

Object-Oriented Software Engineering

YOGESH SINGH

Vice Chancellor
The Maharaja Sayajirao University of Baroda
Vadodara

RUCHIKA MALHOTRA

Assistant Professor
Department of Computer and Software Engineering
Delhi Technological University
Delhi

PHI Learning Private Limited

New Delhi-110001
2012

OBJECT-ORIENTED SOFTWARE ENGINEERING

Yogesh Singh and Ruchika Malhotra

© 2012 by PHI Learning Private Limited, New Delhi. All rights reserved. No part of this book may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publisher.

ISBN-978-81-203-4535-5

The export rights of this book are vested solely with the publisher.

Published by Asoke K. Ghosh, PHI Learning Private Limited, M-97, Connaught Circus, New Delhi-110001 and Printed by Rajkamal Electric Press, Plot No. 2, Phase IV, HSIDC, Kundli-131028, Sonepat, Haryana.

Contents

Preface xi

1. Introduction	1–31
1.1 What is Software Engineering? 2	
1.1.1 Program vs. Software 3	
1.1.2 Characteristics of Software 3	
1.2 What is Object Orientation? 6	
1.2.1 Classes and Objects 6	
1.2.2 Messages, Attributes and Methods 8	
1.2.3 Encapsulation 8	
1.2.4 Inheritance 11	
1.2.5 Polymorphism 13	
1.2.6 Responsibility and Abstraction 15	
1.2.7 Object Composition 15	
1.3 Object-Oriented Methodologies 16	
1.3.1 Coad and Yourdon Methodology 16	
1.3.2 Booch Methodology 17	
1.3.3 Rumbaugh Methodology 19	
1.3.4 Jacobson Methodology 19	
1.4 Object-Oriented Modelling 20	
1.5 Some Terminologies 20	
1.5.1 Customers, Developers and Users 21	
1.5.2 Product and Process 21	
1.5.3 Actor, Use Case, Use Case Model and Use Case Scenario 21	
1.5.4 System and Subsystems 21	
1.5.5 Class, Responsibility and Collaboration 22	
1.5.6 Measures, Metrics and Measurement 22	
1.5.7 Software Quality and Reliability 22	

1.5.8	Quality Assurance and Quality Control	23
1.5.9	Verification and Validation	23
1.5.10	Fault, Error, Bug and Failure	24
1.5.11	States and Events	24
1.5.12	Traditional Approach and Object-Oriented Approach	24
<i>Review Questions</i>		25
<i>Multiple Choice Questions</i>		26
<i>Further Reading</i>		31
2.	Software Development Life Cycle Models	32–61
2.1	Conventional Software Life Cycle Models	32
2.1.1	Build-and-Fix Model	33
2.1.2	Waterfall Model	34
2.1.3	Prototyping Model	36
2.1.4	Iterative Enhancement Model	37
2.1.5	Spiral Model	38
2.1.6	Extreme Programming	40
2.2	Object-Oriented Software Life Cycle Models	43
2.2.1	Fountain Model	44
2.2.2	Rational Unified Process	46
2.3	Selection of Software Development Life Cycle Models	54
<i>Review Questions</i>		56
<i>Multiple Choice Questions</i>		57
<i>Further Reading</i>		61
3.	Software Requirements Elicitation and Analysis	62–122
3.1	Case Study: Library Management System	63
3.2	What is Software Requirement?	63
3.2.1	Identification of Stakeholders	64
3.2.2	Functional and Non-functional Requirements	65
3.2.3	Known and Unknown Requirements	65
3.3	Requirements Elicitation Techniques	65
3.3.1	Interviews	66
3.3.2	Brainstorming Sessions	69
3.3.3	Facilitated Application Specification Technique	70
3.3.4	Prototyping	71
3.4	Initial Requirements Document	71
3.5	Use Case Approach	73
3.5.1	Use Cases and Actors	73
3.5.2	Identification of Actors	74
3.5.3	Identification of Use Cases	74
3.5.4	Defining Relationships between Use Cases	75
3.5.5	Use Case Diagram	76

3.5.6	Use Case Description	78
3.5.7	Generation of Scenario Diagrams	80
3.5.8	Creation of Use Case Scenario Matrix	81
3.6	Characteristics of a Good Requirement	82
3.6.1	Correct	83
3.6.2	Unambiguous	83
3.6.3	Complete	83
3.6.4	Consistent	84
3.6.5	Verifiable	84
3.6.6	Modifiable	85
3.6.7	Clear	85
3.6.8	Feasible	85
3.6.9	Necessary	85
3.6.10	Understandable	85
3.7	Software Requirements Specification Document	86
3.7.1	Nature of the SRS Document	86
3.7.2	Organization of the SRS Document	86
3.8	Requirements Change Management	111
3.8.1	Is Change Necessary?	112
3.8.2	Establishment of Baseline	112
3.8.3	Requirements Traceability	112
3.8.4	Change Control	114
	<i>Review Questions</i>	114
	<i>Multiple Choice Questions</i>	118
	<i>Further Reading</i>	121
4.	Object-Oriented Software Estimation	123–173
4.1	Need of Object-Oriented Software Estimation	124
4.2	Lorenz and Kidd Estimation Method	124
4.3	Use Case Points Method	126
4.3.1	Classification of Actors and Use Cases	127
4.3.2	Computing Unadjusted Use Case Points	128
4.3.3	Calculating Technical Complexity Factors	128
4.3.4	Calculating Environmental Complexity Factors	129
4.3.5	Calculating Use Case Points	130
4.4	Class Point Method	134
4.4.1	Identification of Classes	135
4.4.2	Classifying Class Complexity	135
4.4.3	Calculating Unadjusted Class Points	136
4.4.4	Calculating Technical Complexity Factor	137
4.4.5	Calculating Class Point and Effort	137
4.5	Object-Oriented Function Point	140
4.5.1	Relationship between Function Points and Object Points	141
4.5.2	Counting Internal Classes, External Classes and Services	142

4.5.3	Calculating Unadjusted Object Points	144
4.5.4	Adjustment Factors	145
4.6	Risk Management	146
4.6.1	What is Risk?	147
4.6.2	Framework for Managing Risks	147
4.6.3	Risk Identification	148
4.6.4	Risk Analysis and Prioritization	149
4.6.5	Risk Avoidance and Mitigation Strategies	151
4.6.6	Risk Monitoring	151
4.6.7	Estimating Risk Based on Schedule	152
<i>Review Questions</i>		166
<i>Multiple Choice Questions</i>		170
<i>Further Reading</i>		173
5.	Object-Oriented Analysis	174–202
5.1	Structured Analysis versus Object-Oriented Analysis	174
5.2	Identification of Classes	175
5.2.1	Entity Classes	175
5.2.2	Interface Classes	176
5.2.3	Control Classes	176
5.3	Identification of Relationships	180
5.3.1	Association	180
5.3.2	Aggregation	181
5.3.3	Multiplicity	182
5.3.4	Composition	182
5.3.5	Dependency	183
5.3.6	Generalization	183
5.3.7	Modelling Relationships	184
5.4	Identifying State and Behaviour	187
5.4.1	Attributes	187
5.4.2	Operations	189
5.4.3	Example: Issue Book in LMS	190
5.5	Case Study: LMS	194
5.6	Moving Towards Object-Oriented Design	194
<i>Review Questions</i>		195
<i>Multiple Choice Questions</i>		199
<i>Further Reading</i>		201
6.	Object-Oriented Design	203–259
6.1	What is Done in Object-Oriented Design?	203
6.2	Interaction Diagrams	204
6.3	Sequence Diagrams	205
6.3.1	Objects, Lifeline and Focus of Control	205

6.3.2	Messages	205
6.3.3	Creating Sequence Diagrams	208
6.3.4	Creating Sequence Diagram of Use Cases with Extensions	214
6.4	Collaboration Diagrams	222
6.4.1	Objects, Links and Messages	222
6.4.2	Creating Collaboration Diagrams	223
6.5	Refinement of Use Case Description	226
6.6	Refinement of Classes and Relationships	232
6.7	Identification of Operations to Reflect the Implementation Environment	232
6.8	Construction of Detailed Class Diagram	234
6.9	Development of Detailed Design and Creation of Software Design Document	234
6.10	Generating Test Cases from Use Cases	238
6.10.1	Commonly Used Testing Terminology	238
6.10.2	Deriving Test Cases from Use Cases: A Five-Step Process	239
6.11	Object-Oriented Design Principles for Improving Software Quality	244
	<i>Review Questions</i>	254
	<i>Multiple Choice Questions</i>	256
	<i>Further Reading</i>	258
7.	Moving Towards Implementation	260–286
7.1	Activity Diagrams	260
7.1.1	Activities and Transitions	261
7.1.2	Branching	262
7.1.3	Modelling Concurrency	262
7.1.4	Using Swimlanes	263
7.1.5	Uses of Activity Diagrams	265
7.2	Statechart Diagrams	268
7.2.1	States and State Transition	269
7.2.2	Event, Action and Guard Condition	270
7.2.3	Modelling Life Cycle of an Object	271
7.2.4	Creating Substates	272
7.3	Storing Persistent Data in Database	275
7.3.1	Mapping Entity Classes to Database Tables	276
7.3.2	Representing Inheritance in Tables	277
7.4	Implementing the Classes	280
7.4.1	Good Programming Practices	281
7.4.2	Coding Standards	281
7.4.3	Refactoring	282
7.4.4	Reusability	282
	<i>Review Questions</i>	282
	<i>Multiple Choice Questions</i>	284
	<i>Further Reading</i>	286

8. Software Quality and Metrics	287–347
8.1 What is Software Quality? 287	
8.1.1 Software Quality Attributes 288	
8.1.2 Elements of a Quality System 289	
8.2 Software Quality Models 296	
8.2.1 McCall's Software Quality Model 296	
8.2.2 Boehm's Software Quality Model 297	
8.2.3 ISO 9000 299	
8.2.4 ISO 9126 300	
8.2.5 Capability Maturity Model 301	
8.3 Measurement Basics 303	
8.3.1 What are Software Metrics? 304	
8.3.2 Application Areas of Metrics 304	
8.3.3 Categories of Metrics 305	
8.3.4 Measurement Scale 307	
8.3.5 Axiomatic Evaluation of Metrics on Weyuker's Properties 308	
8.4 Analysing the Metric Data 310	
8.4.1 Summary Statistics for Preexamining Data 310	
8.4.2 Metric Data Distribution 312	
8.4.3 Outlier Analysis 313	
8.4.4 Correlation Analysis 319	
8.4.5 Exploring Analysis 319	
8.5 Metrics for Measuring Size and Structure 320	
8.5.1 Size Estimation 320	
8.5.2 Halstead Software Science Metrics 321	
8.5.3 Information Flow Metrics 324	
8.6 Measuring Software Quality 324	
8.6.1 Software Quality Metrics Based on Defects 324	
8.6.2 Usability Metrics 326	
8.6.3 Testing Metrics 327	
8.7 Object-Oriented Metrics 328	
8.7.1 Coupling Metrics 328	
8.7.2 Cohesion Metrics 331	
8.7.3 Inheritance Metrics 334	
8.7.4 Reuse Metrics 335	
8.7.5 Size Metrics 338	
8.7.6 Popular Metric Suites 338	
<i>Review Questions</i> 340	
<i>Multiple Choice Questions</i> 343	
<i>Further Reading</i> 346	

9. Software Testing	348–413
9.1 What is Software Testing? 348	
9.1.1 Verification 349	
9.1.2 Validation 349	
9.2 Software Verification Techniques 349	
9.2.1 Peer Reviews 350	
9.2.2 Walkthroughs 350	
9.2.3 Inspections 350	
9.3 Checklist: A Popular Verification Tool 352	
9.3.1 SRS Document Checklist 352	
9.3.2 Object-Oriented Analysis Checklist 354	
9.3.3 Object-Oriented Design Checklist 355	
9.4 Functional Testing 356	
9.4.1 Boundary Value Analysis 356	
9.4.2 Equivalence Class Testing 370	
9.4.3 Decision Table-Based Testing 380	
9.5 Structural Testing 385	
9.5.1 Path Testing 385	
9.6 Class Testing 393	
9.7 State-Based Testing 395	
9.7.1 Design of Test Cases 396	
9.8 Mutation Testing 397	
9.8.1 Mutation Testing and Mutants 398	
9.8.2 Mutation Operators 400	
9.8.3 Mutation Score 401	
9.9 Levels of Testing 403	
9.9.1 Unit Testing 403	
9.9.2 Integration Testing 404	
9.9.3 System Testing 404	
9.9.4 Acceptance Testing 405	
9.10 Software Testing Tools 405	
9.10.1 Static Testing Tools 405	
9.10.2 Dynamic Testing Tools 406	
9.10.3 Process Management Tools 406	
<i>Review Questions</i> 407	
<i>Multiple Choice Questions</i> 409	
<i>Further Reading</i> 412	
10. Software Maintenance	414–434
10.1 What is Software Maintenance? 414	
10.2 Categories of Software Maintenance 415	
10.2.1 Corrective Maintenance 415	
10.2.2 Adaptive Maintenance 415	

10.2.3	Perfective Maintenance	416
10.2.4	Other Types of Maintenance Activities	416
10.3	Challenges of Software Maintenance	417
10.3.1	High Staff Turnover	417
10.3.2	Flexible Nature of Software	417
10.3.3	Poor Documentation and Manuals	418
10.3.4	Inadequate Budgetary Provisions	418
10.3.5	Emergency Fixing of Bugs	418
10.4	Maintenance of Object-Oriented Software	418
10.5	Software Rejuvenation	421
10.5.1	Reverse Engineering	421
10.5.2	Software Re-engineering	422
10.5.3	Redocumentation and Restructuring	423
10.6	Estimation of Maintenance Effort	424
10.6.1	Belady and Lehman Model	424
10.6.2	Boehm Model	425
10.7	Configuration Management	426
10.7.1	Configuration Identification	427
10.7.2	Configuration Control	427
10.7.3	Configuration Accounting	428
10.8	Regression Testing	428
<i>Review Questions</i>		430
<i>Multiple Choice Questions</i>		431
<i>Further Reading</i>		433
Appendix		435–467
References		469–472
Answers to Multiple Choice Questions		473–474
Index		475–479

Preface

The complexity, criticality and size of the software is increasing every day, and resulting in a situation where the traditional approaches to software development may not be effectively applicable. In the traditional approaches, there is a difficulty to produce reusable and low-cost maintainable software. The popular approach to address such situations can be addressed using object-oriented paradigm. Hence, the software industry is shifting towards development of software using object-oriented concepts and practices at each phase of software development, and object-oriented software engineering has emerged as a separate and important discipline. Object-oriented software engineering emphasizes visual modelling based on real-world objects and entities. Modelling helps in understanding the problems, developing proper documents and producing well-designed programs. Modelling produces well-understood requirements, robust designs, and high-quality and maintainable systems. The models must depict the various views of the system and these models must be verified to check whether they capture the customer's requirements. In this book, we have used Unified Modelling Language (UML) notations for constructing various models at various stages of object-oriented software development.

The worldwide universities and software industries have started realizing the importance of object-oriented software engineering. Most of the universities are now offering a full course on object-oriented software engineering for their undergraduate and postgraduate students in the disciplines of computer science and engineering, software engineering, computer applications, and information technology.

The book focuses on object-oriented software engineering in the context of an overall effort to present object-oriented concepts, techniques and models that can be applied in software estimation, analysis, design, testing and quality improvement. This book is designed for use as a main textbook for a course in object-oriented software engineering and object-oriented analysis and design, and as a useful resource for academicians and software practitioners.

The book is the result of our several years of experience of classroom teaching. It consists of numerous solved examples, chapter-end review questions, multiple choice questions and case studies. The concepts and the models explained and developed in this book are demonstrated using a real-life case study of library management system (LMS).

The main features of this book are as follows:

- It focuses on the basic concepts, need and importance of object-oriented paradigm. It presents the origin and benefits of visual modelling. The basic object-oriented software methodologies are also given.
- It presents traditional and object-oriented software development life cycle models with a special focus on Rational Unified Process model.
- It presents requirements gathering techniques and proposes the development of Initial Requirement Document (IRD). These techniques may help to gather requirements effectively and efficiently in the early phases of software development.
- It introduces object-oriented software estimation techniques which can be used in practice in order to measure the effort of object-oriented software. It addresses important issues of risk management, and two techniques for risk estimation such as Critical Path Method and PERT are explained.
- It presents object-oriented analysis and explains the types of classes, their relationships and structure using Unified Modelling Language (UML) notations.
- It presents object-oriented design including construction of sequence and collaboration diagram as part of design activity. The development of detailed design of classes and identification of operations from sequence diagram is also explained. It promotes the use of use case testing, for generating test cases from use cases.
- It presents activity diagrams and statechart diagrams which are used to model dynamic aspects of the system. These models will help the developers to better understand the system under development.
- It addresses important issues of improving software quality and measuring various object-oriented constructs using object-oriented metrics. In addition, the techniques for cleaning and analysing the metric data are also explained.
- It signifies the importance of object-oriented testing and maintenance of software developed using object-oriented software engineering techniques and methods.

We do acknowledge the significance of feedback of our readers for enriching the contents of the book. We are waiting for microscopic analysis and critical assessment of the contents of the book. Any suggestion for addition, deletion and modification shall be considered positively for the improvement of the book. Every feedback will motivate us to work hard for the next edition for our beloved readers. Till then goodbye!

**Yogesh Singh
Ruchika Malhotra**

1

Introduction

The world has seen many inventions and discoveries in the last 250 years. Which one has affected our civilization the most? Electricity may be the answer. It has become the reason for many applications and comforts. After electricity, what is next? It may be computer and software. Numerous applications are developed and more and more are in the line to change our way of life. Social networking websites like Facebook, Twitter, etc., research scholar platforms like Google Scholar, and email service providing websites like Hotmail, Gmail, Rediffmail, etc. are adding various dimensions to our life. We cannot think of a day without using Internet, without using cell phone, without using laptop for various activities, etc. Software is a driving force for such applications. We may see a large number of software applications with more features and versatility in coming times.

The size and complexity of software are increasing day by day. Conventional approaches of software design and implementation may not be effectively applicable. To handle a large size software development project, we may require different approaches, skill sets, tools and mindsets. We have seen this in civil engineering where constructing a house in a 250 m² plot requires different approaches and tools than constructing a 50-storeyed building. House and multi-storeyed building constructions are handled differently by the architects, structural engineers and civil contractors. The same analogy is valid for software development where a large size software project requires more thorough and systematic approaches. We want to simplify the development process and produce high-quality maintainable software. As we all know, development may take a few years and the same may have to be maintained for many years. A maintainable software may reduce the maintenance cost and high quality may enhance the sustainability of the software.

The necessity of developing and maintaining a large size, complex and varied functionalities software system has forced us to look for new approaches of software design and development. The conventional approaches like “waterfall model” may not be very useful due to non-availability of iterations, no provision of reuse and difficulty in incorporating changing requirements. We

may also build every software system from the scratch that results into a costly software system including very high maintenance cost. An object-oriented paradigm may address such issues. It is becoming popular to design, develop and maintain large size, complex and critical software systems using the object-oriented paradigm. Due to its popularity and acceptability in customers, companies are also releasing the object-oriented versions of their existing software products. Many of the customers expect object-oriented software solutions and request the same for their forthcoming projects.

In this chapter, a framework of object-oriented software engineering is given. The object-oriented concepts are introduced and explained. The object-oriented methodologies are discussed and key terms are defined.

1.1 What is Software Engineering?

The dependency on software is increasing every day and its correct operation is becoming a real challenge for automated modern civilization. In order to handle this challenge, we must have sound engineering principles and processes to produce a good quality software product and such software should also be maintainable. There are two words ‘software’ and ‘engineering’. Engineering forces us to focus on systematic, scientific and well-defined processes to produce a good quality product. The early years’ debate on “whether software development is an engineering discipline?” is over now. We all have realized that survival is not possible if it is not taken as an engineering discipline. **Engineering activities expect to produce proper documents after every stage of software development.** The development should be carried out as per well-documented standards and guidelines and should be completed within time and within budget. The same is expected in software development and all certifying agencies like SEI-CMM, ISO, etc. are directing us to document all processes and produce meaningful documentation after every stage of software development.

The term ‘software engineering’ was coined at the first conference on software engineering held in 1968 where Fritz et al. (1968) defined software engineering as:

The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines.

IEEE Computer Society’s software engineering body of knowledge defines software engineering as (Abren et al., 2004):

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, and the study of these approaches, that is, the application of engineering to software.

Wikipedia, the free encyclopedia has also defined software engineering as (Wikipedia 2010):

It is a profession dedicated to designing, implementing, and modifying software so that it is of higher quality, more affordable, maintainable, and faster to build.

All the above definitions are popular and find places in software engineering literature and practices. Our objective is to deliver good quality software to the customer and at a low cost. The software engineering discipline helps us to achieve this objective and also increases the probability of survival of the software during operation and maintenance.

1.1.1 Program vs. Software

Most of the times, we use these two terms, software and program interchangeably. Generally, we say “write a program” and “develop a software”. How are these terms different? Are these terms related? **Program is a set of instructions written for a specific purpose.** It may contain comment instructions (statements) to enhance the readability and understandability of a program.

However, **software is a combination of program(s), documentation (documents produced during development) and operating procedure manuals (delivered with programs to the customer at the time of release).** Hence, a software project of any reasonable size and functionalities should have associated documentations and operating procedure manuals. The various parts of software are shown in Figure 1.1.

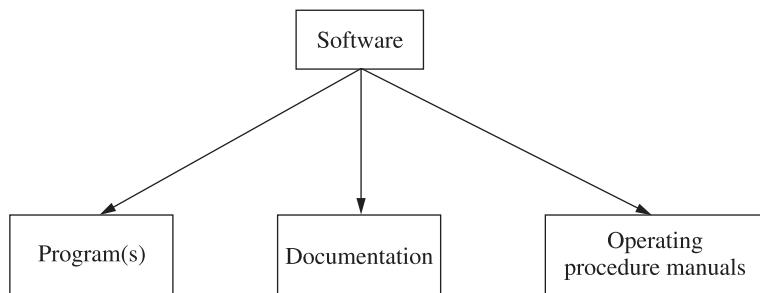


Figure 1.1 Parts of software.

We may produce many documents during software development. These documents are used not only during software development but also for maintaining the software. Some of the documents are given as:

- (i) Software requirements and specification document
- (ii) Software design document
- (iii) Test plan document
- (iv) Test suite document
- (v) Source code

Operating procedure manuals are prepared to help the customer to install, operate and maintain the software. They are delivered along with program(s) at the time of release. Some of the operating procedure manuals are given as:

- (i) Installation manual
- (ii) System administration manual
- (iii) Beginner's guide tutorial
- (iv) System overview
- (v) Reference guide

1.1.2 Characteristics of Software

The software is different from other products in terms of its special characteristic that it does not wear out. If it works correctly for a set of inputs in the specified environment, it is expected

to work in the same way after many years for the same set of inputs and environment. Some of the important characteristics of software are discussed here.

Software Does Not Wear Out

There is no concept of ‘aging’ in software. Hardware parts of any product (say automobile) start malfunctioning after the specified time due to aging of various parts. The same is true for every hardware product. If we draw a graph between failure rate and time, we get a curve which is similar to ‘bath tub’ and is known as **bath tub curve**. The curve has three phases, namely, **burn-in phase, useful life phase and wear-out phase** as shown in Figure 1.2.

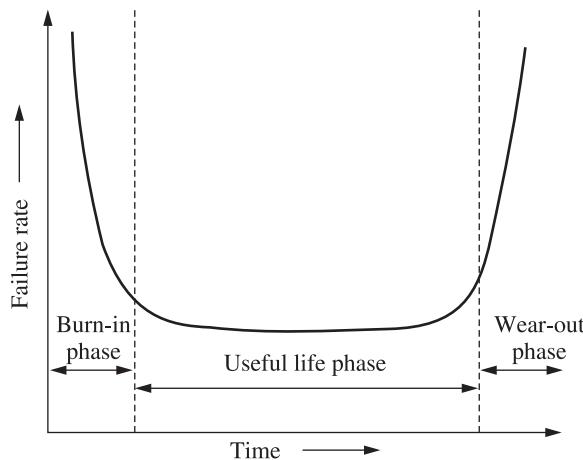


Figure 1.2 Bath tub curve.

In the initial phase (**burn-in phase**), the failure rate is high. When we start testing the product, we detect faults. These faults are corrected which results into a drop in the failure rate. However, the drop may stabilize after certain time of testing and the curve becomes stable. This is the end of initial phase (burn-in phase) which is expected to be performed in the company itself. Hence, after the completion, the product is **released** to the customer. The second phase where the failure rate is more or less constant is called useful life of any product. After the completion of the useful life phase, the failure rate again increases due to the effects of environmental conditions (like temperature, humidity, pressure, etc.), rains, dusts and rusting on various parts of product. This means, various parts begin to wear out. This phase is called **wear-out phase**.

We do not have the wear-out phase in software. The curve for software is shown in Figure 1.3.

As the testing progresses, many failures are experienced and the required corrective actions are performed to remove faults. Hence, the software becomes reliable over time instead of wearing out. It may become obsolete due to other reasons but not because of wear-out. Some of such reasons are **change in technologies, change in expectations, poor response time, deterioration in the structure of the source code, complexity issues, etc.**

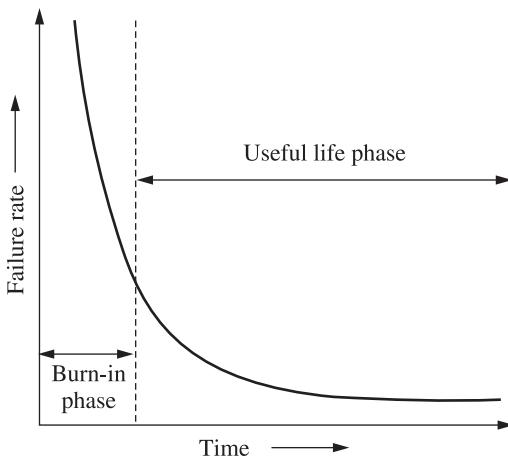


Figure 1.3 Software curve.

