important part of a CORBA system is the Object Request Broker (ORB). The ORB is the middleware that establishes the client-server relationships between components. Using an ORB, a client can invoke a method on a server object. CORBA offers a consistent distributed programming and run-time environment over common programming languages, operating systems, and distributed networks. Hence, it is widely used in OO distributed systems including component-based software systems.

A comparison of three component technologies is given in the table below.

It is obvious that CBSD is modeled and implemented in an OO paradigm/language. Therefore, while understanding the components, the traditional techniques in the OO paradigm such as OO framework, design patterns, architecture patterns, and meta-patterns are very important.

Table 1. Comparison of component technologies

| | COM | EJB | CORBA |
|-------------------------------|--|--|---|
| Platform dependency | Platform dependent. It is supported by Microsoft Windows platform | Platform independent | Platform independent |
| Language dependency | Language independent | Language-dependent; applicable only for Java | Language independent |
| Development environment | Supported by a wide range of strong development environments | Emerging | Underdeveloped |
| Binary interfacing standard | A binary standard for component interaction is the heart of COM | Based on COM; Java specific | Not binary standards |
| Compatibility and portability | Not having any concept of source-level standard of standard language binding | Portable due to Java language; but not very compatible | Particularly strong in standardizing language bindings; but not so portable |
| Modification and maintenance | Need extra modification and maintenance | Easier modification and maintenance | Need extra modification and maintenance |
| Strengths | Traditional desktop applications | Clients/server system on web | Traditional enterprise computing |

INDEX

.NET Framework 177

A

Abstraction 112

Access Layer Prototypes 165

Activity Diagram 134

Activity Network 290

Adaptive maintenance 378

Aggregation 115, 145, 155

Antipatterns 171

Application software 3

Architectural Design 162

Architectural Design metrics *see* High-level design metrics

Architectural pattern 170

Association 117, 145

Attributes 146

Attribute hiding factor(AHF) 261

Attributes inheritance factor(AIF) 260

Availability 254

Axiom *see* design axiom

B

Backbone Integration Testing 236

Baldridge (MBNQA) Model 312

Bang Metrics 252

Behavioural model 56

Behavioural view 124

Big-bang Testing 234

Black Box Testing 220

Blob 173

Booch Model 118

Bottom-up Integration Testing 234

Boundary Value Analysis 223

Branch Coverage 227

C

Capability Maturity Model (CMM) 15, 31, 323

CASE Environments 211

CASE Tools 210

CASE Workbenches 211

Cause-Effect Graphs 221

Characteristics of User Interface 182

CK Metrics Suite 259

Class 108, 137

Class Classification 140–143

Class Design 165

Classification of Software Projects 270

COCOMO II 276–278

COCOMO Model 271–275

Coding Guidelines 203

Coding Norms 202

Coding Standards 200

Cohesion 57, 90, 165, 257

Cohesion Metrics 258

Collaboration Diagram 136

Collaborative Requirement Gathering 41

COM 365

Command Language-Based Interface 184

Component-based GUI *see* Widget

Component-based Software Development (CBSID) 383

Component-based Software Engineering(CBSE) *see* CBSID

Component Diagram 167

Component Object Model *see* COM
 Condition coverage 227
 Configuration Management 302–303
 Control Class 139
 Control flow graph (CFG) 229
 Control flow-based design 9
 CORBA 386
 Corrective maintenance 378
 Coupling 90,164
 Coupling between objects (CBO) 260
 Coupling Factor (CF) 261
 Coupling Metrics 258
 CRC Card 143
 Critical Path Method (CPM) 294

D

Data abstraction 113
 Data Design 97
 Data Dictionary 76–80
 Data Flow Diagram 67–71, 93
 Data Flow-based Testing 229
 Data flow-oriented design 10
 Data hiding 113
 Data structure-oriented design 10
 Dataflow diagram *see* DFD
 Debugging 220
 Decision Table 72
 Decision Table-based Testing 221
 Decision Tree 76
 Decomposition 56, 90, 108
 Defect Density 254
 Defect Estimation 256
 Delphi method 267
 Deming Prize 312
 Deployment Diagram 168
 Depth of Inheritance Tree (DIT) 259
 Design Axioms 164
 Design Framework 163
 Design Metrics 257–258
 Design pattern 172
 Detail Design 98
 DFD 56, 67
 Direct manipulation interface 182, 185
 Documentation Standard 206
 Domain Analysis 142

E

Earned Value Monitoring 297
 EFQM Business Excellence Model 314
 EJB 386
 Encapsulation 113
 Enterprise JavaBeans *see* EJB
 Entity Class 138
 Entity object 138
 Entity Relationship Model 80
 Equivalence Partitioning 223
 ER Diagram 80
 Estimation Approaches 266
 Estimation by Analogy 268
 Estimation by Parametric Modelling 269
 Estimation of Software Project 265
 Estimation Techniques 267
 Evolutionary model 323
 Expert judgment technique 267
 Exploratory programming 9

F

Fan in 259
 Fan out 257, 259
 Feature Point Metrics 251
 Final Functional Requirements 44
 Flow chart 9
 Framework 175
 Function Oriented Approach 54
 Function Point Metrics 249
 Functional requirements 45

G

Gantt Chart 297
 Generalization 130
 Generations of Computers 5
 Generations of software 6
 Graphical User Interface *see* GUI
 GUI 185