Flexibility of Software

Flexibility of software may create confusion in stakeholders. They may feel that this unique characteristic is a real strength of a software product in terms of making changes, incorporating new functionalities, modifying source code and managing complexity. This confusion has enhanced the expectation of stakeholders and propagated the theory of “any change possible at any time without much difficulty”. A stakeholder is a person who is having direct or indirect interest in the software system, including customers, software developers, managers, users, testers and so on. As we all know, it is easy to modify the source code but it is very difficult to manage the effect of changes in other parts of the source code. Addition of new functionality at later stages always makes the life difficult for software developers. Many times, developers may make compromises in terms of quality to accommodate the requests of changes. However, making changes in software is easier as compared to hardware and such changes must be made after proper study and analysis of its impact on other parts of the source code.

Reusability of Software

The concept of reusability of components is used in almost all disciplines of engineering. If we have to manufacture a computer under a brand name of our company, we may purchase monitor, keyboard, motherboard, microprocessor, switched-mode power supply (SMPS), etc. from the market. We may also manufacture a few components, say cabinet, mouse, UPS, few I/O cards, universal serial port, etc. in our company. Finally, we integrate all the components as per specified design and guidelines, and test the product as per the testing standards. The product is then released to the market under the brand name of the respective manufacturing companies. We do not manufacture every component in the company. We purchase those components from the market (like motherboard, UPS, SMPS, etc.) and reuse those components for manufacturing of computers. The reusable components are very common and popular in practice and also reduce the time and cost of manufacturing although maintaining the quality standards.

In software companies, most of the projects are developed for specific customers. Every customer has unique requirements and expectations. The concept of **reusability is rarely used and that has resulted into the development of costly software with prolonged development time.** Reusability is the process of integrating the already built well-tested components in the new software under development. By reusing the existing source code, the development effort and cost of development and testing are reduced. Some attempts have been made in terms of Microsoft Foundation Classes (MFC), Commercially off-the-shelf (COTS) components with a very limited applicability. More effort is required to standardize the design principles, guidelines and interfaces of COTS components. Reusability has introduced an emerging area of study and that is known as "**component-based-software-engineering (CBSE)**". Many popular COTS components use the concept of object-oriented analysis and design to make it an independent piece of software which can easily be plugged into other software to make an effective, reliable and efficient software for its customers. The object-oriented concepts may fulfil the long-standing desire of software engineering to produce libraries of standard software components which can easily be integrated in order to produce a good quality product in time and within budget. It is a very powerful and significant concept of object-oriented paradigm.

1.2 What is Object Orientation?

Why is object-oriented software development taking centre stage in software industry? Why is object-oriented version of existing software products coming in the market? Some of us also believe that conventional approaches of software engineering are old fashioned and shall not be able to meet today's challenges. We also feel that the real strength of the object-oriented approach is its **modelling ability to represent real-world situations.** A model helps us to visualize and understand a real situation along with its behaviour. Architects use models to demonstrate their conceptual constructs which may also increase the confidence of their clients in terms of design, aesthetics and feel of the proposed project. In object-oriented approaches, real-world situations are effectively represented at any level of abstraction with a few basic building blocks like objects, classes, messages, interfaces, inheritance and polymorphism.

1.2.1 Classes and Objects

Objects are the fundamental entities which are used to **model any system.** Anything and everything may become an object. It is like a packet containing data and operations. All objects have unique identification and are also distinguishable. Suppose there are 10 books in a library of the same subject, language, publisher and author; but these books are distinguishable due to their own title and accession number. All objects have unique identification like accession number in case of a book in the library. In a library, book, student, faculty, employee, etc. are the examples of objects. An object has attributes (information/state) and operations (behaviour). The objects of the same attributes and operations are combined to form a class. All objects are the instances of a class. **A class defines the structure of the objects which include attributes and operations.** An object is assigned to them and it behaves like a structure of C and record of Pascal programming language. A class 'book' is given in Figure 1.4 with eight attributes and four operations along with its implementation in C++ language.

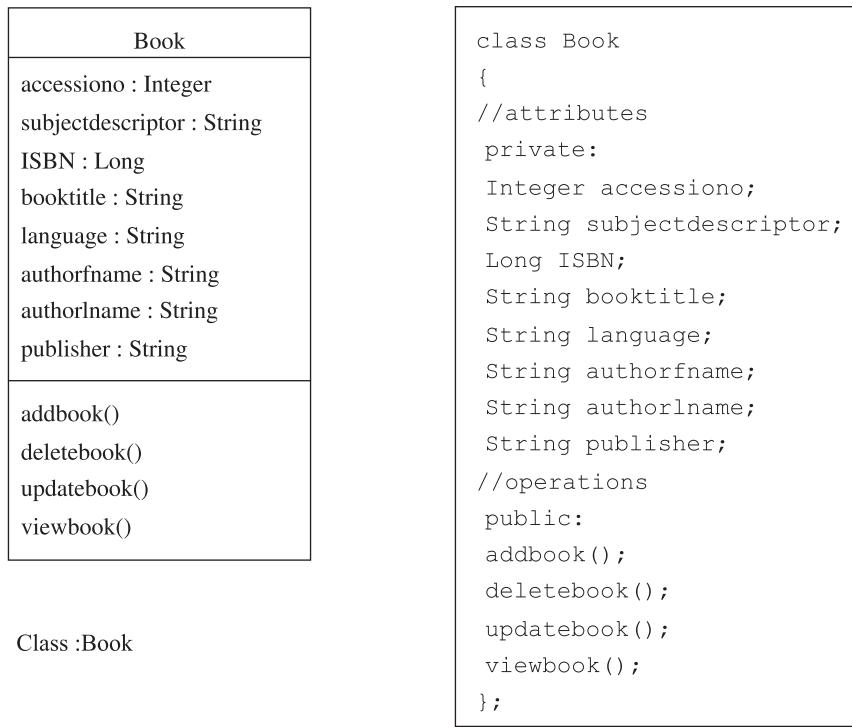


Figure 1.4 Class book with its C++ implementation.

A class represents a group of similar objects. There are three parts of a class as shown in Figure 1.4. The first part consists of the name of the class, the second part consists of all the attributes (or information) and the third part has all the operations (or behaviour). The source code of book class is also shown in Figure 1.4. The access specifiers (private and public) are used to specify the visibility of data and operations to the other objects. The private data/operations will not be available to the outside world. Only the operations of the class can access this data, whereas public data/operations are available to other classes. In class book, the data is declared as private and the operations are declared as public. This supports the concept of data hiding. The book class provides controlled access to its data attributes. In C++, three access specifiers—private, public and protected—are used. If no access specifier is specified, then by default all the members of the class are private.

A class has a name, list of attributes and operations. Jacobson et al. (1999) have defined a class as:

A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structures.

In the object-oriented system, every object belongs to a class and is known as *instance* of that class. An object is also defined by Jacobson et al. (1999) as:

An instance is an object created from a class. The class describes the (**behaviour and information structure**) of the instance, while the current state of the instance is defined by the operations performed on the instance.

For example, a book class may have many objects like software engineering book, software testing book, software quality book, etc. as shown in Figure 1.5. As shown in C++ implementation in Figure 1.5, the objects can be created in the same way as the variables are declared. For example, the variables are declared as:

```
int x;  
char str;
```

The objects can be declared as:

```
Book SE;
```

Each object has its own copy of attributes and operations as shown in Figure 1.5.

1.2.2 Messages, Attributes and Methods

In order to show the dynamic behaviour of any system, the **objects are required to communicate with each other by passing messages**. A message is a request for performing an operation by any object of the system. A message may consist of the identification of the target object, operation to be performed and data to perform the request as shown in Figure 1.6. An object which generates the message is called the *sender* and the object which receives a message can be sent to other objects and to the object itself to perform an operation.

An attribute is the data of the class. Each class should define the attributes that store the state of an object. For example, in the previous section, the book class contains eight attributes accessiono, subjectdescriptor, ISBN, booktitle, language, authorfname, authorlname and publisher. All these attributes are declared as private, which implies that they cannot be accessed outside the class and are only available to the operations of the class book.

A method is the sequence of steps (or a set of operations) to be performed to complete the designated work. There may be many methods available for the accomplishment of any work. It is the responsibility of the receiver to select an appropriate method to fulfil the work efficiently. In Figure 1.5, four methods addbook(), deletebook(), updatebook() and viewbook() are available for the book class. In order to add a book in the database, addbook() method should be invoked. The words methods, functions and operations are used interchangeably in the object-oriented systems.

1.2.3 Encapsulation

Encapsulation is also known as *information hiding concept*. The data and operations are combined into a single unit. The only way to access data is through operations which are designed to operate on the data. The data is not available to the external world. This concept may make the data safe and secure from external interventions. Accidental modifications from the outside world are not possible as shown in Figure 1.7. If the data of any object is required to be modified, it is possible through the specified functions. This process of encapsulating the

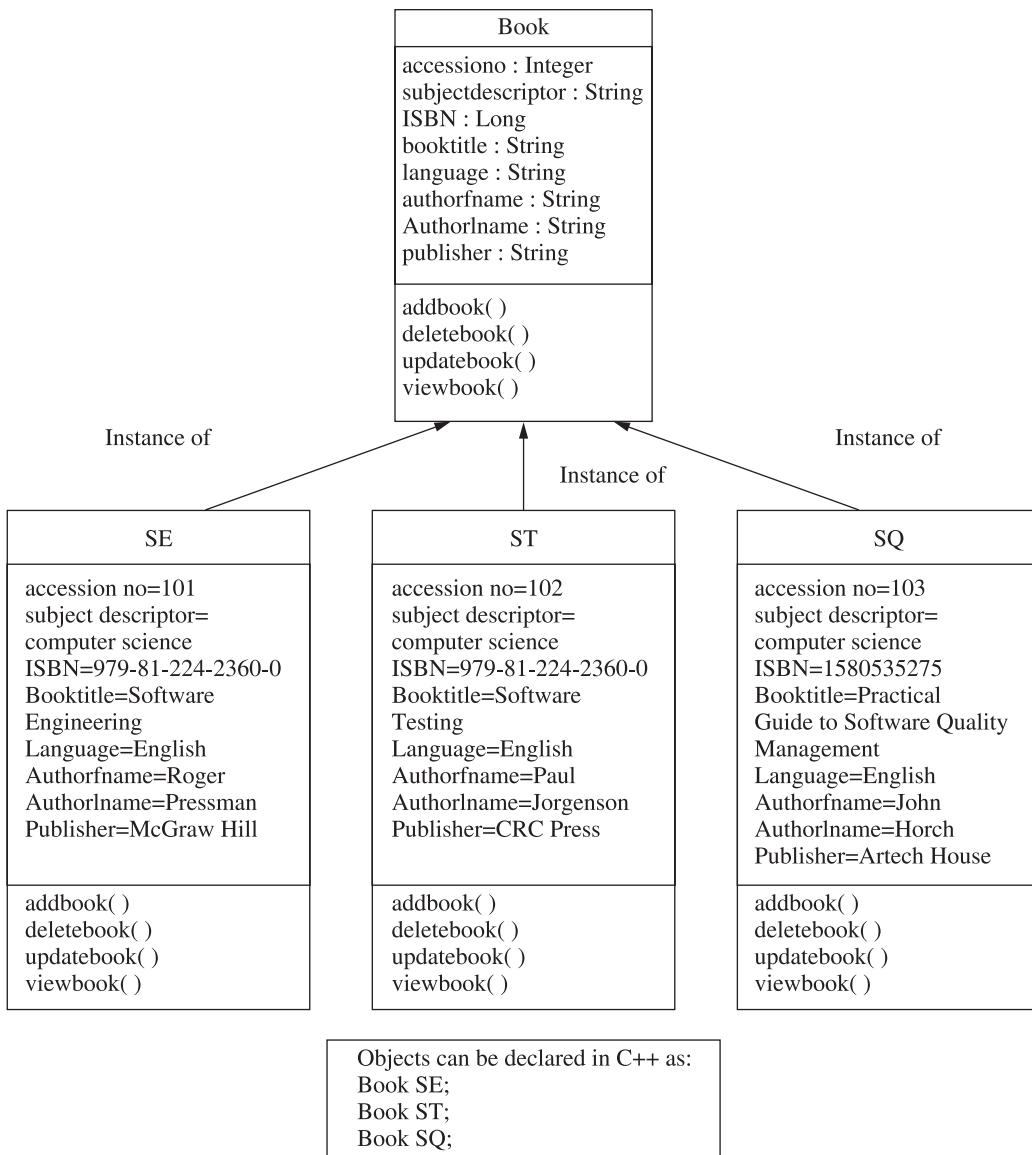


Figure 1.5 Instances of class book.

data and operations into a single unit simplifies the activities of writing, modifying, debugging and maintaining the program.

The availability of data only for specified operations limits the scope of data and this may help us to minimize the impact of any change in the program. In the class book of library management system, all the attributes are private and only the operations can access the attributes (see Figure 1.4). The private attributes are not accessible to the outside world. Thus, the private

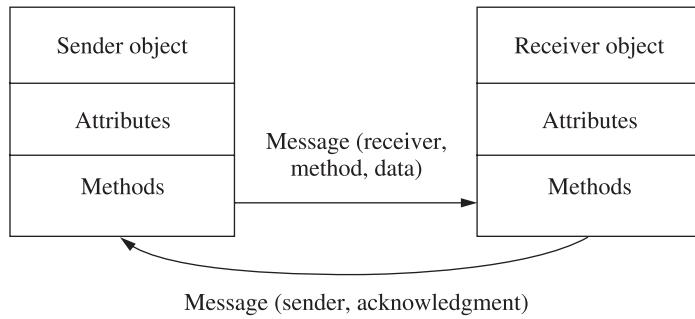


Figure 1.6 Interactions of objects.

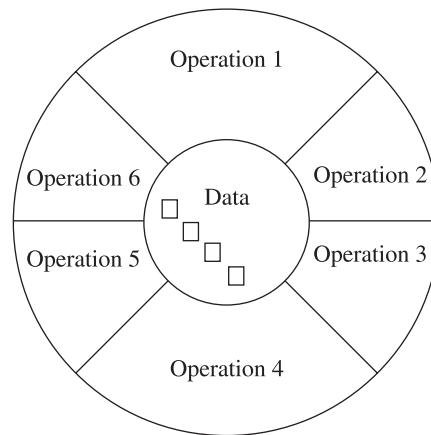


Figure 1.7 Data and operations are combined in a unit.

part in the implementation is hidden from the users. The outside world can only interact with the class book through public interfaces, i.e. the operations. If we need to interact with the operation addbook() the following syntax is used:

Book b1;
b1. addbook();

In Wikipedia, encapsulation is defined as:

It means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its field.

The encapsulation concept enhances the security and makes the program more maintainable and sustainable.

1.2.4 Inheritance

Inheritance is a very important feature of object-oriented systems. When we define a class, we may find common attributes and operations amongst the classes. In a library management system, we may define classes like student, faculty and employee. All these classes have many common attributes (say memberID, name, date of birth, phone, email, etc.) and operations (say addmember(), deletemember()). The common characteristics can be extracted and a new class “member” may be created with common attributes and operations. The member class contains attributes and operations which are common to all three classes, namely, student, faculty and employee as shown in Figure 1.8.

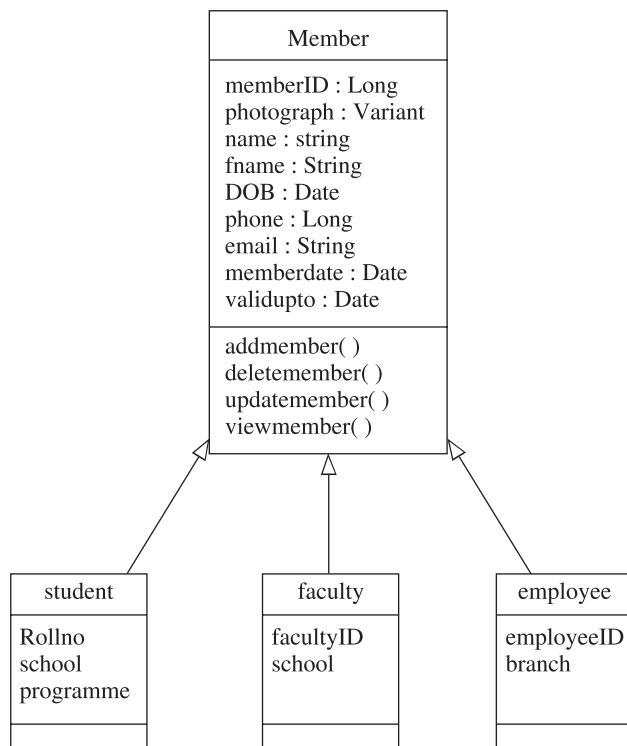


Figure 1.8 Member class with three subclasses student, faculty and employee.

The member class is called the base class and other classes are called subclasses. The subclasses inherit the attributes and operations of the base class and have their own behaviour. Every subclass may have specific attributes (like rollno and school in the case of student base class) and operations which are not available to other subclasses of the same base class. This type of specialization-generalization relationship is known as ‘is-a’ relationship. The C++ implementation is given in Figure 1.9.

```
class Member
{
private:
    long int memberID;
    char photograph [300];
    char name[50];
    char fname [50];
    Date DOB;
    long int phone;
    char email [60];
    Date memberdate;
    Date validupto;

public:
    addmember();
    deletemember();
    updatemember();
    viewmember();
};

class employee : public Member //publicly derived class
{
    long int employeeID;
    char branch[60];
};

class faculty : public Member //public inheritance
{
    long int facultyID;
    char school[100];
};

class student : public Member //public inheritance
{
    long int rollno;
    char school[100];
    char programme[100];
};
```

Figure 1.9 C++ implementation of class member with three subclasses.

All the attributes and operations of a base class become the part of its subclasses and we may say that a subclass inherits the base class. If any change is made in the base class, that will be equally applicable to all its subclasses. Inheritance property also helps to remove the redundant information from the classes. This may reduce the size of the classes and they may become easier to understand. In Figure 1.8, the class student inherits the class member and the class student is called the derived class and class member is called the base class. Inheritance allows the developer to reuse a class and also modify the classes as per requirements with minimizing their side effects into the rest of the class.

1.2.5 Polymorphism

The dictionary meaning of polymorphism is “many forms”. If we give **an instruction to many objects across classes, the objects’ responses may be different**. For example, human, animal, bird, etc. are subclasses of mammal class. When we give an instruction “come fast” to all mammals, their responses will be different. Animals may run, birds may fly and humans may walk or take an automobile in response to the given instruction. This concept is called *polymorphism*, where the **same instruction is given to many objects across the classes, but responses of objects may be different**. Objects interact with each other by sending messages. A sending object may not know the class of a receiving object. Jacobson et al. (1999) have rightly said that:

Polymorphism means that the sender of a stimulus (message) does not need to know the receiving instance’s class. The receiving instance can belong to an arbitrary class.

Polymorphism can be achieved through **function overriding**. For example, consider the media class with two operations read and show. The publication media need to read and display different attributes such as the number of pages and publisher, whereas videotapes also need to read and display different attributes such as time. Thus, the class publication and the class videotapes override the operations read and show of the base class media and provide their own read and show operations. Function overriding involves replacement of the same operation in the base class with the one in the derived class. An object of the derived class refers to its own function if the same function is defined in its base class. The example is shown in Figure 1.10 and its C++ implementation is given in Figure 1.11.

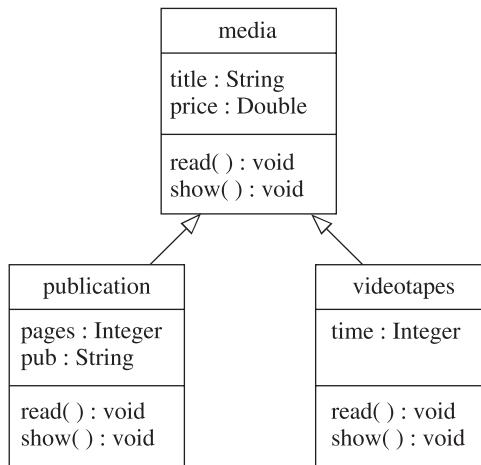


Figure 1.10 Example of function overriding.

The read and show operations in media class are overridden by the operations in publication and videotapes derived class. The read and show operations in the publication class read and display their own specific attribute pages and pub in addition to the attributes in the base class. Similarly, read and show operations in the videotapes class read and display their own specific attribute time in addition to the attributes in the base class media. Thus, the same message is sent to the media classes, and their behaviour is different depending upon which subclass of media is used.

```
class media
{
private:
    char title[MAX];
    float price;
public:
void read()
{
cout<<"\n\tEnter title:"; 
cin>>title;
cout<<"\n\n\tEnter price:"; 
cin>>price;
}
virtual void show()
{
cout<<"\n\n\tTitle: "<<title;
cout<<"\n\n\tPrice: "<<price;
}
};
class publication : public media //Class to display information about books
{
private:
int pages;
char pub[MAX];
public:
void read()
{
media::read();
cout<<"\n\n\tEnter publication: ";
cin>>pub;
cout<<"\n\n\tEnter number of pages:"; 
cin>>pages;
getch();
}
void show()
{
media::show();
cout<<"\n\n\tPublication: "<<pub;
cout<<"\n\n\tPages:"<<pages;
}
};
class videotapes:public media //Class to display information about videotapes
{
private:
int time;
public:
void read()
{
media::read();
cout<<"\n\n\tEnter time in seconds";
cin>>time;
}
void show()
{
media::show();
cout<<"\n\n\tTime: "<<time;
}
};
```

Figure 1.11 C++ implementation of classes in Figure 1.10.

When the object of the derived classes publication and videotapes will be created, the read and show methods included in the derived classes will be called instead of the read and show methods included in the base class media shown as follows:

```
publication p1;
p1.read(); //read method defined in publication class will be called instead of read
//method defined in media class
```

The concept of polymorphism is frequently used in object-oriented programming languages like C++ and Java. An operator ‘+’ is used for the purpose of addition and in Boolean, the same operator is used for logical ‘OR’ operation. Hence, the same operator has two (many) forms. We also know that arithmetic operators like +, -, *, etc. are used on primary data types (int float) and the same may be used in user-defined data types (objects) and is called operator overloading. An operator may have many forms depending on its usage in the programming language.

1.2.6 Responsibility and Abstraction

Every object has responsibilities which are associated with its behaviour. In order to execute a request, a receiver object decides a method out of a set of available methods. The selection of an appropriate method is the responsibility of the receiver object. Hence, objects behave and work which is a very important characteristic for solving any complex problem. The complexity may also be managed using right level of abstraction. An abstraction is the elimination of irrelevant and the amplification of the essential details. We learn driving a car without knowing the internal details of the engine, batteries, control system, etc. The details may not be required to know for a learner and may create unnecessary confusion. Hence, abstraction may improve the understanding of the system. All complex details are not relevant for everyone. Levels of abstraction may vary in different situations. We only show the necessary details which may improve the understandability. We perform most of our works by knowing only the essential details. We work on computers without knowing the architecture of motherboard and other peripheral devices. These details are not required for any normal user. If we have to improve the design of the motherboard, obviously, details of architectural design are important but not otherwise. Such details are required only by the motherboard designers and not by the users. Hence, the right level of abstraction is the key to success in all situations.

1.2.7 Object Composition

The use of objects as data members in another class is referred to as object composition. The object is a collection of a set of objects represented through ‘has-a’ relationship. In object-oriented systems, has-a relationship depicts that an object is declared as an attribute in another class. For example, a CPU has-a motherboard, memory chip, CD-ROM and so on. In inheritance, there is ‘is-a’ relationship between the base class and the derived class. In Figure 1.12, an example of object composition between two classes along with their C++ implementation is shown. Class B has-a member Oa of type class A. Thus, the object of type class A is used as data member in class B.

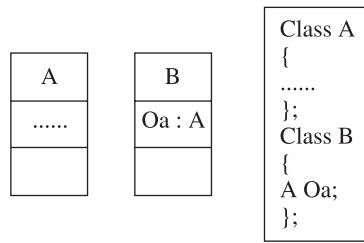


Figure 1.12 Object composition.

1.3 Object-Oriented Methodologies

Methodology is a set of methods and principles that signify a systematic way of developing a system using tools, techniques, processes and procedures. As a result of increase in keen interest in object-oriented techniques, several object-oriented methodologies have been developed in the literature. These methodologies include:

- Object-oriented analysis by Coad and Yourdon
- Object-oriented design by Booch
- Object modelling technique by Rumbaugh et al.
- Object-oriented software engineering by Jacobson et al.

Each methodology is discussed in the next subsections.

1.3.1 Coad and Yourdon Methodology

Coad and Yourdon's methodology is popularly known as *Object-Oriented Analysis (OOA)* (Coad and Yourdon, 1990). It was developed in the late 1980s. In OOA, an analysis model is developed and it consists of five steps:

1. Identification of classes and objects
2. Identification of structures
3. Definition of subjects
4. Definition of attributes
5. Definition of services (methods)

Identification of classes and objects involves investigating the application domain and the system's environment. The behaviour of each object is found and this information is documented. Identification of structures involves identification of is-a and whole-part relationships. The is-a relationship captures class inheritance (known as Gen-Spec structure) and the whole-part relationship captures the information about how an object is a part of another object. Each structure is classified into a subject. Attributes are the data members of the class. The attributes for each object are defined and kept at the appropriate level in the inheritance hierarchy. Defining services involves identification of operations in a class. This also involves identification of interfaces amongst the objects through messages. Each message may contain parameters for communication. The graphical notation of the analysis model is shown in Figure 1.13.

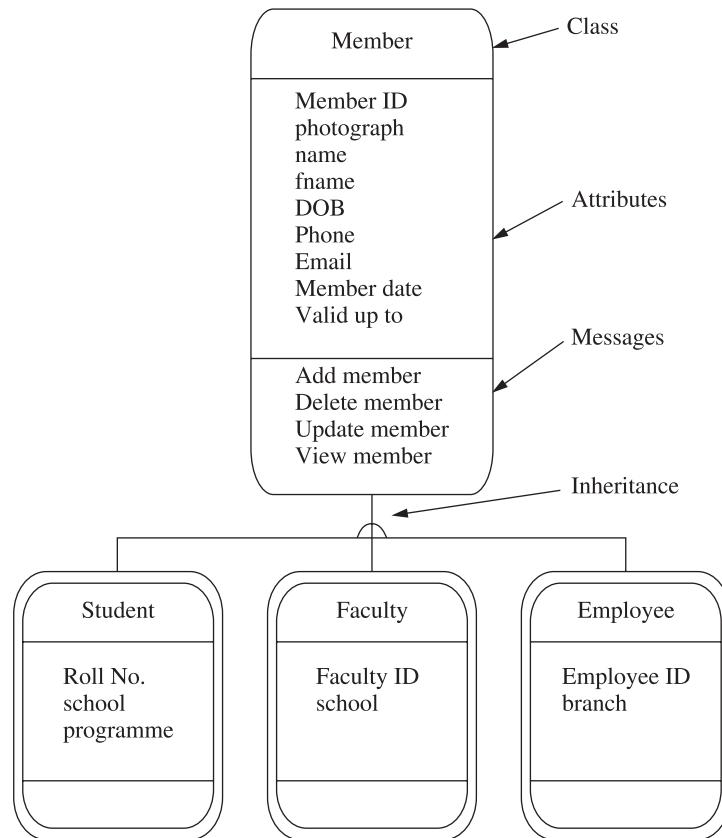


Figure 1.13 Coad and Yourdon's methodology.

In the OOA methodology, there is no systematic way to identify classes and objects. The interfaces are also not determined in this methodology.

1.3.2 Booch Methodology

Grady Booch proposed object-oriented methodology in his book *Object-Oriented Design (OOD)* in 1991 (Booch, 1991). The primary aim of OOD was to establish a base for the implementation of object-oriented systems. The Booch methodology considered **identification of interfaces at various levels**. The main concern of the Booch methodology is the iterative process and creativity of designer in order to develop OOD. The Booch methodology follows and **incremental and iterative life cycle**. It combines analysis, design and implementation and provides a sound object-oriented software engineering process for analysts, designers and developers. The Booch methodology can be broadly divided into two processes: macro process and micro process.

Macro Process

It is a high-level process that **describes the activities** to be followed by the development team throughout the life cycle as shown in Figure 1.14.

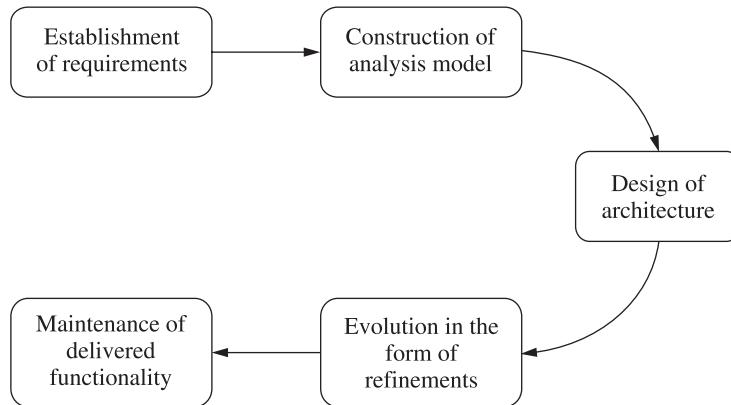


Figure 1.14 Macro process of Booch methodology.

In the first phase, the requirements are established using context diagrams and prototypes. The outcomes of this phase are core requirements of the system. The analysis process involves requirement capturing and understanding. It involves “what of the system”. This phase consists of construction of use cases, identification and prioritization of risks. The design phase focuses on the construction of architecture of the system and involves:

- Identifying horizontal layers
- Mapping classes to subsystems
- Release planning
- Attaching risks identified in the analysis phase to releases

The evolutionary phase involves implementation of the system and each release adds to the functionality of the system. The maintenance phase consists of post-deployment activities.

Micro Process

It is the lower level process. The following recursive steps are followed in OOD micro process:

1. Identification of classes and objects
2. Identification of semantics of classes and objects
3. Identification of relationship amongst classes and objects
4. Specification of interfaces and implementation of classes and objects

Identification of classes and objects involves finding of classes and objects at specific level of abstraction. The second step identifies the semantics of the classes and objects by finding the meaning of the classes and objects in terms of attributes and operations. It also involves determining the use of an object by another object. Identification of relationship involves the process of determining the interaction amongst the classes and objects. The relationships identified include inheritance, uses and instance of. The last step involves specification of interface amongst classes and objects and their implementation. It identifies data type of attributes and signature of operations.

In this methodology, no formal notations for constructing visual diagrams are given.

1.3.3 Rumbaugh Methodology

Rumbaugh developed a technique that focuses on analysis, design and implementation of the system. This technique is popularly known as *Object Modelling Technique (OMT)* (Rumbaugh et al., 1990). The OMT consists of four phases: analysis, system design, object design and implementation.

- **Analysis phase:** Analysis phase is composed of three submodels given below:
 1. **Object model:** It captures the static aspect of the system.
 2. **Dynamic model:** It captures the behavioural aspects of the object models and describes state of the objects.
 3. **Functional model:** It represents the functional aspects of the system in terms of operations defined in the classes.
- In the object model, the requirements are stated in the problem statement. The relevant classes along with inheritance relationships are extracted from this problem statement. The dynamic model identifies states and events in the classes identified in object models. The functional model depicts the functionality of the system by creating data flow diagrams of the system in order to understand the processes (in the form of input and output information).
- **System design phase:** In this phase, a high-level design is developed taking the implementation environment including DBMS and communication protocols into account.
- **Object design phase:** The goal of this phase is to define the objects in detail. The algorithms and operations of the objects are defined in this phase. New objects may be identified to represent the intermediate functionality.
- **Implementation phase:** Finally, the objects are implemented following coding standards and guidelines.

1.3.4 Jacobson Methodology

All the methodologies described above still lack a comprehensive architecture to develop a software project. The Jacobson's methodology known as *Object-Oriented Software Engineering (OOSE)* consists of five models (Jacobson et al., 1999):

1. **The requirement model:** The aim of this model is to gather software requirements.
2. **The analysis model:** The goal of this model is to produce ideal, robust and modifiable structure of an object.
3. **The design model:** It refines the objects keeping the implementation environment in mind.
4. **The implementation model:** It implements the objects.
5. **The test model:** The goal of the test model is to validate and verify the functionality of the system.

The requirement model focuses on capturing functionality of the system through **use cases and actors** and determining the **interfaces amongst the use cases**. In the analysis model, an ideal model is developed while not considering the implementation environment. It involves the identification of interface objects, **database-related objects (entity objects)** and objects representing control between interface and **database-related objects (control objects)**. The objects are grouped into subsystems. In the design model the objects identified in the analysis model, are refined to the actual implementation environment. These objects are called *blocks*. These blocks are converted into modules in the implementation model and finally the modules are integrated and tested in the test model.

All the methods discussed above have their own advantages and disadvantages. The OOA method lacks the design of interfaces and notations. The OOD method is stronger in design but weaker in analysis of the system. The OMT method is stronger in analysis part but weaker in design part. The Jacobson's methodology is stronger in behavioural areas as compared to the other areas (Quatrani, 2003).

1.4 Object-Oriented Modelling

Object-oriented modelling is a way of constructing visual models based on real-world objects. Modelling helps in understanding the problems, developing proper documents and producing well-designed programs. **Modelling produces well-understood requirements, robust designs, and high-quality maintainable systems**. The models must depict the various views of the system and these models must be verified to check whether they capture the customer's requirements. After they represent the required details, these models may be transformed into source code. Due to the increase in object-oriented techniques and complexity of the software, various challenges are faced by the system designers. Modelling helps to visualize, organize, document and understand complex relationships, and to produce well-designed systems.

As discussed in the previous section, three most popular methodologies were OOD (Booch), OMT (Rumbaugh) and OOSE (Jacobson). All these methods are combined into the Unified Modelling Language (UML). Thus, the UML represents the combination of the notations used by Booch, Rumbaugh and Jacobson. The best concepts and processes were extracted from all the methodologies till date and combined into UML. The UML was adopted by Object Management Group (OMG) in November 1997.

The UML is defined as a language for visual modelling that allows to specify, visualize, construct, understand and document the various artifacts of the system (Rumbaugh et al. 2004). UML models the static and dynamic aspects of the system. The static aspect of the system models the objects and relationship amongst these objects. The dynamic aspect of the system models the events and states used for interaction amongst objects over time. Thus, modelling the system from different aspects allows us to obtain a clear understanding of the system.

1.5 Some Terminologies

Some terminologies which are frequently used in object-oriented software engineering are discussed in this section.

1.5.1 Customers, Developers and Users

Customers are persons who request the system, approve the system and pay for the system. Developers are the persons at the supplier side who are responsible for the development of the system. Users are the persons who actually use the system.

For example, in the library management system developed for a university, the customer is the university, the developer is the one at the supplier side who develops the system and the users are the persons in the library staff who will actually work on the system.

1.5.2 Product and Process

Product is what is delivered to the customer. It may include software requirement specification (SRS) document, source code, test reports, user manuals and system guide.

Process is the way in which the software is produced. A process is like a tunnel through which the project goes in order to produce a product as shown in Figure 1.15. It consists of a set of activities that are combined to produce a product. An effective process is required to produce a high-quality maintainable product.

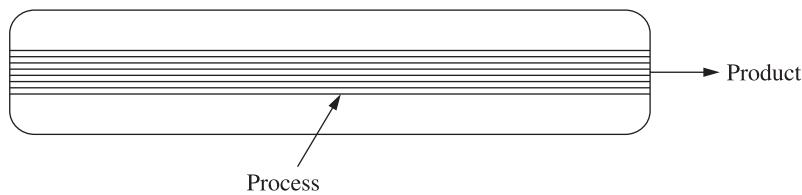


Figure 1.15 Process and product.

1.5.3 Actor, Use Case, Use Case Model and Use Case Scenario

An actor represents the role of a user that interacts with the system. Some of the examples of the actors used in “Library Management System” are administrator, data entry operator, student, library staff and faculty.

A use case describes who (any user) does what (interaction) with the system, for what goal, without considering the internal details of the system. A complete set of use cases explains the various ways to use the system. The use case model depicts actors, use cases and the relationship between them.

A use case scenario is an instance of a use case or a complete path through the use case. The basic flow is one scenario and every alternative path gives another scenario.

1.5.4 System and Subsystems

A system is an organized and arranged structure as a whole that consists of interrelated and well-defined procedures, processes and methods. All systems consist of inputs, outputs, feedback mechanisms and boundaries.

The system may consist of several subsystems. Subsystems are a way of reducing complexity of the system. For example, in a company, accounts, sales, marketing, etc. are different subsystems. In object-oriented analysis, objects may be grouped together to form a subsystem.

1.5.5 Class, Responsibility and Collaboration

A class is a template that consists of attributes and operations. Responsibilities are attributes and operations included in a class. Collaborations are the other classes that a class calls in order to achieve the functionality.

The class, responsibility and collaboration are often combined together in object-oriented analysis to depict the functionality of a class and the relationship between classes.

1.5.6 Measures, Metrics and Measurement

These terms are often used interchangeably. However, we should understand the difference amongst these terms. Pressman (2005) explained this clearly as:

A measure provides a quantitative indication of the extent, amount, dimension, capacity or size of some attributes of a product or process. Measurement is the act of determining a measure. The metric is a quantitative measure of the degree to which a product or process possesses a given attribute.

For example, a measure is the number of failures experienced during testing. Measurement is the way of recording such failures. A software metric may be an average number of failures experienced per hour during testing.

Fenton and Pfleeger (2004) have defined measurement as:

It is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.

Software metrics can be defined by Goodman (1993) as:

The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.

1.5.7 Software Quality and Reliability

Software **reliability** is one of the important factors of software **quality**. Other factors are **understandability, completeness, portability, consistency, maintainability, usability, efficiency**, etc. These quality factors are known as non-functional requirements for a software.

Software reliability is defined as “the probability of failure free operation for a specified time in a specified environment” (ANSI, 1991). Although software reliability is defined as a probabilistic function and comes with the notion of time, still it is not a direct function of time. The software does not wear out like the hardware during the software development life cycle. There is no aging concept in software and it will change only when we intentionally change or upgrade the software.

Software quality determines how good the software designed is (quality of design), and how good the software conforms to that design (quality of conformance).

Some software practitioners also feel that quality and reliability are the same thing. If we are testing a program till it is stable, reliable and dependable, we are assuring a high-quality

product. Unfortunately, that is not necessarily true. Reliability is just one part of quality. To produce a good quality product, a software tester must verify and validate throughout the software development process.

1.5.8 Quality Assurance and Quality Control

The purpose of **quality assurance (QA)** activity is to **enforce standards and techniques to improve the development process and prevent the previous bugs from ever occurring**. A good QA activity enforces good software engineering practices which help to produce good quality software. The **QA group monitors and guides throughout the software development life cycle**. This is a defect-prevention technique and concentrates on the process of the software development. Examples are **reviews, audits, etc.**

Quality control attempts to build a software and test it thoroughly. If failures are experienced, remove the causes of failures and ensure the correctness of removal. It concentrates on specific products rather than processes as in the case of QA. This is a defect-detection and correction activity which is usually done after the completion of the software development. Example is software testing at various levels.

1.5.9 Verification and Validation

These terms are used interchangeably and some of us may also feel that both are synonyms. The Institute of Electrical and Electronics Engineers (IEEE) has given definitions which are largely accepted by software testing community. **Verification is a static activity often related to static testing which is performed manually. We only inspect and review the document. However, validation is a dynamic activity and requires the execution of the program.**

Verification: As per IEEE (2001), “**It is the process of evaluating the system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase**”. We apply verification activities from the early phases of the software development and check/review the documents generated after the completion of each phase. Hence, it is the process of reviewing the requirement document, design document, source code and other related documents of the project. **This is a manual testing and involves only looking at the documents in order to ensure what comes out is that we expected to get.**

Validation: As per IEEE (2001), “**It is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements**”. **It requires the actual execution of the program.** It is a dynamic testing and requires computer for execution of the program. Here, we experience failures and identify the causes of such failures.

Hence, testing includes both verification and validation. Thus,

$$\text{Testing} = \text{Verification} + \text{Validation}$$

Both are essential and complementary activities of software testing. If an effective verification is carried out, it may minimize the need of validation and more number of errors may be detected in the earlier phases of the software development. Unfortunately, primarily the validation-oriented activity is followed while testing a software product.

1.5.10 Fault, Error, Bug and Failure

All these terms are used interchangeably, although the terms error, mistake and defect are synonyms in software testing terminology. When we make an error during coding, we call this 'bug'. Hence, error/mistake/defect in coding is called a bug.

A fault is the representation of an error, where the representation is the mode of expression, such as data flow diagrams, entity-relationship (ER) diagrams, source code, use cases, etc. If the fault is in the source code, we call it a bug.

A failure is the result of execution of a fault and is dynamic in nature. When the expected output does not match with the observed output, we experience a failure. The program has to execute for a failure to occur. A fault may lead to many failures. A particular fault may cause different failures, depending on the inputs to the program.

1.5.11 States and Events

A state is an abstract situation in the life cycle of an entity that occurs in response to occurrence of some event. An event is an input (a message or method call). Due to the occurrence of some event, the system transits from one state to the other.

1.5.12 Traditional Approach and Object-Oriented Approach

A conventional approach is a function-centric approach where data is not given importance, whereas an object-oriented approach is a data-centric approach where data and operations are grouped into a single unit. In today's world, the paradigm is being shifted towards object-oriented systems. The summary of differences between a traditional approach and an object-oriented approach is given in Table 1.1.

Table 1.1 Comparison between traditional and object-oriented approaches

S. No.	Traditional approach	Object-oriented approach
1	The system is viewed as a collection of processes.	The system is viewed as a collection of objects.
2	Data flow diagrams, ER diagrams, data dictionary and structured charts are used to describe the system.	UML models including use case diagram, class diagram, sequence diagrams, component diagrams, etc. are used to describe the system.
3	Reusable source code may not be produced.	The aim is to produce reusable source code.
4	Data flow diagrams depicts the processes and attributes.	Classes are used to describe attributes and functions that operate on these attributes.
5	It follows a top-down approach for modelling the system.	It follows a bottom-up approach for modelling the system.
6	It is non-iterative.	It is highly iterative.

Object-oriented software engineering is an upcoming area of research, practice and industrial applications. All companies are making the processes compliant to object-oriented paradigm. Developers are focusing these learning processes on object-oriented concepts and programming languages like C++, Java, etc. Customers are also changing their mindsets towards object-oriented software products.

Review Questions

1. What is software engineering? What is the aim of software engineering?
2. Describe the characteristics of software. How are the characteristics of software different from those of hardware?
3. Explain the difference between program and software. What is the importance of producing documents and operating manuals in software development?
4. Software does not wear out. Comment.
5. Give the characteristics of an object-oriented system. What are the main advantages of object-oriented software development?
6. Describe attributes and operations with respect to an object with relevant example.
7. (a) What is object orientation? How is it close to real-life situations? Explain the basic concepts which help us to model a real-life situation.
(b) Describe the following terms:
 - (i) Messages
 - (ii) Methods
 - (iii) Responsibility
 - (iv) Abstraction
8. (a) How is object-oriented software development different from traditional software development?
(b) Discuss the following terms:
 - (i) Object
 - (ii) Class
 - (iii) Message
 - (iv) Inheritance
 - (v) Polymorphism
9. What is modelling? Explain how modelling can play an important role in designing a system.
10. Explain the requirement and analysis model in the Jacobson methodology.
11. Compare and contrast the object-oriented methodology of Booch, Rumbaugh and Jacobson.
12. Explain the macro and micro process of the Booch methodology.
13. Mention the models in the analysis phase of the Rumbaugh methodology and explain their roles for describing the system.
14. Explain the following:
 - (a) Object modelling technique
 - (b) Object composition with an example
15. Give a brief description of the characteristics of object-oriented modelling.
16. Explain object model, dynamic model and functional model.

17. What is UML? Explain the objectives of modelling.
18. Differentiate between the following:
 - (a) Measure and metrics
 - (b) Program and software
 - (c) Verification and validation
 - (d) Object-oriented and structural approach
 - (e) Class and object
 - (f) Customer and user
 - (g) System and subsystem
 - (h) Fault, bug and failure
19. Define the following terms with examples:
 - (a) Class and object
 - (b) Polymorphism
 - (c) Object composition
 - (d) Inheritance
 - (e) Encapsulation
 - (f) Customer and user
20. Consider a result management system of a university. Identify the objects, classes, attributes and methods. Give C++ code for all of them.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. Software engineering focuses on producing:

(a) Good quality product	(b) Defect-free product
(c) High performance product	(d) Reusable product
2. Software consists of:

(a) Programs	(b) Programs and documentations
(c) Set of instructions and operating system	(d) Programs, documentations and operating procedure manuals
3. Operating manuals help to:

(a) Understand the software	(b) Operate the software
(c) Maintain the software	(d) All of the above
4. Which is **not** a part of operating procedure manuals?

(a) Installation guide	(b) Reference guide
(c) Test plan	(d) System administration manual
5. Which of the following is **not** a part of documentations?

(a) Software requirement specification document	(b) System overview
(c) Test plan	(d) Source code

-
6. Which one is **not** a characteristic of software?
- (a) Software does not wear out (b) Software is not flexible
(c) Software is enhanceable (d) Software is reusable
7. The full form of COTS is:
- (a) Commercially off-the-shelf components
(b) Commercially off-the-shelf classes
(c) Components off-the-shelf
(d) Commercially off-the-shelf components
8. Stakeholders consist of:
- (a) Developers (b) Management
(c) Users (d) All of the above
9. The use of components promotes the concept of:
- (a) Flexibility (b) Reusability
(c) Invisibility (d) Conformity
10. A class has:
- (a) Attributes and operations (b) Attributes and states
(c) Operations and behaviour (d) State and information
11. An object is defined as:
- (a) Information of a class (b) Instance of a class
(c) Attribute of a class (d) Operation of a class
12. The objects of the same class have:
- (a) Different definitions for operations and information
(b) Same definition for operations and information
(c) Different operations
(d) Different formats
13. A class inherits information from
- (a) Descendant classes (b) Same class
(c) Ancestor classes (d) Descendant and ancestor classes
14. Encapsulation is known as:
- (a) Data sharing concept (b) Data retrieval concept
(c) Data hiding concept (d) Data transfer concept
15. A method is:
- (a) The sequence of steps to be performed to fulfil the assigned task
(b) The set of operations for a particular task
(c) Both (a) and (b)
(d) None of the above
16. The literal meaning of polymorphism is:
- (a) Few forms (b) Many forms
(c) No form (d) Different things with the same meaning

- 17.** Function overriding is a type of:

 - (a) Encapsulation
 - (b) Inheritance
 - (c) Polymorphism
 - (d) Reusability

18. Abstraction is:

 - (a) Elimination of relevant details
 - (b) Elimination of irrelevant details
 - (c) Getting distracted by thoughts
 - (d) Reducing the understanding of the system

19. Object composition refers to:

 - (a) Use of object of one class as data type in another class
 - (b) Derived class inheriting attributes and operations from the base class
 - (c) Polymorphism
 - (d) Data hiding concept

20. Object-oriented methodologies include:

 - (a) Coad and Yourdon methodology
 - (b) Booch methodology
 - (c) Rumbaugh methodology
 - (d) All of the above

21. Object-oriented methodologies do **not** include:

 - (a) Coad and Yourdon methodology
 - (b) Jacobson methodology
 - (c) Boehm methodology
 - (d) Rumbaugh methodology

22. A class inherits data and operations from its:

 - (a) Subclass
 - (b) Superclass
 - (c) Derived class
 - (d) All of the above

23. What is encapsulation?

 - (a) Enforcement of data hiding concept
 - (b) Division of a module into submodules
 - (c) Including data members within a class
 - (d) Including operations within a class

24. What is the importance of data hiding?

 - (a) Preventing data from intentional modifications
 - (b) Making data available in correct format
 - (c) Preventing data from accidental modifications
 - (d) None of the above

25. What are the benefits of inheritance?

 - (a) Lowers the number of lines of code
 - (b) Lowers effort
 - (c) Removes redundancy
 - (d) All of the above

26. What is the relationship between a class A and another class B which includes the object of type class A as its attribute?

 - (a) has-a
 - (b) is-a
 - (c) uses-a
 - (d) includes-a

- 27.** Generalization and specialization relationship between two classes is known as:
- (a) has-a relationship
 - (b) uses-a relationship
 - (c) is-a relationship
 - (d) includes-a relationship
- 28.** The Booch methodology can be broadly divided into:
- (a) Micro process and major process
 - (b) Macro process and minor process
 - (c) Macro process and micro process
 - (d) Macro process and mini process
- 29.** The Rumbaugh methodology is popularly known as:
- (a) Object method technique
 - (b) Object-oriented design
 - (c) Object modelling technology
 - (d) Object modelling technique
- 30.** The analysis phase of the OMT method consists of:
- (a) Class model, static model, functional model
 - (b) Object model, dynamic model, functional model
 - (c) Object model, static model, functional model
 - (d) Object model, static model, dynamic model
- 31.** OOSE stands for:
- (a) Object-oriented system engineering
 - (b) Object-oriented system evolution
 - (c) Object-oriented software evolution
 - (d) Object-oriented software engineering
- 32.** The Jacobson methodology is popularly known as:
- (a) Object modelling technique
 - (b) Object-oriented software engineering
 - (c) Object-oriented design
 - (d) Object-oriented analysis and design
- 33.** Which of the following objects are identified in the analysis model?
- | | |
|-----------------------|------------------------|
| (i) Analysis objects | (ii) Entity objects |
| (iii) Control objects | (iv) Interface objects |
- (a) (i), (iii), (iv)
 - (b) (i), (ii), (iv)
 - (c) (ii), (iii), (iv)
 - (d) All of the above
- 34.** OMT stands for:
- (a) Object modelling technique
 - (b) Oriented modelling technique
 - (c) Object modelling technology
 - (d) Object mobile technique
- 35.** The Jacobson methodology is:
- (a) Stronger in analysis part and weaker in design part
 - (b) Stronger in design part and weaker in analysis part
 - (c) Stronger in behavioural areas and weaker in other areas
 - (d) Stronger in both design and analysis part
- 36.** Object-oriented modelling produces:
- | | |
|----------------------------------|---------------------------|
| (a) Robust design | (b) High-quality software |
| (c) Well-understood requirements | (d) All of the above |

- 37.** UML combines methodologies given by:

 - (a) Rumbaugh and Jacobson
 - (b) Rumbaugh and Booch
 - (c) Booch and Jacobson
 - (d) Booch, Rumbaugh and Jacobson

38. Which one of the following is true?

 - (a) Customers are persons who approve and pay for the system and users are persons who use the system
 - (b) Users are persons who approve and pay for the system and customers are persons who use the system
 - (c) Customers are persons who develop the system and users are persons who use the system
 - (d) Customers are persons who develop and use the system

39. Process is:

 - (a) The way in which software documents are developed
 - (b) The way in which software is produced
 - (c) The end product
 - (d) The procedure to measure the progress of the software

40. Use case scenario depicts:

 - (a) Functionality of the use case
 - (b) Internal details of the use case
 - (c) Path of the use case
 - (d) None of the above

41. What is a software metric?

 - (a) An ISO property
 - (b) Quantitative measure of the degree to which a product or process possesses a given attribute
 - (c) Qualitative measure of the degree to which a product or process possesses a given attribute
 - (d) Indication of the extent, amount or size of some attributes of a product or process

42. Software validation is also known as:

 - (a) Dynamic testing
 - (b) Source code design
 - (c) Alpha testing
 - (d) Static testing

43. Verification activities come under the category of:

 - (a) Dynamic testing
 - (b) Source code design
 - (c) Alpha testing
 - (d) Static testing

44. Which of the following is **not** a factor of software quality?

 - (a) Reliability
 - (b) Maintainability
 - (c) Usability
 - (d) Invisibility

45. A bug is a:

 - (a) Fault in source code
 - (b) Fault in design
 - (c) Fault in requirement
 - (d) Similar to fault

46. UML is related to:

 - (a) Object-oriented concepts
 - (b) Operation-oriented concepts
 - (c) Procedure-oriented concepts
 - (d) All of the above

- 47.** CMM certification is related to:
- (a) Product
 - (b) Process
 - (c) Capability of a developer
 - (d) Capability of a company
- 48.** Reliability is related to:
- (a) Objects
 - (b) Classes
 - (c) Failures
 - (d) Operations
- 49.** Testing is related to:
- (a) Verification
 - (b) Validation
 - (c) Both (a) and (b)
 - (d) None of the above
- 50.** Object-oriented approach is:
- (a) Non-iterative
 - (b) Iterative
 - (c) Highly iterative
 - (d) None of the above

Further Reading

A good number of concepts of software engineering are explained in the following books:

Pressman, R., *Software Engineering: A Practitioner's Approach*. New York: Tata-McGraw-Hill, 2005.