H

Halstead's Metrics 252, 256
 Halstead's software science 252
 High-level Design Metrics 257
 HIPO Documentation 104

I

- Inception 37
- Incremental Integration Testing 234
- Inheritance 111
- Initial Technical Requirements 44
- Initial User Requirements 44
- Integration Testing 233
- Interaction Diagrams 133, 136
- Interface Class 138
- ISO 12207 321
- ISO 9000 320
- Iterative Waterfall Model 28
- Iterative-enhancement 379

J

- Jacobson's Model 120

L

- Lack of Cohesion in Methods (LCOM) 260
- Life cycle model 22
- Lines of Code (LOC) 248
- LOC Metrics 249
- Lower CASE Tools 211

M

- Maintenance process models 378
- McCabe's Cyclomatic Complexity 229
- Mean Time to Failure (MTTF) 254
- Mean Time to Repair (MTTR) 254
- Menu-based interface 185
- Message 111
- Metaclasses 118
- Metaphor 194
- Method hiding Factor (MHF) 261
- Method Inheritance Factor (MIF) 260
- Method over loading 114
- Method overriding 114
- Methods 147
- Metrics for OO Design (MOOD) 260
- Mobile Web Engineering 337
- Mode-based Interface 187
- Mode-less Interface 187
- Modeling Techniques 118
- Model-View-Controller (MVC) 175

- Modularity 90, 321
- Multiple inheritance 112
- Mutation testing 232

N

- Naming Convention 202
- Non-functional Requirement 163
- Noun Phrase Approach 140
- Number of Children (NOC) 259

O

- Object 108
- Object Identifier 117
- Object-oriented Metrics 259
- Object-oriented Testing 61, 236
- Object Persistence 118
- Object References 117
- Object-oriented Analysis (OOA) 59, 127
- Object-oriented Analysis and Design 58
- Object-oriented Design (OOD) 10, 60, 161
- Object-Oriented Maintenance 61
- OMT 119
- Organizational Processes 323

P

- Package 60
- Path Coverage 228
- Patterns 169–171
- Perfective maintenance 378
- Performance testing 242
- PERT 295
- Polymorphism 113
- Polymorphism Factor(PF) 261
- Preventive maintenance 378
- Primary Processes 321
- Process Improvement 317
- Process Metrics 248, 255
- Process Specification 72
- Process Standard 29, 320
- Product Metrics 247
- Program Flowchart 98
- Programming Languages 7, 8
- Project estimation techniques *see* estimation

Project Failure 298

Project Manager 285

Project Plan Types 286

Project Planning 285

Project Scheduling 290

Prototyping Model 26

Pseudocode 102

Q

QC Tools 312

Quality Assurance 315

Quality Assurance Activities 316

Quality Concept 307

Quality Gurus 309

Quality Metrics 253

Quality Models 312

Quality system 320

Quick-fix Model 378

R

Record Review 41

Regression testing 243

Relationships 115–117, 145

Reliability 308

Requirement Elaboration 44

Requirement Elicitation 38–43

Requirement Negotiation 46

Requirement Validation 47

Resource Metrics 248

Response for a Class (RFC) 260

Reverse engineering 380

Risk Management 299–301

Rumbaugh's OMT 119

S

SAD 54

Sandwiched Testing 236

Scenario-based Testing 241

Scheduling 290

SDLC Models 22

Security Web Engineering 338

Sequence Diagram 136

Six Sigma 317

Software Characteristics 3

Software crisis 11

Software Development Plan 286

Software Documentation 204

Software Evolution 4

Software maintenance 377

Software Metrics 247

Software Process 21

Software Project Management 283

Software Project Parameters 265

Software Quality Assurance 315

Software Quality Management 307

Software Reengineering 379, 380

Software Requirement 36

Software Reverse Engineering 380

Software Revision 305

Software Size Metrics 248

Software Testing 217

Software Version 305

SPICE 31

Spiral Model 29

SRS document 36, 47

Stakeholders 37

State chart diagram 124

State Diagram 134

State Transition Diagram 82

State Transition Testing 239

Statement Coverage 226

Stereotype 169

Stress testing 242

Structure Chart 92–93

Structured Analysis 65

Structured design 89

Structured Flowchart 101

Structured programming 10

Struts frame work 175

Subsystems 162

Support Processes 323

System Boundary 162

System engineering 23

System software 2

System Testing 242

T

Testing Process 218

Textual User Interface 185

Thread Integration Testing 236

Top-down Integration Testing 235

Transaction Analysis 94

Transaction flow testing 241

Transform Analysis 95

TUI 185

Types of Design Classes 167

Types of User Interfaces 181

U

UML 59, 62, 119, 123

Unified Approach to Modelling 123

Unified Modelling Language *see* UML

Unified Process *see* unified approach

Unit Testing 233

Upper CASE Tools 211

Usability Testing 243

Usage mode 207, 215–216, 351

Use-Case 127–133

User documentation 206–209

User Interface Design 181, 194

V

Validation 218

Verification 218

V-Model 25

V-Model for testing 219

W

Waterfall Model 23

Web Design Principles 332

Web Engineering 327–329

Web Engineering Process 330

Web Metrics 333–336

Weighted methods per class (WMC) 259

White Box Testing 225

Widget-based GUI 189

Window Management System 188

Window manager 189

Work Breakdown Structure 287

This page is intentionally left blank.