Somerville, I., *Software Engineering*. Reading, MA: Addison-Wesley, 2004.

The object-oriented concepts are effectively addressed in:

Korson, T., and McGregor, J., Understanding object oriented: A unifying paradigm. *Communications of the ACM*, 33(9), September 1990.

The basic concepts of object-oriented programming in a language-independent manner are presented in:

Budd, T., *An Introduction to Object-Oriented Programming*. Pearson Education, India, 1997.

The definitions of object, class, inheritance and aggregation may be read from:

Berard, Edward V., *Basic Object-Oriented Concepts*, <http://www.ipipan.gda.pl/~marek/objects/TOA/oobasics/oobasics.html>

The overview of object-oriented software engineering methodologies is presented in:

Jacobson, I.V. *Object Oriented Software Engineering: A Use Case Driven Approach*. New Delhi: Pearson Education, 1999.

Some of the useful facts about software may be found in:

Glass, R.L., *Facts and Fallacies of Software Engineering*. Pearson Education, 2003.

The details about the methodologies described in this chapter can be obtained from:

Coad, P., and Yourdon, E., *Object-Oriented Analysis*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

Jacobson, I.V., et al., *Object Oriented Software Engineering*. Pearson Education, 1999.

Booch, G., *Object-Oriented Design with Applications*. Redwood City, CA: Benjamin-Cummings, 1991.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

2

Software Development Life Cycle Models

When we decide to develop a software product, many initial questions focusing on the need of the software come into our mind, such as “Would it be better to electronically generate mark sheets?,” or “Is there any market for the word processing system?,” or “Is it economically viable if we electronically generate payroll of an employee?”. Once the needs are determined, the software goes through a series of phases of development. These phases include analysis, design, implementation, testing, maintenance and retirement. The software remains useful between the analysis and maintenance phases and ends with the retirement phase. The period from which a software development process starts till the retirement phase is known as life cycle of the software.

The life cycle period of one software differs from that of another. The software development life cycle (SDLC) models depict the procedure used for development of the software in its life cycle. A number of software development life cycle models are available in literature and are based on the methods and techniques used for developing and maintaining the software. In this chapter, we describe a variety of software development life cycle models. The three widely used models are described which include waterfall model, prototyping model and iterative enhancement model. We also present spiral model which has received attention in past few years. In addition, an emerging model extreme programming is introduced. Finally, two object-oriented software development models (fountain model and rational unified process model) are discussed.

2.1 Conventional Software Life Cycle Models

In software engineering, the conventional software development life cycle models are based on methodology followed in the basic model popularly known as “waterfall” model. The software

development life cycle is generally viewed in terms of functions or processes. The software is developed using functional decomposition techniques and the design is implemented in terms of functions or operations. The final software is seen primarily in terms of functions or modules. More importance is given to modules or functions as compared to data. A top-down analysis and design methodology are followed. The methodologies used in conventional software development life cycle for requirement analysis, design and implementation are shown in Figure 2.1.

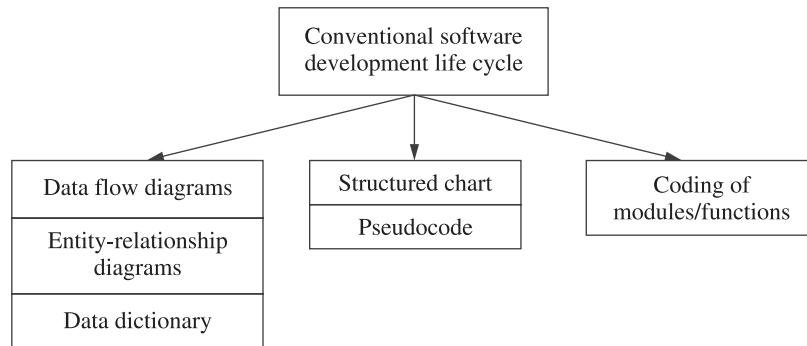


Figure 2.1 Methodologies used in conventional software development life cycle.

In the next subsections, we describe six popular conventional software development life cycle models.

2.1.1 Build-and-Fix Model

Some projects are built without conducting analysis and design. The product is simply built and tested again and again until the customer's needs are satisfied. This approach is termed build-and-fix model and is shown in Figure 2.2.

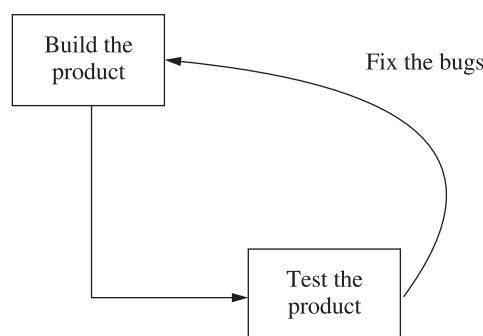


Figure 2.2 Build-and-fix model.

The build-and-fix model may work well on small programs of approximately 50–200 lines of source code, but it is not suitable for reasonable sized programs. The cost of correcting defects increases exponentially if the defect is detected in the later phases of software development. Hence, the cost of fixing defects of the program developed using build-and-fix model is higher.

as compared to the cost of defect correction in any program which is developed using any other specified software development life cycle model. The maintenance of the software also becomes very difficult without any requirement or design documents. In addition, the source code is modified a number of times using ad hoc build-and-fix approach which makes the source code poorly structured and unenhancable after some period of time. Many a time even a well-developed program does not match the customer's requirements and thus is not useful for the customer. This necessitates the need of effective requirement analysis phase before the development of the source code. Hence, a mature approach must be followed and a life cycle model with pre-defined phases such as requirement analysis, design, implementation, testing, and maintenance must be selected.

2.1.2 Waterfall Model

A disciplined and organized approach for software development is used in waterfall model and is shown in Figure 2.3. It has five phases: requirement analysis, design, implementation and unit testing, integration and testing, and deployment and maintenance. The phases are completed in sequential order and do not overlap with each other. Each phase defines its own set of functions that are distinct from those of the other phases. Each phase should be completed before the commencement of the next phase. The output from one phase goes as input into the next phase. The relationship between phases is explained in Rovce (1970) as:

The ordering of steps is based on the following concept: that as each step progresses and the design is further detailed, there is an iteration with the preceding and succeeding steps but rarely with the more remote steps in the sequence. The virtue of all of this is that as the design proceeds the change process is scoped down to manageable limits. At any point in the design process after the requirements analysis is completed there exists a firm and closeup, moving baseline to which to return in the event of unforeseen design difficulties.

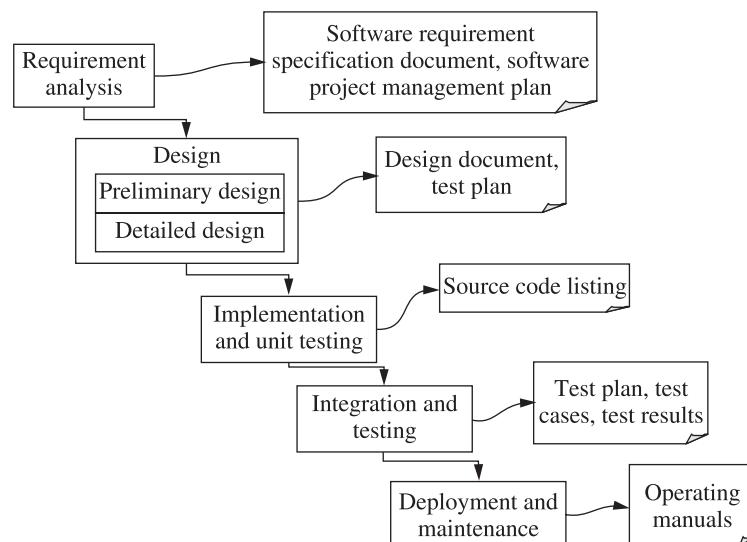


Figure 2.3 Waterfall model.

The waterfall model begins with the requirement analysis phase which describes “what” of a system. This phase gathers requirements from the customer. At the end of the requirement analysis phase, software requirement specification (SRS) document and software project plan are produced. The SRS document may act as a contract between the customer and the developer. After this phase, the requirements are freezed. In the design phase, the software requirement specifications given in the requirement analysis phase are transformed into design structure that is suitable for implementation in a specified programming language. The design of the software describes “how” of the system. The design phase first constructs the preliminary design and then the detailed design. The software design description document and initial test plan are produced at the end of the design phase.

During the implementation phase, the design is transformed into source code. In unit testing, the small modules are tested in isolation and the overhead code is written for handling communication amongst these modules. After implementation and unit testing, the modules are integrated to form a complete system. Integration and testing are carried out to verify the functioning of the software. Testing guarantees the functioning of the system as a whole. Testing is an essential and expensive activity that ensures that a software with high quality, lower maintenance costs and greater accuracy and reliability is delivered to the customer. After adequate testing, the software is given to the customer for acceptance testing. The deliverables of integration and testing phase are test plans, test cases and test results. Finally, if the software is developed as per the customer’s requirement, then it is delivered to the customer and installed at the customer’s site. Thus, the software goes into the deployment and maintenance phase. Software maintenance is a very crucial activity that involves defect correction, quality improvements, enhancements and changes. The operating manuals including user manuals and system guides are outcome of this phase.

The advantages of the waterfall model include its simplicity, ease of understandability and distinct phases with their own set of functions. It is highly suitable for the projects where requirements are completely known in the beginning of the project. The waterfall model has the following disadvantages:

1. A large number of documents are produced. The customer may not understand the formal terminologies and notations used in these documents. The software is built on the basis of written document that the customer may have partially understood.
2. It freezes the requirement at the end of the requirement analysis phase. The requirements may not be well understood by the customer and the developer at the beginning. Thus, it is unrealistic to determine all the requirements at the start of the project itself.
3. The customer does not get the opportunity to see the working product until late after the implementation and testing is completed. The customer may find the software different than what he/she actually wanted.
4. It may take years to complete the project, and at that point, technology may become old and obsolete or the customer’s expectations may change.
5. Testing the system as a whole becomes difficult.
6. Real-life projects are rarely sequential.
7. This approach is particularly not useful in the case of interactive user application.

One of the disadvantages of the waterfall model is overcome by building prototypes to determine the customer's needs. The prototyping model is described next.

2.1.3 Prototyping Model

In prototyping model, a working prototype is constructed in order to determine and understand the customer's needs. The customer evaluates this prototype and then the final SRS document is prepared on the basis of refined requirements. The prototyping model is shown in Figure 2.4. Since the prototype will finally be discarded, it need not be flexible, maintainable and fault tolerant.

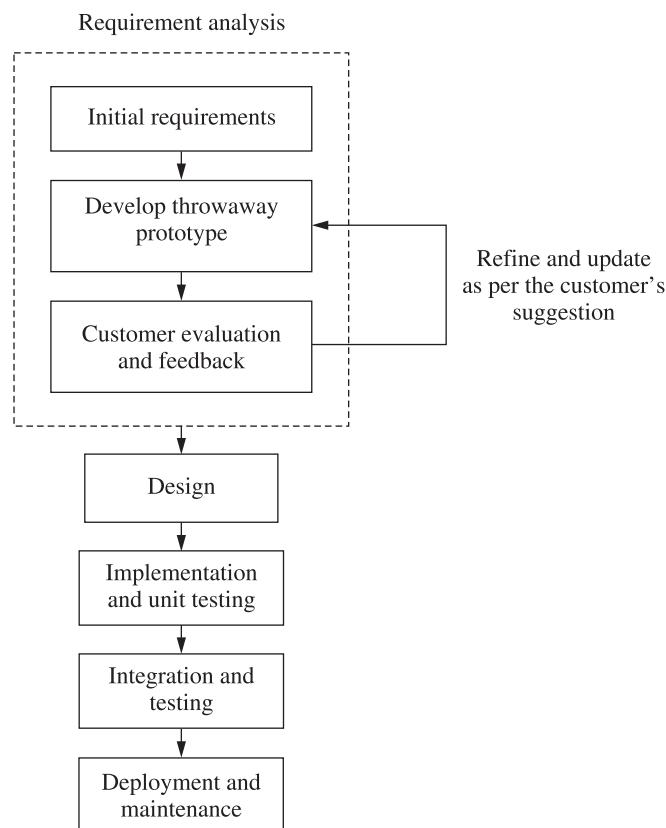


Figure 2.4 Prototyping model.

A throwaway prototype is developed which undergoes design, coding and testing but these phases are not carried out thoroughly and rigorously. The throwaway prototype is discarded once the requirements are determined and the final system is developed from the scratch. This prototype is generally built quickly to give the customer the feel of the system.

During the requirement analysis phase a “quick and dirty” prototype is constructed and given to the customer to obtain feedback in order to determine the necessary and correct requirements. The user interfaces are also validated. After obtaining the customer feedback,

the prototype is refined until the requirements are stabilized/finalized. After the requirements have been determined, the prototype is thrown away. The final SRS document is prepared which serves as input to the next phase similar to that of the waterfall model.

The advantages of the prototyping model are as follows:

1. *Stable requirements:* Requirements are stabilized by refining the prototype again and again based on the customer's feedback. Thus, the probability of change in the requirements after developing the final SRS will be minimized.
2. *High-quality system:* The quality of the system will be high as the developers and customers have gained a lot of experience while constructing the prototype.
3. *Low cost:* The cost of the actual system will be reduced.

The disadvantages of the prototyping model are that the customer may expect quick delivery of the software, and the time for development may exceed in some cases.

2.1.4 Iterative Enhancement Model

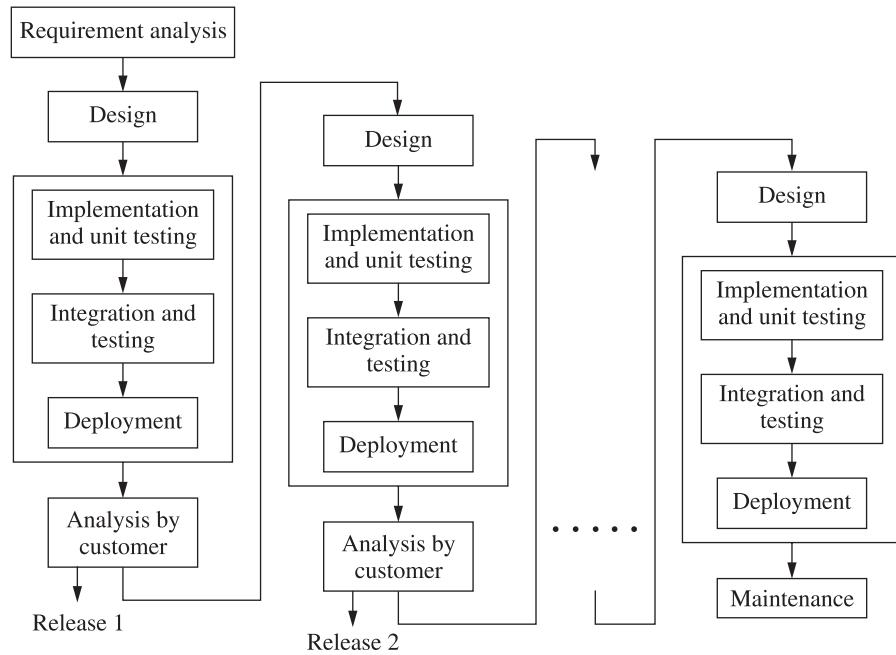
In the waterfall model, the product is delivered after the completion of implementation and testing phases. This problem of the waterfall model is overcome in the iterative enhancement model. **The iterative enhancement model combines the advantages of both the waterfall model and the prototyping model.** This model is recommended and discussed in Basili and Turner (1975). The phases in the iterative enhancement model are the same as in the waterfall model, but they are carried out in many cycles. A usable product is delivered at the end of each cycle. This allows obtaining the customer's feedback after each cycle. As the product is developed in increments, the testing of the product also becomes easy.

In the requirement analysis phase, the requirements are gathered from the customers. Customers and developers prioritize the requirements. Developers implement these requirements in various cycles based on their priorities. A project control list is created consisting of a set of tasks to be completed before the final delivery of the product. The project control list also keeps track of the required work that needs to be completed before the system is ready for final deployment.

The iterative enhancement model is shown in Figure 2.5. The model delivers an operational product that partially satisfies the customer's requirements at the end of each cycle, where each cycle provides an additional functionality. **The complete product is delivered after many releases (see Figure 2.5).** The iterative process is carried out till all the requirements of the control list are implemented.

The iterative enhancement model broadly consists of three iterative phases—design, implementation and analysis by customer. After the requirement analysis, these phases are carried out repeatedly until complete functionality is delivered to the customer. After the final release, the system goes into the maintenance phase as shown in Figure 2.5.

The iterative enhancement model allows to accommodate changing requirements and adds new features to the software during each phase. Unlike the waterfall model, the iterative process carried through various cycles permits the customer to see the progress of the software continuously. This model is particularly useful when the partial portion of the software is to be quickly delivered to the customer.

**Figure 2.5 Iterative enhancement model.**

In the iterative enhancement model, the later software increments may require modifications in earlier ones which may increase costs.

2.1.5 Spiral Model

Spiral model is developed by Barry W. Boehm in 1986. The spiral model is a **risk-driven model** that deals with uncertainty of the outcome of an activity. Risk measures the combined effect of probability of occurrence of failure and the impact of that failure on the software operation. The failure may be defined in terms of cost and schedule overrun, low quality, occurrence of defects, programmatic risks, etc. High risk activities may cause threat to the project. The spiral model incorporates “risk assessment” into the life cycle phases and is shown in Figure 2.6.

The radial dimension in Figure 2.6 shows the cumulative cost of the project and the angular dimension shows the progress made in the completion of the final software. The spiral model is divided into four phases: determine objectives, alternatives and constraints; evaluate alternatives, identify and resolve risks; develop and verify next-level product; and plan next phases. Each cycle of the spiral model begins with identification of objectives (functional and non-functional requirements, etc.), determination of alternative means of implementing the specified part of the product (design 1, design 2, reuse, buy, component, etc.) and identification of constraints imposed on the completion of the portion of the product (resources, cost, schedule, etc.).

The next phase is to evaluate the alternatives in order to identify the significant project risks. This includes identification of means to resolve the risks. This may apply risk resolution techniques like prototyping, customer interviews, modelling, etc. The third phase involves the development of next-level prototype by addressing all the major risk-related issues. When all

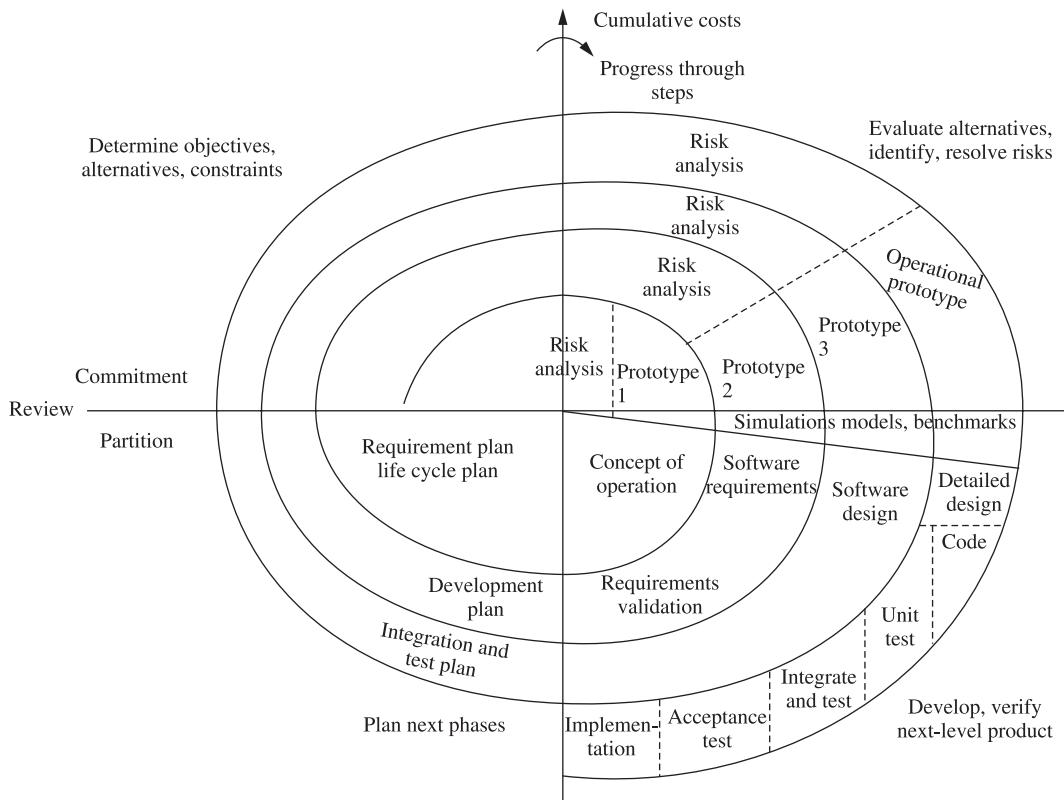


Figure 2.6 Spiral model.

the performance, development and interface-related risks are resolved, the waterfall approach is followed incrementally in the next steps which includes concept of operations, software requirements, software design, implementation and testing. Each level incorporates validation of the activities by conducting reviews and preparing various plans for the next cycle.

The spiral model repeats the four phases to the following rounds as shown in Figure 2.6:

Round 0: Feasibility study

As recommended by Boehm (1986), the initial round 0 must be carried out in order to determine the feasibility of the system in terms of resources, cost, schedule and productivity.

Round 1: Concept of operation

The output of this round is a basic requirement of the life cycle plan for the implementation of the concepts emerged till now.

Round 2: Top-level requirements analysis

In this round, the requirements are specified and validated following the risk-driven approach. The output produced is the development plan.

Round 3: Software design

A preliminary design is created, verified and validated. Integration and test plan is produced after the completion of round 3.

Round 4: Design, implementation and testing

Round 4 constructs the detailed design of the product. This includes coding, unit testing, integration testing and acceptance testing.

The spiral model incorporates validation and verification steps in order to reduce risks and defects in early phases of software development. Thus, it embeds software quality in the life cycle. The spiral model has some disadvantages that include lack of expertise to determine and resolve risks and it is applicable only to large-sized projects. Sometimes the cost of performing risk analysis may outweigh the returns of the spiral model.

2.1.6 Extreme Programming

Extreme programming (XP) project started in the middle of the 1990s. XP is based on the principles of agile processes. Agile means the ability to move quickly and easily. The agile process is both quick and easy. **Agile processes are based on team cohesiveness, quick feedbacks, incremental development, experienced developers and automated testing.** The key features of agile processes are shown in Figure 2.7.

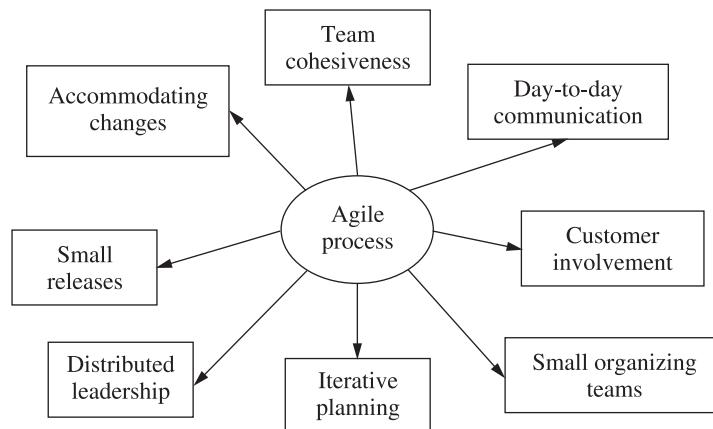


Figure 2.7 Key features of agile processes.

The rules followed in the agile processes are given as follows:

- Team cohesiveness and the way in which a team works are given more importance than any process.
- The customer (a domain expert) is always a part of the team and makes decisions related to deadlines.
- Requirement changes are accepted unlike the traditional approaches where the requirement specification was assumed to be rigid and complete.
- A working software is produced very early and shown to the customer within a few weeks of development.
- The progress is measured by how much of the working software has been built rather than by documents.

- Iterative planning is done instead of iterative development followed in traditional models. Plans are changed based on the learned processes.
- Distributed leadership is a key feature of agile processes. Decisions are not taken by only the project manager as done by traditional strategies of team build-up.

XP follows the methodology of **agile processes** and **focuses on customer satisfaction**. It emphasizes on continuous customer involvement and testing. The importance of an XP case is effectively given by Wake (2002) as:

Extreme programming (XP) is a new, lightweight approach to developing software. XP uses rapid feedback and high-bandwidth communication to maximize delivered value, via an on-site customer, a particular planning approach, and constant testing.

The features of XP include **simple processes, continuous feedback, incremental growth, high communication and courage**. For this, the customer (a domain expert) is available at all the times. This reduces the feedback time greatly as the solution to a problem is obtained in a very short time (few minutes to few hours). The software is developed iteratively and the self-organizing teams adapt to the changes in the methodologies easily. A typical life cycle of an XP is shown in Figure 2.8 and the phases are discussed as follows:

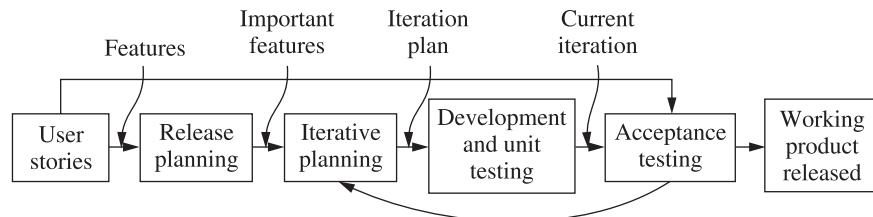


Figure 2.8 A typical life cycle of XP.

1. **User stories:** XP begins with collection of requirements termed “user stories” from the customer. These stories are short and written by the customer(s). User stories differ with traditional requirement specification in the sense that they only provide details for making an estimate of how much time it will take to implement a user story. The detailed requirements are collected at the time of development from the customer. These stories are used in release planning and to create acceptance test plans and test cases (Figure 2.9).

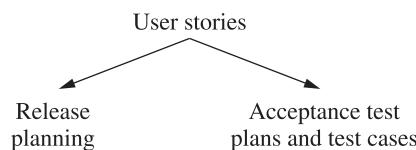


Figure 2.9 User stories.

2. **Release planning:** In this phase, the developers estimate the time to implement the user stories and the customer selects the order in which the stories will be implemented. In

this phase, the developer may do a quick exploration (spike) of a user story in order to estimate its cost. The customer may also divide a user story into substories, when and where required. The steps followed in release planning are shown in Figure 2.10. The release date of each story is finalized. Finally, the customer sorts and extracts the high ranked and high-risk stories.

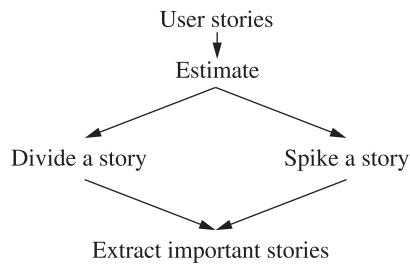


Figure 2.10 Steps followed in release planning.

3. *Iteration planning:* In iteration planning, user stories to be implemented in current iteration are broken into tasks and these tasks are assigned to the developers (selected by their importance). A working product is produced after each iteration, rather than producing a software product at the end of the life cycle as done in traditional approaches. After the iteration planning, the acceptance test cases are generated from the user stories.
4. *Development and unit tests:* In this phase, the most important tasks chosen by the customers are implemented. In order to increase software quality, pair programming is used in which two people work together on the same computer. Refactoring is carried out throughout the development cycle by removing obsolete designs, unused functionality and redundancy. Automated unit tests are created before the source code is developed. Unit tests also enable to identify the need of refactoring. After each change, the developer verifies through the unit tests any change in functionality. The unit testing enables effective functionality and regression testing so that refactoring process is carried out efficiently.
5. *Acceptance testing:* Acceptance test plans and test cases are created from user stories. These tests are carried out to ensure that requirements given by the customer are met and the software developed in current iteration is acceptable. During each development iteration, the acceptance test plans and test cases are prepared from the user stories. Acceptance test cases are created using black box testing techniques. In the acceptance testing, the test cases are run by the customer and the actual result is compared with the expected result. Automated acceptance tests are run and the failed tests are fixed.
6. *Working product released:* After the acceptance testing, the decision about the release of the product is left to the customer. There are frequent releases in which a working product is delivered to the customer in each release.

The XP rules are given as follows:

1. It follows the 80–20 rule which states that 80% of the profits come from 20% of the development.
2. Computers must be placed in the middle of a big room.
3. Tasks which are longer than three days are further broken down and which are shorter than one day are grouped together.
4. Pair programming should not be misinterpreted as a student–teacher relationship.
5. Each iteration must not last more than a few weeks.
6. The customer must always be present at the developer's site.
7. Spikes are used for elaborating user stories.
8. Continuous refactoring is followed.
9. Class responsibility collaboration (CRC) cards are used to design the system. These are used to determine classes and interactions between them.
10. The entire team is responsible for the system's architecture.
11. Integration is done within every few hours.
12. The developer cannot do overtime for the second consecutive week.

The main drawback of XP is that it is “**hard to do simple things**”. XP is not suitable for projects where high documentation is required by the customer. It is also not suitable in cases where the customer cannot be involved continuously. **XP may not be used where technology is complex and/or the time to obtain feedback as a consequence of integration becomes difficult.**

2.2 Object-Oriented Software Life Cycle Models

As discussed in Chapter 1, object-oriented concepts include classes, objects, inheritance, and polymorphism. Object-oriented system models deal with real-world objects. These objects may include human entities, business objects, data and storage constructs. The bases of the object-oriented methodology are objects incorporating attributes and functions that operate on these attributes. For example, in the case of result management system, a student is an object, course is an object and even subject is an object. Thus, the object-oriented methodology concerns with real-world objects and interrelationships between these objects.

Although the phases of the object-oriented software life cycle are similar to the traditional life cycle models, they follow different methodologies of analysis and design. The focus of the object-oriented software life cycle is on objects rather than on processes as in the case of traditional waterfall and other models. The object-oriented software life cycle models enable the designers and coders to properly design the objects and the interrelationships between these objects. This allows to produce a simple-to-understand, modular, flexible and reusable design of the object-oriented software under development. Table 2.1 highlights the major differences between the phases of traditional models and object-oriented software life cycle models.

Table 2.1 Comparison between phases of traditional and object-oriented approaches

	Traditional approach	Object-oriented approach
Methodology	Functional and process driven.	Object driven.
Requirement analysis phase	Data flow diagrams, data dictionary, ER diagrams.	Use case approach for requirement capturing.
Analysis phase	Data flow diagrams, data dictionary, ER diagrams.	Object identification and description, attributes and functions that operate on those attributes are determined.
Design phase	Structured chart, flow chart, pseudo code.	Class diagram, object diagrams, sequence diagrams, collaboration diagrams.
Implementation and testing phase	Implementing processes, functions, unit, integration and system testing.	Implementing objects and interactions amongst the objects. Unit, integration and system testing.
Documentation	Many documents are produced at the end of each phase.	A document may or may not be produced at the end of each phase.

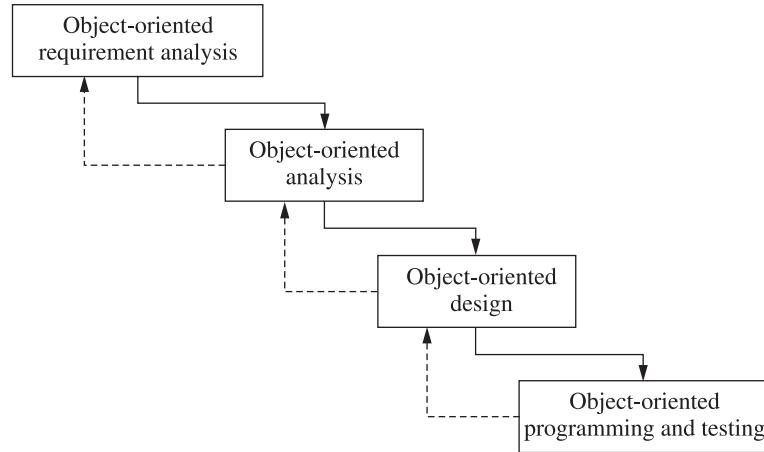
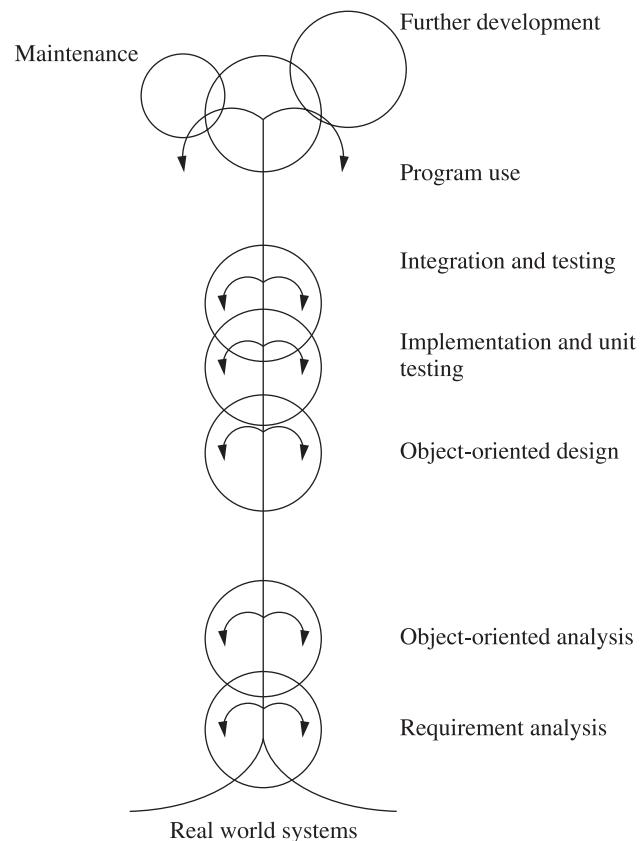
The object-oriented software development life cycle includes the following phases:

1. *Object-oriented requirement analysis:* In this phase, the use case approach is used for capturing requirements from the customer. The SRS document containing the use case description may be produced during this phase.
2. *Object-oriented analysis:* In this phase, the objects and the interrelationships between these objects are identified. The functions of the objects are determined. An ideal structure is created in order to build high-quality maintainable software.
3. *Object-oriented design:* In this phase, the objects are defined in detail keeping in mind the implementation environment. All the objects identified during the object-oriented analysis phase must be traced to the design phase.
4. *Object-oriented programming and testing:* The objects, the functions encapsulated in these objects and interactions amongst the objects are implemented in terms of source code. The existing developed and well-tested components may be used and new components are created. This feature supports reusability of the source code. The developed components are tested using object-oriented testing techniques.

A typical object-oriented life cycle model is depicted in Figure 2.11.

2.2.1 Fountain Model

The fountain model provides a clear representation of iterations and overlapping phases. The model emphasizes the reusability of the source code. Figure 2.12 shows the fountain model. The circles depict the overlapping phases and the arrows within the circles depict iterations within the phases. The smallest circle represents the maintenance phase showing that the maintenance phase is the smallest in the fountain model.

**Figure 2.11 Object-oriented life cycle model.****Figure 2.12 Fountain model.**

Changes can be easily made between class specification and requirement specification. Thus, there is no need to freeze the requirements in the early phases of software development. Since the system is developed in terms of classes, the phases of the software development life cycle can be applied to individual classes rather than system as a whole (Henderson-Sellers and Edwards, 1990).

The fountain model may follow an undisciplined approach in which the developer moves randomly between phases.

2.2.2 Rational Unified Process

Rational unified process (RUP) is maintained by Rational Software. It provides a process framework that may be adapted by organizations according to their needs. The RUP supports iterative software development where the software goes through a series of short length mini projects called *iterations*.

The RUP forms the basis for providing guidelines on the use of Unified Modelling Language (UML). The **UML is a popular standard for visually modelling elements of the system and is governed by Object Management Group (OMG)**. The UML is a language for creating, visualizing and documenting various diagrams that depict different aspects of the software.

The RUP emphasizes iterative software development, effective requirement capturing, visual modelling, use and development of reusable components, ensuring quality of the software, change control management and automated testing. Figure 2.13 depicts the key features of RUP, which are discussed here.

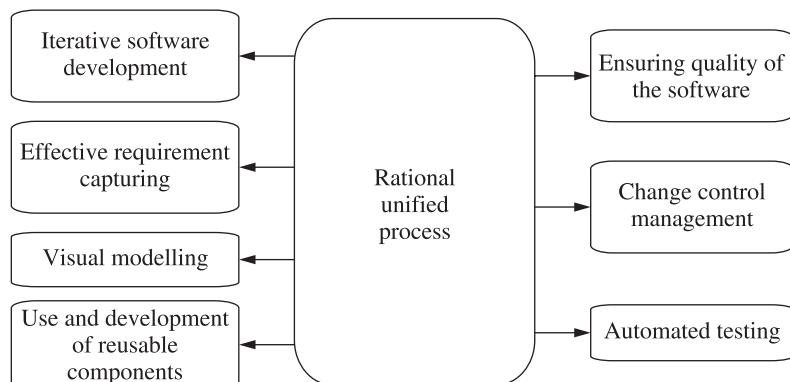


Figure 2.13 Key features of rational unified process.

1. *Iterative software development:* The software is developed through a series of iterations. This allows the customer to provide continuous feedback after each iteration and hence reduces risk and increases quality of the software. The iterative development helps monitoring the schedule and budget of the project and also makes easier to accommodate changing requirements.
2. *Effective requirement elicitation:* The RUP promotes the use of an effective requirement elicitation technique popularly known as *use case approach*. Every requirement is

verifiable and traceable and every piece of design may be traced back to one or more requirements in the software. It ensures that effective testing is carried out by generating test cases from use cases. These test cases are used in acceptance testing and ensure that the final software produced fulfils the user's requirements.

3. *Visual modelling:* Visual modelling creates models to solve problems around the real world. Visual modelling provides abstraction by hiding non-essential details and building models that portray different views of the system. Models are an efficient way to understand, visualize and document the real-world objects. Modelling safeguards the system by providing understandable and manageable requirements, better software designs and flexible software. The UML is an industry acceptable standard that provides the foundation for visual modelling by using notations, models and diagrams.
4. *Use and development of reusable components:* The RUP supports component-based software engineering. A component is an independent subsystem that fulfils a goal in a clear fashion. The well-coded and thoroughly tested existing components are used and new well-designed components are created to promote reuse.
5. *Ensuring quality of a system:* The cost to fix defects increases drastically as one moves from the requirement phase to the testing phase and finally to the maintenance phase as shown in Figure 2.14. Hence, quality must be continuously assessed in order to produce a flexible and manageable system.

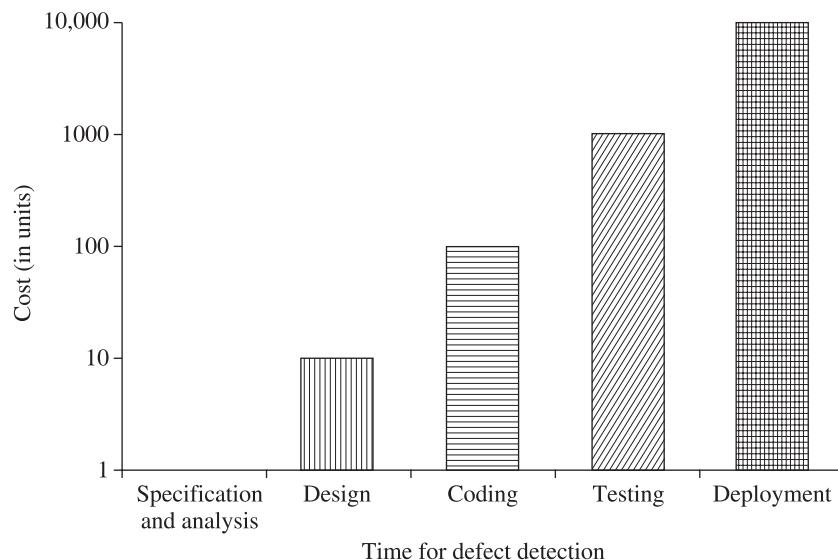


Figure 2.14 Phase-wise cost of fixing a defect.

Software quality is composed of many attributes such as **usability, reliability, maintainability and portability**. The RUP provides guidelines for continuous quality assessments, product evaluation and process monitoring. It puts emphasis on building quality into the system from its inception. Quality is assessed by using measurement-based techniques and models through which the quality of the software is determined.

6. *Change control and management:* The RUP provides guidelines for managing, controlling and tracking changes in order to enable effective iterative software development.
7. *Automated testing:* The RUP promotes the use of automated testing. The automated testing does repetitive testing, unattended without any human intervention. Both functional and non-functional requirements can be tested by using a tool. This saves much of the effort, time and resources.

RUP Overview

The process can be divided into two structures:

1. *Static structure:* It provides the process description in terms of roles, activities, artifacts, disciplines and workflows.
2. *Dynamic structure:* The dynamic aspect of the process can be viewed in terms of iterative development.

The structure of an RUP is shown in Figure 2.15. The vertical dimension depicts the static structure of the process and the time dimension depicts the dynamic structure of the process. Each activity on the time dimension is carried out iteratively. As it can be seen from Figure 2.15, implementation is started much later in the software development. Most of the requirements are predefined in the early phases, but some new requirements may be added in the later stages of software development. Testing is carried out throughout the software life cycle whereas deployment activities begin much later in the software development. Although the focus on activities keeps on increasing and decreasing throughout the software life cycle, they can be carried out at any time during the software life cycle.

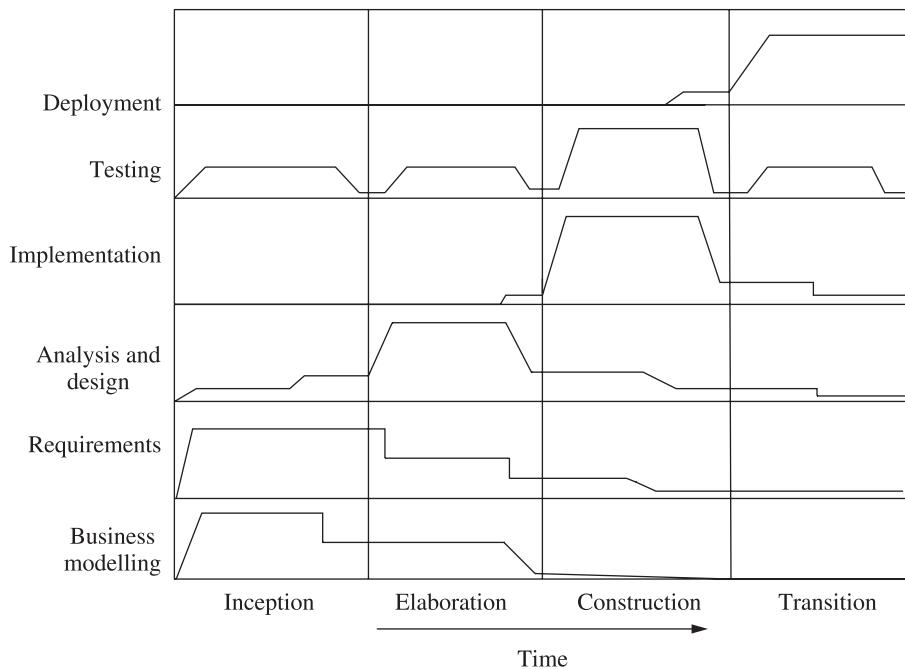


Figure 2.15 Structure of an RUP.

Static structure of RUP: The static structure of the RUP describes who (roles) does how (activities), what (artifacts) and when (workflows). Five major elements form the static structure of the RUP: role, activities, artifacts, disciplines and workflows. Figure 2.16 presents the relationship between roles, activities and artifacts.

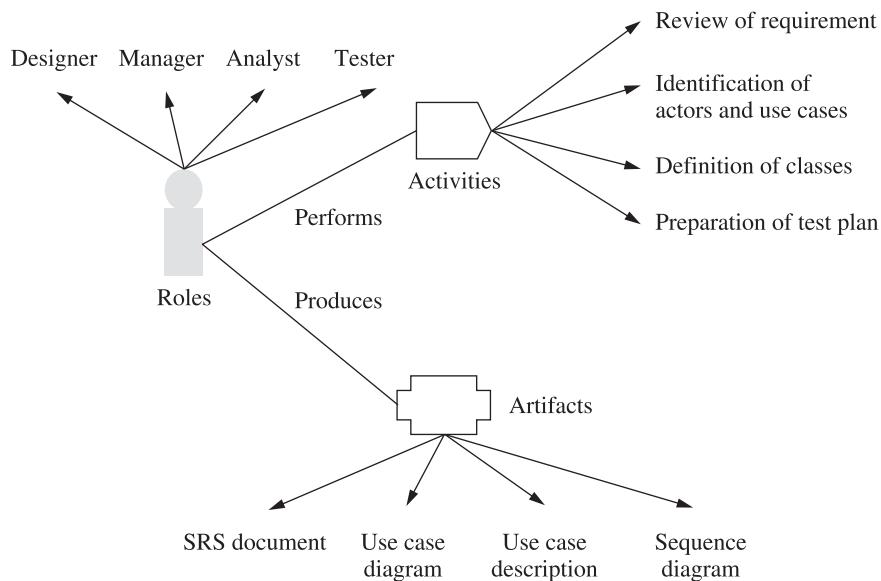


Figure 2.16 Relationship between roles, activities and artifacts.

Roles: A role describes functions or position that a person is expected to have in a project. The functions are known as activities that a person has to perform in order to achieve an artifact. A person may have many roles. For example, as shown in Figure 2.17, Rita plays the role of a designer and a use case creator simultaneously. The role of the person may keep on changing depending on the project. For example, Ram may be a project manager at one time and a design reviewer at the other time.

The roles of four persons involved in an example application are shown in Figure 2.17. In the figure, each role is associated with the corresponding activities to be performed by the role player. For example, Mary is a designer and is responsible for creating class design and identification of attributes and functions of a class.

Activities: Activities are the work performed by a person in a specific role to produce the required result from the system. Activity is expressed in terms of creation or design of an artifact such as use case, object, state chart, etc. The following are examples of activities performed by persons in different roles:

1. Estimation of effort and schedule
2. Design of classes
3. Review of SRS document
4. Verification of quality of system design
5. Creation of test plan and test cases

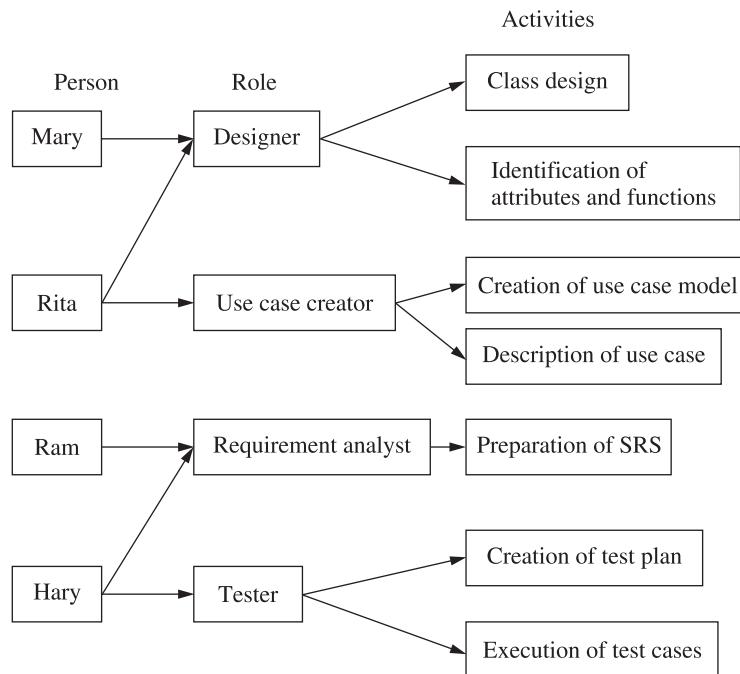


Figure 2.17 Roles and activities.

An activity may be further decomposed into various subactivities. For example, design of a class may be decomposed into the following steps:

1. Identification of attributes
2. Identification of functions
3. Identification of interaction amongst functions
4. Description of functions
5. Description of relationship of a class with other classes in the class model
6. Evaluation of the relationship amongst classes
7. Development of a description of the class
8. Review of the results

Artifacts: Artifacts are the outputs produced during the development life cycle of the software. They may be end products produced or inputs used by the activities while developing the software. An activity is performed by a role to produce an artifact. The following are the examples of artifacts:

1. Software requirement specification
2. Use case model
3. Class model
4. Test plan
5. Source code
6. User manual

Disciplines: Disciplines are used to organize a set of activities. An RUP consists of six major disciplines: business modelling, requirements, analysis and design, implementation, testing and deployment. These disciplines are traced again and again throughout the software development life cycle.

Workflows: Workflows consist of a series of activities to produce a relevant output.

Dynamic structure of RUP: The dynamic structure of an RUP is organized along time as shown in Figure 2.18. It consists of four phases as shown in the figure: inception, elaboration, construction and transition. Each phase achieves a predefined milestone that ascertains the status of the project.

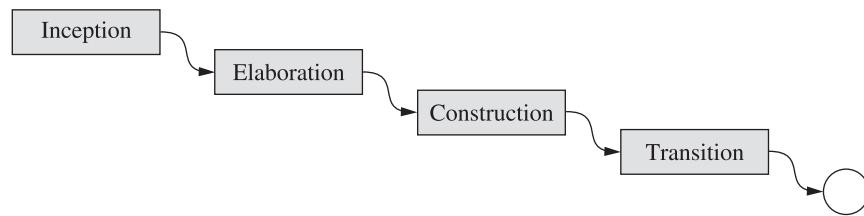


Figure 2.18 Phases of the process.

The distribution of effort and schedule is different in all the phases and is shown graphically for a medium-sized project in Figure 2.19.

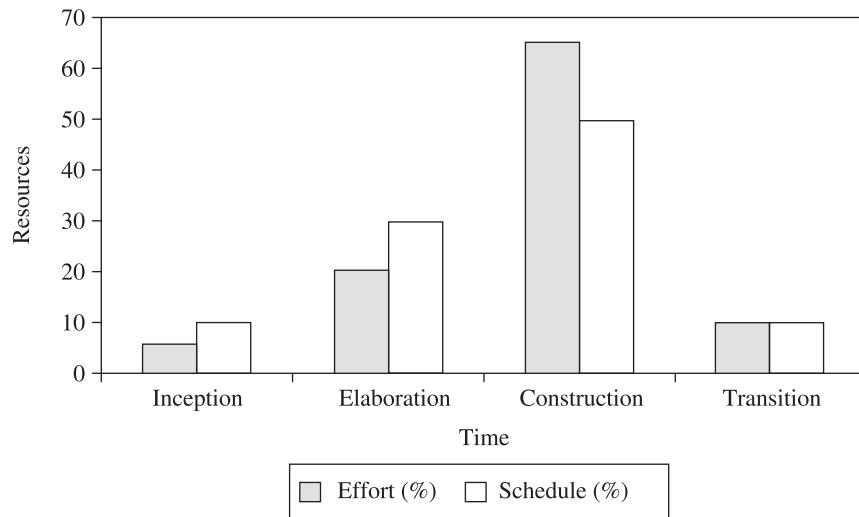


Figure 2.19 Phase-wise distribution of resources.

The RUP follows an iterative development process as shown in Figure 2.20. The first pass through inception, elaboration, construction and transition produces the first version of the software. The user feedback is obtained along with defect reports and improvement needs. The same process is repeated to evolve the next generation of the software.

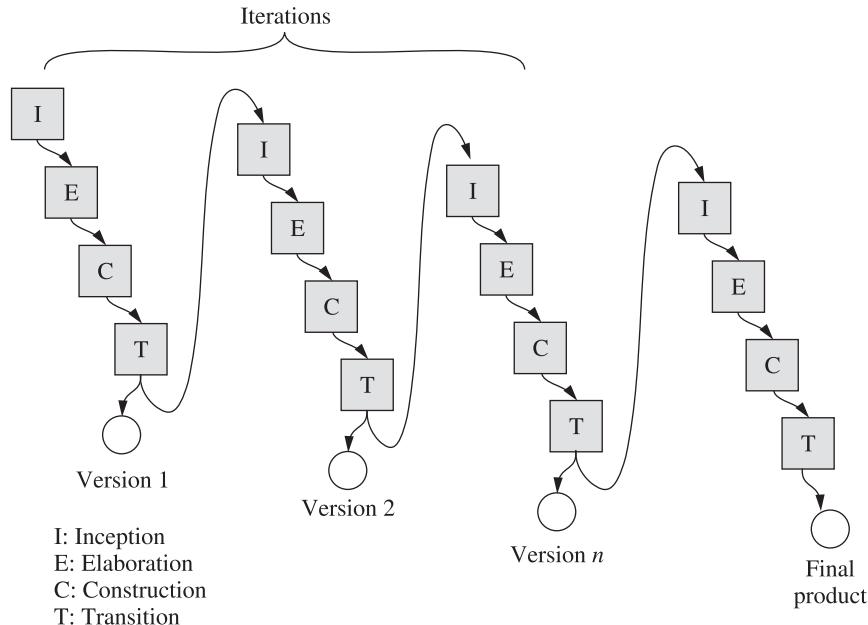


Figure 2.20 Iterative RUP.

Inception: The inception phase begins with the following questions: How long would it take to complete the project? How feasible is the project? What are the high-level requirements? In the inception phase, business problem is defined, an actors (external entities that interact with the system) and use cases are identified. The significant use cases are described.

Inception is basically a non-iterative process as compared to the other phases which are highly iterative. An iteration plan is developed that describes during which iteration which use case will be implemented. The essential activities of the inception phase are as follows:

1. Establishment of scope and boundary of the project.
2. Determination of cost and schedule of the project.
3. Identification of actors and use cases.
4. Development of initial iteration plan.
5. Determination of high-level risks.

The artifacts produced by the inception phase are shown in Figure 2.21. The vision document addresses the scope and requirements of the project. The main constraints of the project are also identified. Business case is defined and an initial list of risks involved in the project is prepared. The objectives and durations of the initial phases are determined in software development plan. The iteration plan consists of details of which use case to be covered during which iteration. The required tools in the project are listed. An initial glossary consisting of definitions of important terms is prepared. The initial use case model identifies important actors and use cases along with flow of events for significant use cases. The environment for configuration management is set up by establishment of project repository. Prototypes are very useful for the understanding of the project (optional artifacts).

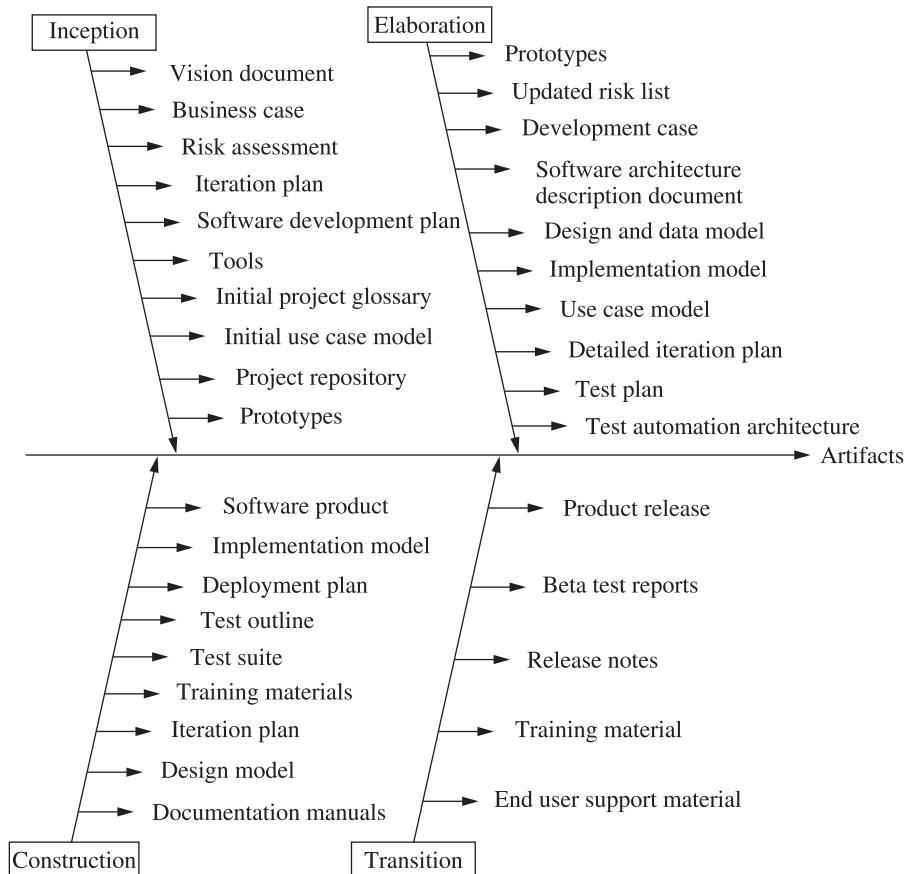


Figure 2.21 Fish bone diagram showing the artifacts produced.

Elaboration: The elaboration phase consists of planning and architectural design. This is the most critical phase of RUP. As given in the iteration phase, the elaboration phase is followed for each use case in the current iteration. In this phase, detailed description of use cases is prepared. The requirements gathered in the form of use cases are documented in the software requirement specification document. The other activities of the elaboration phase include review of use case model and SRS document for the assessment of quality. The essential activities of the elaboration phase are as follows:

1. Establishment and validation of architectural baselines.
2. Addressing significant risks.
3. Design of use case model.
4. Selection of components and formation of policies for their purchase and usage.
5. Creation of detailed iteration plans.
6. Development of prototypes.

The outcomes of the elaboration phase are given in Figure 2.21. Executable architectural prototypes are created to explore significant functionality of the system. Revised and updated

risk list is prepared which primarily handles non-functional requirements. Software architecture description document includes description of use cases and design elements. The design model and data model are also designed. The implementation model consists of initial structure and identifies significant components. The detailed iteration plan for the construction phase is prepared. A use case model (approximately 80% complete) is designed. All the use cases along with actors are identified and described. Test plan includes test cases to validate the system. Test automation architecture is also designed.

Construction: On the basis of architecture and design developed in the elaboration phase, the product is developed and tested in the construction phase. During this phase, the remaining requirements are determined. The deployable products are produced from this phase. The essential activities of the construction phase are as follows:

1. Optimization of resources by avoiding rework and unnecessary coding.
2. Assessment and verification of quality.
3. Testing all the functionalities of the product. Testing activities include unit, integration and system testing.

The outcome includes the software product along with the user manuals. A completed and reviewed iteration plan is produced for the transition phase. A design model with new and updated design elements is prepared. The test outline and test suite are also preserved as they may be required during the maintenance phase.

Transition: The transition phase is started when a usable product with sufficient quality has been built. The objective of this phase is to hand over the software product to the customer. This phase includes delivering, training users and maintaining the software. Elaboration involves beta releases, bug fixes and enhancement releases.

Product release produces the product that conforms to the customer's requirements. The outcome of the transition phase includes various reports and documents such as beta test reports, release notes, training material and end-user support material.

The RUP is iterative in nature and thus has many advantages. It allows to develop flexible, low-risk, reusable and maintainable software. Hence, a good quality software product is expected to be produced from the RUP.

2.3 Selection of Software Development Life Cycle Models

The selection of software development life cycle models depends on various parameters such as nature and type of projects, customers, developers, nature of requirements, associated risks, etc. Table 2.2 summarizes the applications, strengths and weaknesses of six conventional and two object-oriented software development life cycle models discussed in previous sections. The table may help the developers in selecting the right kind of software development life cycle model for their project.

Table 2.2 Summary of conventional and object-oriented software development life cycle models

Model	Applications	Strengths	Weaknesses
Build-and-fix	Very small sized applications (50–200 lines of source code).	Simple and easy to understand.	Only applicable to very small sized programs. Produces unenhancable source code. Requirements are not predetermined. Resultant product is difficult to maintain. Frequent changes produce unstructured source code.
Waterfall	Small and medium-sized projects. Requirements are completely known and understood initially.	Simple and easy to understand. Separation of functionalities in the form of phases. System progress is measurable.	Software is built on the basis of written documents that may be partially understood. Working product is not produced until late in the life cycle. Unrealistic to determine all the requirements at the start of the project. Lack of iterations.
Prototyping	Projects where complete requirements are not known initially. User interactive projects where human-computer interface is desirable.	Stable requirements. High-quality system. Actual cost of developing software is less as compared to waterfall model. Missing and confusing requirements can be identified easily.	Customer expects quick delivery of the software. Less applicable to existing systems as compared to new systems. Time for implementing prototype may exceed.
Iterative enhancement	Projects where important functionalities are delivered quickly.	Customer does not request for unnecessary requirements— avoids requirements bloating. Customer gets to see results quickly. Additional functionality can be added in the later phases. Early delivery of important functionalities.	Later software increments may require modifications in earlier ones which may increase costs. May result in poor structure due to frequent changes made in the source code.
Spiral	Suitable for real-time, mission critical, risk-oriented applications. Suitable for large sized software.	Combines characteristics of most of the models. Includes risk management. No particular start or end of the project.	Experts for risk assessment and resolution may not be easily available. Usable for only large-sized systems. Can be a costly model. Limits reusability.

(Contd.)

Table 2.2 Summary of conventional and object-oriented software development life cycle models (*Contd.*)

Model	Applications	Strengths	Weaknesses
Extreme programming	Small and medium-sized projects that require quick development. Works good for high-risk projects.	Creates software fast with few defects. Less meetings. Emphasizes team work and communication. Continuous measurement reviews.	Hard to implement. Customers may not like to be involved in every activity. Less challenging for professionals. Lack of documentation. Not suitable for larger projects.
Fountain	Object-oriented projects.	Overlapping phases. Short maintenance cycle. Can easily accommodate changing requirements. Supports reusability.	Undisciplined software development approach may be followed in which developer moves randomly between phases.
Rational unified process	Object-oriented large-sized projects.	Complete methodology. Closer to real-world. Support for visual modelling using UML. Test plans and test cases are constructed directly from use cases in the early phases of software development. Supports reusability in the form of use cases and design of components. This helps to improve the quality of the system and also reduces development time. Can easily accommodate changing requirements. Reduced time and effort on integration as it follows iterative process. Online training and tutorials available.	Complex process. Designers should be expert to utilize this methodology. Reuse not possible in the case of a project built using new technology.

Review Questions

1. Explain the build-and-fix model. What are the disadvantages of such a model?
2. What is software development life cycle? Discuss the waterfall model.
3. List the advantages and disadvantages of the waterfall model.
4. Explain the prototyping model. What is the effect of using prototyping model on the overall cost of the software?
5. What is a throwaway prototype? List the advantages of the prototyping model.

6. Discuss the iterative enhancement model with the help of a block diagram. In what conditions is this type of model suitable?
7. Discuss the spiral model with the help of a neat diagram.
8. Compare waterfall model and prototyping model.
9. How is “risk factor” analysed and handled in the spiral model?
10. What are the limitations of the spiral model?
11. What is the significance of rounds in the spiral model?
12. What are agile processes? Explain their key features.
13. Discuss the life cycle of extreme programming. What are the limitations of such an approach?
14. What steps are involved in release planning of the extreme programming approach? What is the purpose of refactoring in extreme programming?
15. Differentiate between release planning and iteration planning.
16. List and explain the extreme programming rules.
17. What is pair programming? Explain its significance in extreme programming.
18. Define the following terms:
 - (a) Spike
 - (b) Agile process
 - (c) 80-20 rule
 - (d) Pair programming
 - (e) Refactoring
19. What is the role of a customer in the extreme programming model?
20. Differentiate between the conventional approach and the object-oriented approach of software development.
21. Explain the fountain model with the help of a diagram. What is the significance of arrows within circles in this model? List the advantages and disadvantages of this model.
22. What is RUP? Explain the various phases of RUP along with their outcomes.
23. Differentiate between the static structure and dynamic structure of RUP.
24. Establish the relationships between roles, activities and artifacts in RUP.
25. Discuss the parameters based on which we may select a software development life cycle model.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. Which is the most popular model for student programs?

(a) Waterfall model	(b) Build-and-fix model
(c) Spiral model	(d) Rational unified model

- 15.** The spiral model primarily deals with:

 - (a) Non-functional requirements
 - (b) Risk management
 - (c) Quality assurance
 - (d) Defect management

16. The radial dimension of the spiral model shows:

 - (a) Progress made in the completion of the final software
 - (b) Schedule of the project
 - (c) Cumulative cost of the project
 - (d) None of the above

17. The angular dimension of the spiral model shows:

 - (a) Progress made in the completion of the final software
 - (b) Schedule of the project
 - (c) Cumulative cost of the project
 - (d) None of the above

18. The disadvantage of the spiral model is:

 - (a) It is not suitable for large-sized projects
 - (b) It produces low-cost software
 - (c) It requires expertise to determine and resolve risks
 - (d) All of the above

19. The extreme programming is based on:

 - (a) Ad hoc approach
 - (b) Agile processes
 - (c) Formal methods
 - (d) Stochastic processes

20. The key features of agile processes include:

 - (a) Working software is produced very early
 - (b) Iterative planning
 - (c) The customer is always present at the developer's site
 - (d) All of the above

21. Which of the following is **not** a rule of agile processes?

 - (a) Working software is produced very early
 - (b) Iterative planning
 - (c) The customer is always present at the developer's site
 - (d) None of the above

22. Spikes are:

 - (a) Quick exploration of user stories
 - (b) Detailed exploration of user stories
 - (c) Ranking of user stories
 - (d) Used to identify risky user stories

23. XP is suitable for:

 - (a) Small and medium-sized projects
 - (b) Large-sized projects
 - (c) Non-changing requirements
 - (d) All of the above

- 24.** XP is **not** suitable for applications where:
- (a) Documentation is required
 - (b) Continuous customer involvement cannot be achieved
 - (c) Technology is simple
 - (d) Pair programming is suitable
- 25.** Object-oriented life cycle models deal with:
- (a) Changing requirements
 - (b) Real-world projects
 - (c) Business objects
 - (d) All of the above
- 26.** In the fountain model, arrows with circles depict:
- (a) Overlapping phases
 - (b) Iterations
 - (c) Sequential phases
 - (d) Reusability
- 27.** In the fountain, model circles depict:
- (a) Overlapping phases
 - (b) Iterations
 - (c) Sequential phases
 - (d) Reusability
- 28.** RUP stands for:
- (a) Risk-oriented unified process
 - (b) Resource uniform process
 - (c) Rational unified process
 - (d) Rational uniform process
- 29.** RUP is maintained by:
- (a) Microsoft
 - (b) TCS
 - (c) Alcatel
 - (d) IBM Rational Software
- 30.** UML stands for
- (a) Unified Model Link
 - (b) Uniform Modelling Language
 - (c) Unified Modelling Language
 - (d) Uniform Microsoft Language
- 31.** The team of unified process development includes
- (a) I. Jacobson
 - (b) B. Boehm
 - (c) Victor Basili
 - (d) L. Briand
- 32.** How many phases are included in the RUP model?
- (a) Two phases
 - (b) Four phases
 - (c) Five phases
 - (d) Six phases
- 33.** The major elements of the static structure of RUP are:
- (a) Workflows
 - (b) Artifacts
 - (c) Disciplines
 - (d) All of the above
- 34.** Which of the following is **not** an activity of elaboration phase?
- (a) Design of use case model
 - (b) Prototyping
 - (c) Execution of detailed iteration phase
 - (d) Establishment of scope and boundary of the project
- 35.** The outcome of the construction phase includes:
- (a) Software product
 - (b) Use case product
 - (c) Software development plan
 - (d) Initial project glossary

Further Reading

The experiences and suggestions of Royce on the waterfall mode are written in:

Royce, W., Managing the development of large software systems. *IEEE WESCON*, August 1970.

The classic approach of iterative enhancement was proposed by Basili and can be further read in:

Basili, V. and Turner, A., Iterative enhancement: A practical technique for software development.
IEEE Transactions on Software Engineering, 1(4): 390–396, 1975.

The overview of object-oriented life cycle is given in:

Henderson-Sellers, B. and Edwards, J., Object-oriented systems life cycle. *Communications of the ACM*, 33(9), September 1990.

An overview of agile processes may be obtained from:

Cockburn, A., *Agile Software Development*. Boston, MA: Addison-Wesley, 2000.

William explains XP with the perspective of programmer, customer and manager in his book:

Wake, W.C., *Extreme Programming Explored*. Boston, MA: Addison-Wesley, 2001.

Beck gives stories, methodology, myths and applications of XP in his book:

Beck, K., *Extreme Programming Explained*. Boston, MA: Addison-Wesley, 2004.

The details on rational unified process can be obtained from:

Rational Software, *Rational Unified Process Version*, 2002.

3

Software Requirements Elicitation and Analysis

Whenever we get a request for software development from a customer, we would like to first understand the project. This understanding may compel us to think about the **feasibility** of the project. Some projects are very good conceptually, but may not be feasible due to limitations of technology, time, cost, market or appropriate skilled manpower. The decision about feasibility also depends upon our understanding of requirements. The non-feasibility of a project must be realized as early as possible, otherwise cancellation of a project in later stages may lead to embarrassment and huge wastage in terms of time, cost and manpower.

The key issue is the capturing of requirements. This is a very difficult and challenging phase of software development life cycle. Most of the requirements are in the minds of the people and extracting them is not an easy task for any developer. After extraction, such requirements should be documented in a specified format. Brooks (1995) has rightly mentioned that:

The hardest part of building a software system is deciding precisely what to build. No other part of conceptual work is as difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

So, if requirements are elicited properly and documented rightly, then it is presumed that effective foundations of the project are laid. In this chapter, we present requirements elicitation techniques. After the requirements are captured, the initial requirements document is prepared. The use case approach widely being used in object-oriented software engineering is explained and the characteristics of a good requirement are defined. Finally, the contents of the SRS are explained. In order to explain the concepts, a case study of library management system is considered throughout this book.

3.1 Case Study: Library Management System

The problem statement for a Library Management System (LMS) case study is presented in this section. The examples from this case study are used to illustrate object-oriented analysis and design techniques throughout this book. The problem statement is prepared by the customer who gives us broad outline of the expectations from the proposed system. The problem statement of the LMS as prepared by the customer is given here.

A software product is to be developed for automating the functioning of a university library. There are three types of members in the library, i.e. students, faculty and employees. There is an upper limit in terms of maximum number of books issued to a member and duration for which a book is issued. This limit may vary from one type of members to other types of members. The LMS performs the following functions:

1. Issue of books

- (a) Books are issued to students, faculty members and employees as per their specified limits of duration and number of books.
- (b) The current date is used as issue date for a book.
- (c) The due date of return of a book is calculated as per specified limit of duration.
- (d) The due date is stamped on the book.

2. Return of books

- (a) Any person can return the issued books.
- (b) If a book is returned after the due date, a fine is charged from the students for each day delayed. The charges may vary from time to time. However, faculty and employees are not charged any fine for late return of a book.

3. Maintenance of information

The LMS maintains the following information:

- (a) Details of all members with their name, address, designation (wherever applicable), phone number and unique identification code.
- (b) Details of all books with their author name(s), publisher, price, number of copies, and ISBN number.
- (c) Details of books issued to every member.

3.2 What is Software Requirement?

Requirements describe the ‘what’ of a system and not the ‘how’. ‘What’ is related to **expectations** from a system. A requirement is defined as “**a condition or capability to which a system must conform**”. Who will provide us these conditions or capabilities? It depends on the type of project. If the project is the enhancement of an existing system or automation of an existing system, then

the existing system may become the source of information, along with the people involved in the system. A generic term is used for the people who are affected directly or indirectly by the system that is being developed and is called *stakeholders*.

3.2.1 Identification of Stakeholders

Identification of stakeholders is a very important task. Every affected person (directly or indirectly) is a stakeholder of the system. All such persons should be identified carefully and a list should be prepared along with their roles in the present system (if available) and also in the system under development. There are many categories of stakeholders and some are discussed here.

Internal People of Customer's Organization

This category contains the internal people of the organization where the new system will be deployed and used. There are two types of stakeholders, i.e. customers and users. Customers request for the development of the system, approve the changes in the system, accept the system and finally pay for the system. Users use the system after its deployment. Both types of stakeholders are important but their expectations may be different. In case of conflict, the customer's expectations are given preference over user's expectations. For example, in the Library Management System (LMS) discussed earlier, the university is the customer and may be represented by a librarian or any other person deputed by the Vice Chancellor. Some of the users of the LMS are faculty, students, employees and data entry operators (DEOs). In the LMS, a librarian (customer) may like to obtain a list of mistakes made by a DEO during entry of data in a day. However, any DEO may not like to have such a provision because it will be a reflection on their performance in terms of mistakes made during data entry. Hence, in the case of conflicting situations, the customer's expectation will prevail over the user's expectations.

External People of Customer's Organization

This category contains the external people of the customer's organization. This may include consultants, domain experts (if engaged only for the project), maintenance persons (if maintenance activities are outsourced) and any outside agency persons responsible for providing rules, regulations and guidelines. All such people are considered as stakeholders.

Internal People of Developer's Organization

Everyone involved in the development of the system are the stakeholders. This may include developers, programmers, requirement writers, testers, project managers, use case writers, graphic designers, etc.

External People of Developer's Organization

This category contains the external people engaged directly or indirectly in the development of the system. This may include consultants, outside domain experts, third party testers and any outside agency persons responsible for providing development and testing guidelines and standards.

3.2.2 Functional and Non-functional Requirements

Requirements are classified into two categories, i.e. functional and non-functional given as follows:

1. **Functional requirements:** They describe ‘what’ the software will do. This ‘what’ part gives us the functional requirements which are nothing but the expectations from the system. Functional requirements are also called product features. Sometimes, functional requirements also specify what the software will not do. They are captured from the stakeholders and related only to the functionality of the system.
2. **Non-functional requirements:** They are nothing but the attributes of software quality. They signify that how well the software does what it has to do. Non-functional requirements are important for the sustainability of the system and some of them are reliability, usability, maintainability, availability, portability, flexibility and testability. These non-functional requirements may make the users happy and satisfied. They help us to measure the quality of the software and are also called quality attributes.

3.2.3 Known and Unknown Requirements

Known requirements are the expectations of the stakeholders and may make stakeholders happy if implemented correctly. Unknown requirements are forgotten by the stakeholders because they may not require now or due to limitation of domain expertise and facilities of available technology. If such unknown requirements are identified, they may add value to the system and may increase the chances of acceptability and sustainability of the system.

Every requirement (known and unknown) may be functional or non-functional. As discussed in the previous section, functional requirements are also called product features and non-functional requirements are called quality attributes.

3.3 Requirements Elicitation Techniques

Requirements are in the minds of the stakeholders. Therefore, it is extremely challenging to find out what they really expect from the proposed system. Wiegers (1999) has rightly said:

Requirements elicitation is perhaps the most difficult, most critical, most error prone, and most communication intensive aspect of software development. Elicitation can succeed only through an effective customer developer partnership.

The objective of effective customer developer partnership is not easy to achieve. Both groups have different mindsets, educational background, vocabulary, communication skills and domain expertise. All such things lead to communication gap which further increases the probability of conflicts, misunderstanding and ambiguities. However, both groups want success of the project. There are many techniques for requirements elicitation. We may use one or more techniques to capture the requirements. One technique may not be sufficient to capture enough details of the expectations. The selection of a technique may be based on the following:

1. It is the only technique that we know.
2. It is our favourite technique.
3. We believe that a particular technique is the best for this project.

The third reason of selection may be the best way to proceed for capturing of requirements. However, in practice, the first two reasons are used to select one or more technique(s) for requirements elicitation.

3.3.1 Interviews

The most popular, simple and conventional requirements elicitation technique is to conduct interviews with the customer. After receiving the request for software development, we would like to understand the project. Hence, we may like to meet the customer to discuss his/her expectations from the project. The interview of the customer may be informal (non-structured) or formal (structured). The first such meeting/interview may help to understand broad expectations. In the informal interview, there is no fixed agenda and open discussions for free flow of view are encouraged. For example, for an LMS described in Section 3.1, we may ask:

- (i) Who is the librarian?
- (ii) Why do you feel the requirement of library management system?
- (iii) How many members are regular visitors?
- (iv) What are the limitations of the present manual library system?
- (v) How many types of memberships are offered?
- (vi) Who is the most conversant with the present system?
- (vii) Is there any opposition of the project?
- (viii) How many persons of library staff are computer friendly?

These types of questions may become instrumental in the understanding of the present manual system. In the structured interview, a proper agenda is prepared along with a questionnaire. The various steps of structured interviews are given in Figure 3.1.

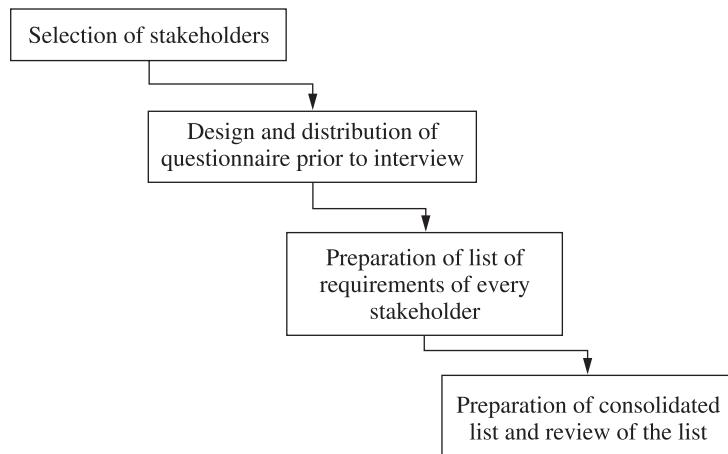


Figure 3.1 Steps of structured interview.

Selection of Stakeholders

There are many stakeholders of any system. All cannot be interviewed. The representatives from a group of stakeholders may be selected on the basis of experience, qualification, intelligence, domain knowledge, credibility, loyalty, etc. There are many groups like entry-level personnel, mid-level personnel, managers, users of the software, top-level management, developers, quality assurance personnel, related vendors, maintenance personnel, etc. Every group has different view points and expectation from the proposed system. One or more persons from every group should be selected and interviewed to understand and capture the requirements.

Design and Distribution of Questionnaire Prior to Interview

We may prepare a list of questions which should be simple and short. Questionnaire should be well drafted and may be distributed to select group of stakeholders prior to interview. During interview, any point may be discussed, clarifications are obtained, and ambiguities are minimized. We may also encourage free flow of ideas and document them properly. A sample questionnaire is given in Table 3.1.

Table 3.1 Sample questionnaire for conducting interviews

Section I	
Name of the stakeholder	
Organization	
Group	
Project	
Date of interview	
Section II	
Please specify your name and role in this project:	
S. No.	Questions
1	Are you satisfied with the manual/present system?
2	What are the problems with the current system?
3	What are the strengths of the present system?
4	What are the limitations of the present system?
5	What are the possible benefits of automating the present system?
6	Are you satisfied with current processes and policies?
7	How are you maintaining the present records?
8	List a few important objectives of the proposed automation.
9	List a few problems which you want to resolve through automation.
10	List point-wise the scope of the project in order of priority.
11	Do you expect additional functionality to improve efficiency and performance of the system?

(Contd.)

Table 3.1 Sample questionnaire for conducting interviews (*Contd.*)

Section II

Please specify your name and role in this project:	
S. No.	Questions
12	List a few stakeholders which must be interviewed.
13	Any specific problem faced in the previous years?
14	List essential expectations from the new system.
15	List essential functional and non-functional requirements.
16	Can you give system diagram that will provide an overview of the system?
17	What information will be stored in the proposed system?
18	What are various threats to the new project?
19	Any suggestions for the success of the project?
20	What mechanism would you like to suggest for the prevention of unauthorized access and protection of your data?

Preparation of List of Requirements of Every Stakeholder

A list of requirements is prepared on the basis of discussion during interview and filled questionnaire. The list is analysed by experts and redundant requirements are removed.

Preparation of Consolidated List and Review of the List

A team of experts should combine the captured requirements. Redundancy should be removed. All requirements are reviewed to remove inconsistencies and ambiguities. A long consolidated list of requirements may help us to understand the expectations from the new system and may become the basis for the design of document. This document may help the developers to know what to build, customers to know what to expect and become a fundamental document to provide foundation to the project. This document is known as initial requirements document (IRD).

EXAMPLE 3.1 Consider the case study of LMS given in Section 3.1. Develop a questionnaire containing a list of questions that may be distributed to the librarian before conducting an interview.

Solution We may frame specific questions for a project. For example, the following questions may be distributed to the librarian before conducting an interview. The questions that may be discussed by the requirement analyst with the librarian during the interview are shown in Table 3.2.

The questionnaire will discipline stakeholders and structure the interview process. All points of the questionnaire will be discussed freely and the questionnaire will be analysed and reviewed to improve the quality of the document.

Table 3.2 List of questions for library management system

Librarian	
S. No.	Questions
1	Do you want to have an automated library management system?
2	What are the problems with the existing manual system of the library?
3	What do you think is the best solution to the problems faced in the existing system?
4	How often are the new members registered?
5	How many users are there in the library?
6	Do you have different types of members? If yes, how many?
7	How many books are present in the library?
8	Do you need a facility to reserve a book in advance?
9	How are the books categorized?
10	What are the reports prepared in the present system?
11	Do you have any format for the reports? Do you want the format of the reports to remain the same?
12	How many books are issued to each type of member and for how many days?
13	Do you have any fine system to penalize the users who return books late?
14	The search procedures should be categorized by: book title, author, publisher or any other.
15	What member and book information do you store in your records?

3.3.2 Brainstorming Sessions

This is also a popular requirements elicitation technique. A group of various stakeholders is constituted to discuss the requirements. The group discussions may encourage new views and ideas quickly and also promotes creative thinking. Group may be heterogeneous where stakeholders from various levels are invited. They may be from entry-level personnel, middle-level managements, project managers, developers, top-level management, actual users, etc. The size of the group may be from 5 to 15 persons. The sessions should be conducted in a conference room with at least two whiteboards and other presentation devices like projector. A facilitator (normally author of the document) explains the purpose of the meeting and presents an already prepared document to sanitize everyone about the purpose. Every stakeholder is allowed to give his/her views using the projector and whiteboard. All views are documented. The idea is to create an environment for free flow of views. No one is allowed to criticize each other.

Brainstorming sessions are very useful for capturing requirements and are used by most of the software companies. These encourage creative thinking, promote new idea and provide a platform to share views, apprehensions and expectations. A long list of requirements is prepared which can further be categorized, prioritized and pruned.

The role of the facilitator is very important. He/she may handle group bias and conflicts carefully. The facilitator is also responsible to handle individual egos and to ensure the conduct of the meeting smoothly. Every idea is written in such a way that everyone can see it. After the session, a detailed report is prepared and the same may be reviewed by the facilitator. All ideas are written in a simple and easy language so that they convey the same meaning to every stakeholder. Incomplete ideas are also documented and must be discussed at any later date.

to make these ideas complete, if possible. Finally, an **initial requirement document (IRD)** is prepared to document the requirements.

3.3.3 Facilitated Application Specification Technique

This is a team-oriented approach and works on the pattern of brainstorming sessions. A joint team of customers and developers is constituted to frame the requirements. All meetings are conducted under the chairpersonship of a facilitator. A facilitator may be appointed from the developers or customers. Sometimes an outsider may also be appointed for this work and is called *third party facilitator*. The objective is to understand the requirements and bridge the gap between expectations and requirements. **Facilitated application specification technique (FAST) is more formal than brainstorming sessions.** All members of the team are required to follow the set procedures and guidelines.

Guidelines for Conducting FAST Session

- (i) FAST session must be conducted at neutral site. All members of the joint team (developers and customers) travel to that site. This may help to get their maximum involvement and focus on the requirements capturing.
- (ii) Rules for participation are framed and circulated to all members in advance.
- (iii) All members are made comfortable to encourage free flow of ideas.
- (iv) The facilitator gives the overview of the project.
- (v) A display mechanism like projector, wall strickers, flip charts, whiteboards, etc. should be available in the committee room where the meeting is conducted.
- (vi) All members should be directed to give only their views. Unnecessary prolonged debates and criticisms should be avoided.

Preparation of FAST Session

Each member is required to make a list of objects that are (i) a part of the environment that surrounds the system, (ii) produced by the system and (iii) used by the system. Each member also makes a list of services (processes or functions) that manipulate or interact with the objects. Finally, a list of constraints (e.g. cost, size, etc.) and performance criteria (e.g. speed, accuracy, etc.) is also developed. 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.

Activities of FAST Session

- (i) Each member presents his/her lists of objects, constraints, services and performance for discussion. A list may be displayed using any display mechanism (say projector) or may be written on the whiteboard in such a way that it is visible to every team member of the team.
- (ii) A small group is constituted to prepare a consolidated list after removing redundant entries.
- (iii) The convener of the small group presents the consolidated list. Discussions take place under the directions of the facilitator. The list is further modified, if required, in order to prepare a consensus list.

- (iv) A few small groups are constituted to draft mini-specifications for one or more entries of the consensus list.
- (v) Each subteam presents mini-specifications to all FAST attendees. After thorough discussions, modifications are carried out in mini-specifications.
- (vi) Some issues may not be resolved during the meeting. A list of such issues is prepared for consideration at any later stage.
- (vii) A validation criterion is also decided for every requirement.
- (viii) A subteam for drafting the specification is constituted. The final draft is prepared considering all inputs of FAST meeting.

FAST is a popular traditional technique for requirements elicitation. It helps us to prepare the IRD in limited time frame under the leadership of a facilitator. Neutral site may help the members to devote more quality time in the requirements capturing and drafting process, although it is an expensive activity as compared to interviews and brainstorming sessions.

3.3.4 Prototyping

Prototyping is one of the expensive but powerful ways to capture requirements. A prototype is a simplified version of the system. When it is shown to the customers, they get an idea about the final system. At this point, their feedback and views are very important and may help us to read their minds. We generally use prototype to understand the requirements. After finalization of the requirements, the purpose of a prototype is over and it is discarded. Hence, we should use the prototype and the final system should be built around the existing prototype. Prototyping is the rapid development of a system for the purpose of understanding requirements. Customers and users can use the prototype to see how the system will look and behave finally. This may reduce the chances of misunderstanding between customers/users and developers. Missing expectations may be highlighted and ambiguous areas may be discussed to make it unambiguous. Initially, prototyping may cost us more but the overall development cost may be reduced. Developers generally use this prototype to refine the requirements and prepare the final specification document. Through prototyping, views of customers are easily incorporated in the IRD. In most of the situations, the source code of the prototype is thrown away; however, experience and feedback gathered from developing and using the prototype helps in developing the actual system. Schach (1999) has rightly mentioned, “The developers should develop the prototype as early as possible to speed up the software development process. After all, sole use of this is to determine the customer’s real needs. Once this has been determined, the prototype is discarded. For this reason, the internal structure of the prototype is not very important”. However, there are people in some organizations who do not believe in this throwaway approach of prototyping. They feel that the prototype provides foundations for the system and huge effort is already invested.

3.4 Initial Requirements Document

After the requirements are captured, the initial requirements document (IRD) may be prepared. The IRD is used to document and list the initial set of requirements gathered through various stakeholders. The template of the IRD is given in Figure 3.2 and the IRD for the LMS is given in Figure 3.3.

Title of the project	
Stakeholders involved in capturing requirements	
Techniques used for requirement capturing	
Name of the persons along with designations	
Date	
Version	
Consolidated list of initial requirements:	

Figure 3.2 Template for initial requirements document.

Title of the project	Library management system
Stakeholders involved in capturing requirements	Librarian, library staff, project leader, students
Techniques used for requirement capturing	Interviewing and brainstorming
Name of the persons along with designations	-
Date	November, 2010
Version	1.0
Consolidated list of initial requirements:	
<ol style="list-style-type: none"> 1. A system is to be implemented which can run on the university's library LAN. 2. A bar code reader should be used to facilitate the process of issue and return of books. 3. The system shall be able to generate login ID and password to the system operator. 4. There are three types of members in the library: students, faculty and employee. 5. The administrator shall be able to maintain details of all the books. 6. The administrator shall be able to maintain details of all the members of the library. 7. The library staff shall be able to issue the books to each of its members. 8. The library staff shall be able to accept the issued books from each of its members. 9. The maximum number of books that can be issued to the student is 5 and to faculty and employees is 10. 10. The system shall calculate fine for late return of books only for students. 11. The library staff shall be able to reserve a book for 24 h. 12. The student/faculty/employee shall access LMS on the university's library LAN to search the availability of a book from the library. 13. The system shall be able to provide the availability of a particular book. 14. The system shall be able to provide the availability of number of copies of a particular book. 15. The system should also be able to generate reports like: <ul style="list-style-type: none"> (i) Details of all books in the library <ul style="list-style-type: none"> • Author-wise • Publisher-wise • Title-wise • Subject-wise (ii) Details of all members. (iii) Details of books issued to members. (iv) Status of fine, member-wise, wherever applicable. 	

Figure 3.3 Initial requirement document for library management system.

3.5 Use Case Approach

The use case approach is primarily designed for object-oriented systems. However, the same may also be used for traditional systems to represent and analyse requirements and for modelling them in a systematic way. For many years, requirement writers used to write stories to specify the expected behaviour of the proposed software system and its interactions with the external world. Jacobson et al. (1999) formalized this story-writing approach into a more systematic and disciplined use case approach. They designed a UML for the software development of object-oriented systems. The use cases address only the functional requirements which explain the expectations of users sitting outside the system. Functional requirements express “what do we expect from the system” without bothering about “how it will be implemented”. The use cases capture the expectations in terms of achieving goals and interactions of the users with the system. Many persons are not able to differentiate amongst the use case, use case scenario and use case diagram, and use these terms interchangeably. However, all the three terms are different. Use cases are structured outlines or templates for the description of requirements, written in a natural language like English. A use case scenario is an instance of a use case. It represents a path through a use case. Use case diagrams are graphical representations that may be decomposed into further level of abstraction.

3.5.1 Use Cases and Actors

Use cases and actors are used to define the scope of the system we are planning to build. Use cases incorporate anything that is within the system, but actors include anything that is external to the system. We identify actors and use cases for the system under development. To do this, a proper understanding of the system is essential. A use case is a description of a system in terms of a sequence of actions.

A use case is initiated by an actor with a specific purpose in mind and completes successfully when that purpose is achieved. The use case describes the sequence of interactions between actors and the system to fulfil the desired purpose. It is nothing but a high-level piece of functionality that the system will provide. It includes a main sequence which a system will follow when an actor interacts with the system. It also includes possible alternative sequences which may occur depending on the interaction and input conditions. Fournier (2009) has rightly given his views about use cases as:

The real value of a use case is the dynamic relationship between the actor and the system. A well written use case clarifies how a system is used by the actor for a given goal or reason. If there are any questions about what a system does to provide some specific value to someone or something outside the system, including conditional behaviour and handling conditions of when something goes wrong, the use case is the place to find the answer.

A use case describes who (any user) does what (interaction) with the system, for what goal, without considering the internal details of the system. The use cases are written in a simple language which should be understandable to every stakeholder. There is no standard outline for a use case. However, a few templates are available in the literature. Requirements are captured in a systematic way and discipline the requirement writers to follow a specified format.

An actor is someone or something that interacts with the system and lies outside the system. The actor may be users of the system, customers, persons giving information to the system or any external system providing data. All stakeholders on the customer's site may be treated as actors if they wish to interact with the system. Hence, everything that is outside the system and wishes to interact with the system is known as an actor.

3.5.2 Identification of Actors

An actor interacts with the system with a particular purpose. The actor may be a person or external system. We should not confuse the actors with devices they use. Devices may help the actors to interact with the system and they are not the actors themselves. We use computers with the help of a keyboard, but the keyboard is not a user (actor), we are the user. Bittner and Spence (2003) have explained the concept as:

The purpose of devices is to support some required behaviors of the system, but devices do not define the requirements of the system. Often systems must produce a printed report of information that it contains. We may want to show printer as an actor that then forwards the report to the real actor. This is not correct, printer is not an actor, it is just a mechanism for conveying information.

We consider the LMS and identify the following actors:

- (i) Administrator
- (ii) Data entry operator
- (iii) Librarian
- (iv) Library staff
- (v) Faculty
- (vi) Student
- (vii) Employee

There are two types of actors—primary actors and secondary actors. Primary actors are the persons who will use the system. Customers are the primary actors for whom the system is built. Secondary actors are the persons who will maintain or monitor the system. For example, administrator is a secondary actor.

The identification of actors with their specified roles may define the scope of the system for every actor and expected actions. An actor may interact with one or more use cases depending on its defined role in the system.

3.5.3 Identification of Use Cases

Use cases describe the functionality of the system. To achieve that functionality, actors interact with the system with a defined purpose. An actor acts from the outside and provides some input(s) to the system. One or more output may be generated by the system after such interaction of actors. From the IRD, we create use cases for the system. Some guidelines for the creation of use cases are given as:

- (i) Every specified functionality should have a use case.
- (ii) A unique name is assigned to a use case. The name should be meaningful and should be able to indicate the purpose of a use case.
- (iii) One or more actors may interact with a use case.
- (iv) The use case should describe the sequence of actions.
- (v) The use case is initiated by an actor with a specified purpose.
- (vi) Role of actors must be defined clearly.
- (vii) The use case should represent a complete and meaningful flow of events.

As explained earlier, we should always remember that use cases describe who (actor) does what (interaction) with the system, for what goal, without considering the internal details of the system.

In the LMS, we may identify the following use cases from the IRD:

- Maintain book details
- Maintain student details
- Maintain faculty/employee details
- Maintain login details
- Issue book
- Return book
- Fine calculation
- Reserve book
- Query book
- Search book
- Report generation

3.5.4 Defining Relationships between Use Cases

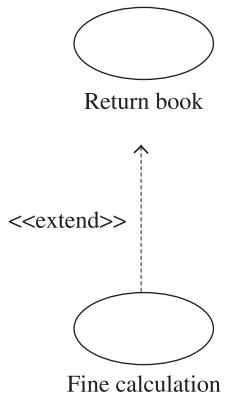
The use case diagram may model two kinds of relationships, namely, extend and include.

Extend Relationship

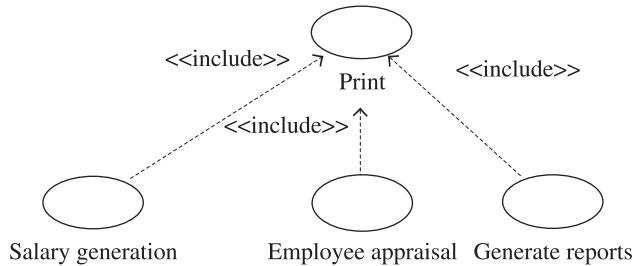
Extend relationship is used to model the occurrence of some optional, alternative or special part of the use case. This relationship is used to extend the functionality of the original use cases. The new use case is inserted into the original use case. The procedure when the extended use case is inserted into the original is as follows:

1. The original use case executes in usual manner to the point where the new use case has to be inserted.
2. At this point, the new use case is inserted and executed.
3. After the inserted use case completes, the original use case resumes again.

For example, in the LMS, the concept of extend can be observed when a student returns the book after the due date. The fine is calculated for late return of the book. We can represent fine calculation use case as a new use case that extends return book use case. The fine calculation use case is called whenever the book is returned after the due date by the student. The example of extend relationship for the LMS is shown in Figure 3.4.

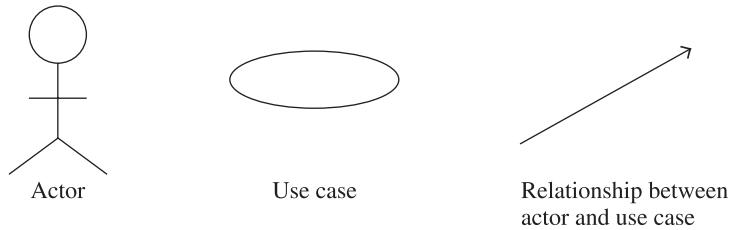
**Figure 3.4 Extend relationship.*****Include Relationship***

The redundant and repeated functionalities amongst use cases can be modelled into include relationship. Thus, the redundancies can be grouped into a single use case. In order to use include relationship, there must be common text in two or more use cases. As in the case of extend relationship, the new use case is inserted into the original use case. The procedure when the included use case is inserted into the original is the same as given in the extend relationship. For example, in employee management system, print use case is required by three use cases—salary generation, appraisal and generate reports (see Figure 3.5).

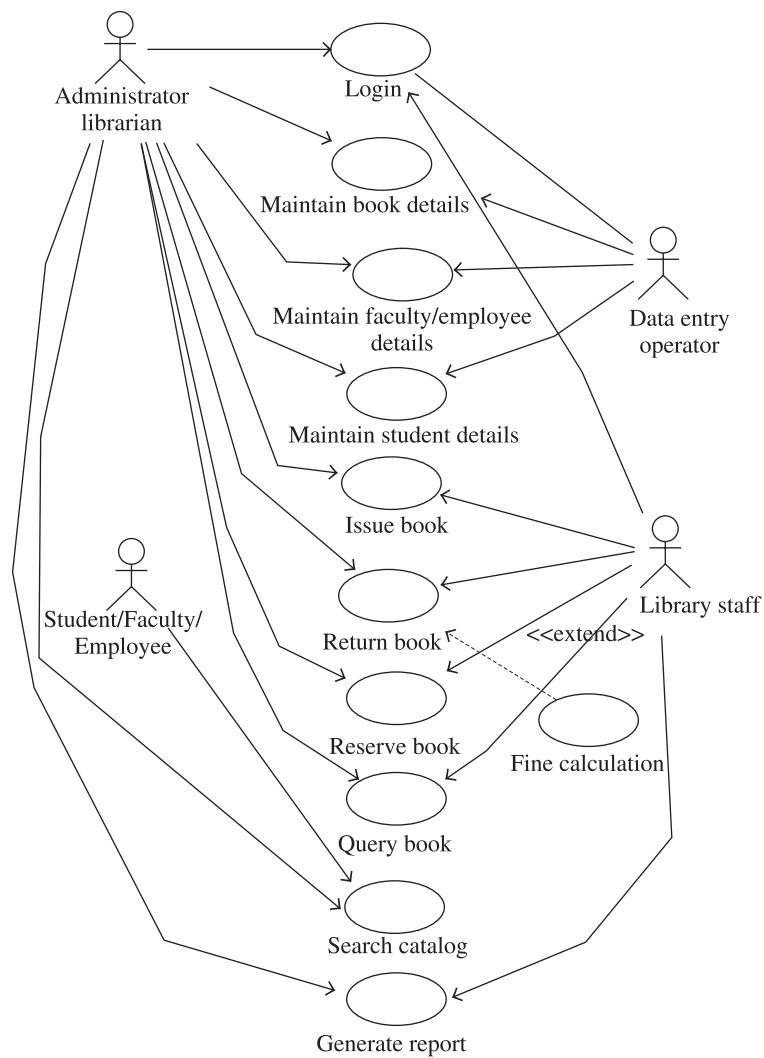
**Figure 3.5 Include relationship.****3.5.5 Use Case Diagram**

Use case diagram represents the top view of the system. The use cases reside within the system and the actors act from outside the system. The use case diagram is also used to present functionality of the system, but for proper explanation, it should read along with use cases. The use case diagram may also be decomposed into further level of abstraction. It also shows the relationship of use cases and actors. It also explains what happens when an actor interacts with the system. The components of the use case diagram are given in Figure 3.6.

An actor is represented by a stick figure (even non-human one) and a use case by an oval labelled with the name of the use case. The relationship between an actor and a use case is shown by an arrow. For small systems, one use case diagram may be sufficient to represent the

**Figure 3.6 Components of use case.**

whole system. However, for a large system, we may have to draw many use case diagrams to represent various portions of the system. The use case diagram of the LMS is given in Figure 3.7.

**Figure 3.7 Use case diagram for library management system.**

3.5.6 Use Case Description

Actors may interact with one or more use cases for specific purposes. Each actor gives some input to the system and expects the system to respond accordingly. All steps which are required to achieve the purpose constitute a flow. There are two types of flows:

1. *Basic flow:* It is the main flow and describes the sequence of events that takes place most of the time between actor and the system to achieve the purpose of the use case.
2. *Alternative flows:* If the basic flow is not successful due to any condition, the system takes an alternative flow. An alternative flow may occur due to failure of an expected service because of occurrence of exceptions/errors. There may be more than one alternative flow of a use case, which may not occur most of the time. Any alternative flow takes place under certain conditions in order to fulfil the purpose of a use case.

There is no standard template for writing use case description. Jacobson et al. (1999) have given a template for writing use cases which is given in Table 3.3. This captures the requirements in a disciplined and effective way and has become a popular format. Another similar template is also given in Table 3.4 which is also used by many software organizations (Cockburn, 2001; Quatrani, 2003).

Table 3.3 Jacobson's use case template

1.	Brief description: Describe a quick background of the use case.
2.	Actors: List the actors that interact and participate in this use case.
3.	Flow of events: 3.1. Basic flow: List the primary events that will occur when this use case is executed. 3.2. Alternative flows: Any subsidiary events that can occur in the use case should be separately listed. List each such event as an alternative flow. A use case can have as many alternative flows as required.
4.	Special requirements: Business rules for the basic and alternative flow should be listed as special requirements in the use case narration. These business rules will also be used for writing test cases. Both success and failure scenarios should be described here.
5.	Precondition: Preconditions that need to be satisfied for the use case to perform.
6.	Postcondition: Define the different states in which we expect the system to be in, after the use case executes.
7.	Extension points: List of related use cases, if any.

Table 3.4 Alternative use case template

1.	Introduction: Describe the brief purpose of the use case.
2.	Actors: List the actors that interact and participate in this use case.
3.	Precondition: Condition that need to be satisfied for the use case to execute.
4.	Postcondition: After the execution of the use case, different states of the systems are defined here.
5.	Flow of events: 5.1. Basic flow: List the primary events that will occur when this use case is executed. 5.2. Alternative flow: Any other possible flow in this use case, if there, should be separately listed. A use case may have many alternative flows.
6.	Special requirements: Business rules for the basic and alternative flows should be listed as special requirements. Both success and failure scenarios should be described.
7.	Associated use cases: List the related use cases, if any.

In the template, we have to list the actors that interact and participate in the use case. Preconditions, if any, are required to be specified for the use case to perform. If the precondition is not true, the use case cannot start its function. Similarly, postconditions are also important and are to be specified in the use case template. They define the different states in which we expect the system to be in, after the execution of the use case. Basic and alternative flows are required to be written step by step and with simple and short sentences. We should write the basic flow independent of the alternative flows and no knowledge of alternative flows is considered. The basic flow must be completed in itself without reference to the alternative flows. The alternative flow knows the details of when and where it is applicable which is opposite to the basic flow. It inserts into the basic flow when a particular condition is true (Bittner and Spence, 2003).

Special requirements are specified in the use case. This may include business rules for the basic and alternative flow execution. All possible conditions need to be specified. We also have to write related use cases which may give an idea about the use case relationships with other associated use cases. The issue book use case of the LMS is given in Table 3.5.

Table 3.5 Use case descriptions of issue book use case

Introduction: This use case documents the steps that must be followed in order to get a book issued.
Actors Administrator Library staff Librarian
Precondition: The administrator/library staff/librarian must be logged onto the system before the use case begins.
Postcondition: If the use case is successful, a book is issued to the student/faculty/employee and the database is updated, else the system state remains unchanged.
Event Flow Basic Flow <ol style="list-style-type: none"> 1. The student/faculty/employee membership number is read through the bar code reader. 2. The system displays information about the student/faculty/employee. 3. Book information is read through the bar code reader. 4. The book is issued for the specified number of days and the return date of the book is calculated and stamped on the book. 5. The book and student/faculty/employee information is saved into the database. Alternative Flow 1: Unauthorized student/faculty/employee If the system does not validate the student/faculty/employee membership number (due to membership expiry or any other reason), then an error message is flagged and the use case returns to the beginning of the basic flow. Alternative Flow 2: Account is full If the student/faculty/employee has requested a book and the account is full, i.e. he/she has already maximum number of books issued on his/her name, then the request for issue is denied and the use case ends. Alternative Flow 3: User exits This allows the user to exit at any time during the use case. The use case ends.
Special requirement None
Associated use case Login

EXAMPLE 3.2 Consider the case study of the LMS given in Section 3.1. Give use case description of return a book use case.

Solution The use case description of return book is given in Table 3.6.

Table 3.6 Use case description of return book use case

Introduction: This use case documents the steps that must be followed in order to return a book.
Actors Administrator Library staff
Precondition: The administrator/library staff must be logged onto the system before the use case begins.
Postcondition: If the use case is successful, the book is returned back to the library and if needed, the “fine calculation” use case is called, otherwise the system state is unchanged.
Event Flow Basic Flow <ol style="list-style-type: none"> 1. The book information is read from the bar code of the book through the bar code reader. 2. The student/faculty/employee detail on whose name the books were issued is displayed on the system. The date of issue and return is also displayed. 3. The administrator/library staff checks the stamp on the book to check the duration of issue of the book. 4. The database is updated and the book status is updated. 5. The date stamp on the book is cancelled.
Alternative Flows Alternative Flow 1: Late return of book If the duration for which the book has been kept by the student is more than 15 days, then a fine calculation use case is called. After the execution of fine calculation use case, the original use case resumes the basic flow. Alternative Flow 2: User exits This allows the user to exit at any time during the use case. The use case ends.
Special requirement None
Associated use case Login, fine calculation

3.5.7 Generation of Scenario Diagrams

A use case scenario is an instance of a use case. This is nothing but a path through a use case. A use case may have many paths. A basic flow is a path which is expected to be traversed most of the time and becomes a scenario of the use case. Every alternative path also generates a scenario. We may have various combinations of basic and/or alternative flows and every combination leads to a path in a use case. The basic and alternative flows for a use case are shown in Figure 3.8.

The basic flow is represented by a straight arrow and the alternative flows by the curves. Some alternative flows return to the basic flow, while others end the use case. Preconditions and postconditions are checked at start and end points of a use case, respectively. We consider the issue a book use case, and its basic and alternative flows are shown in Figure 3.9.

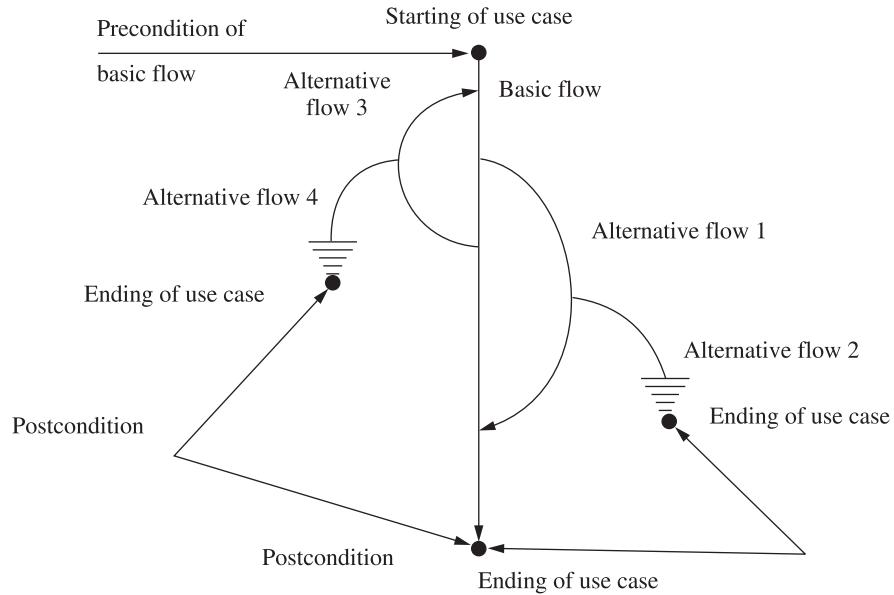


Figure 3.8 Basic and alternative flows with pre- and postconditions.

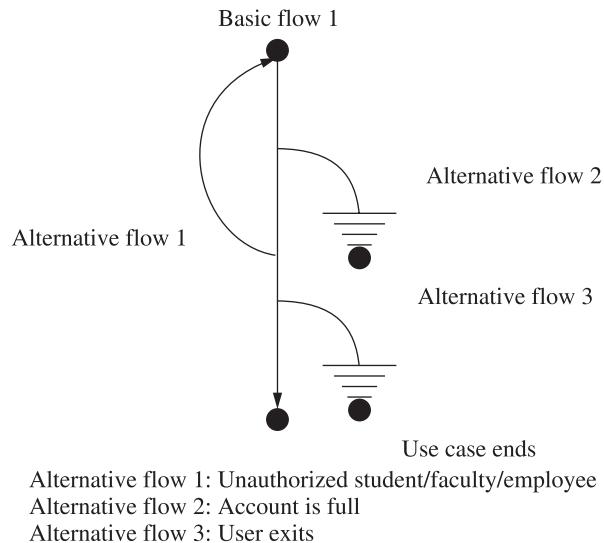


Figure 3.9 Basic and alternative flows for issue book use case.

3.5.8 Creation of Use Case Scenario Matrix

Many scenarios are generated using the use case scenario diagram due to basic flow(s), alternative flows and various combinations of basic and alternative flows. A scenario matrix represents all scenarios of the use case scenario diagram. The scenario matrix of the use case scenario diagram given in Figure 3.8 is given in Table 3.7.

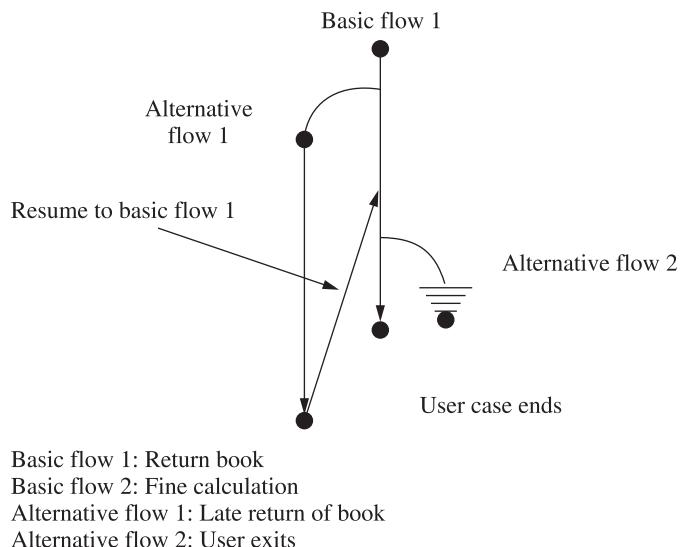
Table 3.7 Scenario matrix for the flow of events shown in Figure 3.8

Scenario 1	Basic flow			
Scenario 2	Basic flow	Alternative flow 1		
Scenario 3	Basic flow	Alternative flow 1	Alternative flow 2	
Scenario 4	Basic flow	Alternative flow 3		
Scenario 5	Basic flow	Alternative flow 3	Alternative flow 4	
Scenario 6	Basic flow	Alternative flow 3	Alternative flow 1	
Scenario 7	Basic flow	Alternative flow 3	Alternative flow 1	Alternative flow 2

The scenario matrix shows all instances of a use case. It represents all possible paths that may be generated in a use case. These instances (paths) may help us to generate test cases. After identification of inputs for a use case and its instances (using scenario matrix), test cases are generated. Hence, scenarios help us to generate test cases from the use cases.

EXAMPLE 3.3 Consider the use case of return book of the LMS case study given in Section 3.1. Draw the use case scenario diagram for it.

Solution Use case scenario diagram for the return book use case is given in Figure 3.10.

**Figure 3.10 Basic and alternative flows for return book use case.**

3.6 Characteristics of a Good Requirement

A requirement should be considered as a good requirement if it has the following characteristics:

- Correct
- Unambiguous

- Complete
- Consistent
- Verifiable
- Modifiable
- Clear (concise, terse, simple, precise)
- Feasible (realistic, possible)
- Necessary
- Understandable

Each of the above-mentioned characteristics is discussed here as per details given in IEEE (2001), Thayer and Dorfman (1997) and Young (2001).

3.6.1 Correct

If a requirement addresses a factual situation(s), then such situation(s) must be factually correct.

Requirement 1: A member of a library gets five books issued for one month.

There are different types of members in a library like faculty, students and employees. Hence, five books figure and issued duration may depend on the type of member. So, five books and one month figures are incorrect.

3.6.2 Unambiguous

The requirement should have only one interpretation. If the requirement is shown to many people, all should have the same interpretation. If there are more than one interpretation, then that requirement is probably ambiguous. Sometimes acronyms are used in a requirement and may become a reason for ambiguity. Consider the following requirement:

Requirement 2: Login ID shall be alphanumeric of length in the range of 4 to 15 characters. Special characters and blank spaces are not allowed.

There is an ambiguity in the above requirement; it is not clear what will the system do.

- (i) Do not allow the user to enter blank spaces or special characters.
- (ii) Display an error message if the user enters special characters or blank spaces.
- (iii) Automatically it will not include special characters and blank spaces.

Hence, the modified requirement shall be given as:

Requirement 2: Login ID shall be alphanumeric of length in the range of 4 to 15 characters. Special characters and blank spaces are not allowed. The system will not allow the user to enter blank spaces or special characters.

Requirements are written in natural languages (e.g. English) which are inherently ambiguous. Care should be taken, while writing requirements, in order to minimize the ambiguities.

3.6.3 Complete

A requirement should specify all conditions that may occur.

Requirement 3: The student shall access the LMS on the university's library LAN to search the availability of a book from the library.

What will happen to the other members of the library? Will they be able to search a book using the university's library LAN? If yes, the sentence should be as follows:

Requirement 3: The student/faculty/employee shall access the LMS on the university's library LAN to search the availability of a book from the library.

If we do not want to extend this facility to the other members (faculty/employee), then the sentence should be as follows:

Requirement 3: The student shall access the LMS on the university's library LAN to search the availability of a book from the library. However, the faculty and employees shall not access this search facility on the university's library LAN.

3.6.4 Consistent

There shall not be any conflicts between the requirements.

Requirement 4: New Password: Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are not allowed. However, special characters are allowed.

Requirement 5: Confirm Password: Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are allowed. However, special characters are not allowed.

There are inconsistencies between Requirement 4 and Requirement 5. Requirement 4 does not allow blank spaces and does allow special characters in password format, whereas Requirement 5 does allow blank spaces and does not allow special characters in the password format. Consistencies should be maintained amongst requirements and conflicting requirements must be corrected immediately.

3.6.5 Verifiable

A requirement should be verifiable. It should not use words like "works well", "good human interface" and "shall usually happen". Requirements using these words are not verifiable because the terms good, well and usually are not quantifiable. These terms have different meanings to different persons. Consider the following requirement:

Requirement 6: Every member should be able to quickly search a book using the university's library LAN.

The word "quickly" is not verifiable. It has different meanings for different persons. This requirement is not verifiable. The modified requirement may be given as:

Requirement 6: Every member should be able to search a book within 5 seconds using the university's library LAN.

Ambiguous words like robust, safe, accurate, flexible, reliable, safely, quickly, etc. should be avoided in framing requirements. These words are not verifiable and also make the sentence ambiguous.

3.6.6 Modifiable

A requirement is modifiable if changes can be made easily, completely and consistently while retaining its structure and style. The requirement should also be non-redundant. Redundancy itself is not an error, but it can easily lead to errors. If a change is made at one place and the other place is ignored, it leads to inconsistency of requirements. Whenever redundancy is necessary, it should be written in a table so that changes shall be made at all places.

3.6.7 Clear

A requirement should be written clearly, precisely and in simple short sentences. Consider the following requirement:

Requirement 7: Login ID should be of specified range of 4 to 15 and alphabets, numeric letters should be used in a login ID. Special characters should not be used however we should be able to differentiate between alphabets and special characters, blank spaces which are sometimes treated as special characters should also not be allowed.

A simple requirement may be written clearly as:

Requirement 7: Login ID: Alphanumeric of length in the range of 4 to 15 characters.
Special characters and blank spaces are not allowed.

Hence, sentences should be simple, short, clear, precise and unambiguous.

3.6.8 Feasible

A requirement shall be feasible if and only if it is implementable within given time, resources, money and technology.

Requirement 8: A library management system shall provide smart card with web-enabled services to every member. Digital versions of books shall be available to every member at any time and from any place using smart card.

This requirement is technically feasible but difficult to implement within short time and limited budget.

3.6.9 Necessary

A requirement is unnecessary if it is not requested by any stakeholder. Moreover, removing such a requirement should also not affect the present system.

This type of requirement may be added by developers with the assumption that it may be needed by stakeholders. If there is no use of such a requirement and if it is not important for the system, that requirement may be removed from the list.

3.6.10 Understandable

Requirements should be written in simple, short and clear sentences. Standard style and conventions should be used. The word “shall” should be used instead of “will”, “must” or “may”.

Sentences should also be grammatically correct and unambiguous. Difficult words should be avoided and their simple synonyms should be used.

3.7 Software Requirements Specification Document

After requirements elicitation, we prepare the IRD. This document gives an overview of the system. Now a detailed document is prepared on the basis of the IRD and is called *software requirements specification* (SRS) document. This document is used as a legal document which acts as a contract between customers and developers. On the basis of this document, the developers know what to build, and the customers know what to expect, and this document may also be used to validate that the built system satisfies the requirements. We use IEEE recommended practice for software requirements specifications—IEEE standard 830-1998 for preparing the SRS document.

3.7.1 Nature of the SRS Document

The following issues shall be addressed by the SRS writer:

1. *Functionality*: What functions the software is supposed to perform?
2. *External interfaces*: With what all external entities does the system interact? This may include the number of users, response time, recovery time, processing time, etc.
3. *Performance*: How does the software address performance issues? This may include people, hardware, external databases, etc.
4. *Quality attributes*: What are the considerations for non-functional requirements? These may include availability, correctness, maintainability, portability, reliability, security, testability, etc. These non-functional requirements should also be properly placed in the SRS document.
5. *Design constraints imposed on implementation*: All constraints should be highlighted which have an impact on implementation. This may include limitations of the operating environment, programming language, database, resources, policies for database integrity, etc.

The SRS writer(s) should not include design and implementation details. It should be written in a simple, clear and unambiguous language which may be understandable to all developers and customers.

3.7.2 Organization of the SRS Document

We have used IEEE recommended practice for SRS which is known as **IEEE Std. 830-1998**. The standard was approved on June 25, 1998. The document is used to specify requirements of the software which is to be developed. It helps the customers to know what they want from the proposed software system. It also helps the developer to understand exactly what the customer wants. There are four sections of the SRS document and are given in Table 3.8 (IEEE, 1998). The first two sections are the same for all projects, but Section 3 provides special tailoring as mentioned “specific requirements” for different projects.

Table 3.8 SRS document outline as per IEEE Std 830-1998

1.	Introduction 1.1 Purpose 1.2 Scope 1.3 Definitions, acronyms and abbreviations 1.4 References 1.5 Overview
2.	Overall description 2.1 Product perspective 2.1.1 System interfaces 2.1.2 User interfaces 2.1.3 Hardware interfaces 2.1.4 Software interfaces 2.1.5 Communications interfaces 2.1.6 Memory constraints 2.1.7 Operations 2.1.8 Site adaptation requirements 2.2 Product functions 2.3 User characteristics 2.4 Constraints 2.5 Assumptions and dependencies 2.6 Apportioning of requirements
3.	Specific requirements 3.1 External interfaces 3.2 Functions 3.3 Performance requirements 3.4 Logical database requirements 3.5 Design constraints 3.5.1 Standards compliance 3.6 Software system attributes 3.6.1 Reliability 3.6.2 Availability 3.6.3 Security 3.6.4 Maintainability 3.6.5 Portability 3.7 Organizing the specific requirements 3.7.1 System mode 3.7.2 User class 3.7.3 Objects 3.7.4 Feature 3.7.5 Stimulus 3.7.6 Response 3.7.7 Functional hierarchy 3.8 Additional comments
4.	Supporting information

These sections are discussed below along with their section numbers as in Table 3.8. The concepts are also explained with the help of the case study of the LMS. The complete SRS is given in the Appendix.

1. Introduction

This section gives the overview of the complete SRS document.

1.1 Purpose

We should describe the purpose of the SRS document and also list the intended audience for the document. The purpose of the LMS is given as:

1.1 Purpose

The library management system (LMS) maintains the information about various books available in the library. The LMS also maintains records of all the students, faculty and employees in the university. These records are maintained in order to provide membership to them.

1.2 Scope

The task of this subsection is to:

- (i) Assign a name to the software under development.
- (ii) List the functions which will be provided and also those functions which the software will not do.
- (iii) Explain the applications of the software along with possible benefits and objectives.
- (iv) Maintain consistency amongst statements.

The scope of the LMS is given as:

1.2 Scope

The name of the software is library management system (LMS). The system will be referred to as LMS in rest of the SRS. The proposed LMS must be able to perform the following functions:

Dos

1. Issue of login ID and password to system operators.
2. Maintain details of books available in the library.
3. Maintain details of the students in the university to provide student membership.
4. Maintain details of the faculty and employees in the university to provide faculty and employees membership.
5. Issue a book.
6. Bring a book back, and calculates fine if the book is being returned after the specified due date.
7. Reserve a book.
8. Provide search facility to check the availability of a book in the library.
9. Generate the following reports:
 - (a) List of books issued by a member.
 - (b) Details of books issued and returned on daily basis.
 - (c) Receipt of fine.

- (d) Total available books in the library.
- (e) List of library members along with issued books.
- (f) List of available books in the library:
 - Author-wise
 - Book title-wise
 - Publisher-wise
 - Subject-wise

Don'ts

1. Periodicals section is not covered.
2. Book bank (if any) is not covered.
3. Records of digital books are not available.
4. Purchase of books facility is not available.

Benefits

The LMS provides the following benefits:

1. Easy searching of books.
2. Efficient issue and return of books.
3. Accurate and automated fine calculation.
4. Printing of reports.

1.3 Definitions, Acronyms and Abbreviations

Define all terms, acronyms and abbreviations. This may help to make the SRS document more readable, understandable and clear to all the stakeholders. The definitions and acronyms used in the LMS are given as:

1.3 Definitions, Acronyms and Abbreviations

SRS: Software requirement specification

LMS: Library management system

System operator: System administrator, library staff, data entry operator

RAM: Random access memory

Accession number: It is a unique sequence number allocated to each book in the catalog.

Student: Any candidate admitted in a programme offered by a school.

System administrator/Administrator: User having all the privileges to operate the LMS.

Data entry operator (DEO): User having privileges to maintain book, student and faculty/employee details.

Library staff: User having privilege to issue, return, reserve and query books.

Faculty: Teaching staff of the university—Professor, Associate Professor, Assistant Professor

School: Academic unit that offers various programmes

1.4 References

This subsection provide a list of all documents including books and research papers referenced anywhere in the SRS document. The referenced material used in LMS is given as:

1.4 References

- (a) *Software Engineering* by K.K. Aggarwal & Yogesh Singh, New Age Publishing House, 3rd Edition, 2008.
- (b) IEEE Recommended Practice for Software Requirements Specifications— IEEE Std 830-1998.
- (c) IEEE Standard for Software Test Documentation—IEEE Std. 829-1998.

1.5 Overview

This subsection of the SRS gives the overview of the software system. Explain what the rest of the SRS contains and also gives details about the organization of the SRS document.

2. Overall Description

This section discusses the general factors which affect the requirements, and the final product specific requirements are not discussed. This section may help to write specific requirements of section 3 of the SRS document. The overview of the LMS is given as:

2. Overall Description

The LMS maintains records of books in the library and membership details of students, faculty and employees in the university. It is assumed that approved books have already been purchased by the acquisition section. It is also assumed that the student has already been admitted in the university for a specific programme. The system administrator will receive lists of the admitted students (school-wise and programme-wise) from the academic section. The establishment section will provide the list of the faculty and employees appointed in the university.

The LMS issues books to students, faculty and employees and brings the same back from them. The LMS generates fine if the book is returned by a student after the due date. It also allows students, faculty and employee to reserve a book. The student/faculty/employee can access LMS on the university's library LAN in order to search the availability of the book from the library.

The administrator/DEO will have to maintain the following information:

- Book details
- Student membership details
- Faculty and employee membership details

The administrator/library staff will perform the following functions:

- Issue a book
- Return a book
- Reserve a book
- Query a book

- Generate reports
 - ◆ List of books issued to a member
 - ◆ Details of books issued and returned on daily basis
 - ◆ Receipt of fine
 - ◆ Total available books in the library
 - ◆ List of library members along with issued books

The administrator/student/faculty/employee requires the following information from the system:

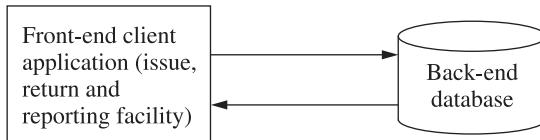
- List of available books in the library:
 - ◆ Author-wise
 - ◆ Book title-wise
 - ◆ Publisher-wise
 - ◆ Subject-wise

2.1 Product Perspective

This subsection puts the product into perspective with other related products. We should specify the environment which may be web-enabled, stand-alone system, or usage of local area network. A block diagram showing the major components of the large system, interconnections and external interfaces can be helpful. The product perspective of LMS is given as:

2.1 Product Perspective

The LMS shall be developed using client/server architecture and will be compatible with Microsoft Windows Operating System. The front-end of the system will be developed using Visual Basic 6.0 and the back-end will be developed using MS SQL Server 2005.



2.1.1 System Interfaces

System interfaces are addressed which are required by the product during its usage.

2.1.2 User Interfaces

This subsection should specify:

- (i) Logical characteristics of each interface between the product and its users.
- (ii) How the system can be best used by the user. A list of dos and don'ts for the user may also be specified.

The user interfaces of the LMS are given as:

2.1.2 User Interfaces

The LMS will have the following user-friendly and menu-driven interfaces:

- (a) **Login:** to allow the entry of only authorized users through valid login ID and password.
- (b) **Book details:** to maintain book details.
- (c) **Student membership details:** to maintain student membership details.
- (d) **Faculty/employee membership details:** to maintain faculty/employee membership details.
- (e) **Issue book:** to allow library staff to issue a book from the library.
- (f) **Return book:** to allow library staff to bring a book back to the library.
- (g) **Reserve book:** to allow library staff to reserve a book.
- (h) **Query book:** to check availability of a book in the library.

The software should generate the following information:

- (a) Details of books issued and returned on daily basis.
- (b) Details of books available in the library.
- (c) Details of books issued to a student.
- (d) List of library members with issued books.
- (e) Receipt of fine

2.1.3 Hardware Interfaces

Logical characteristics of each interface between the software product and the hardware component is to be specified. For example, issues such as screen resolution, support for printer, single user/multi-user system, etc. should be addressed. These issues for the LMS are shown as follows:

2.1.3 Hardware Interfaces

- (a) Screen resolution of at least 640×480 or above.
- (b) Support for printer (dot matrix, deskjet, laserjet).
- (c) Computer systems will be in the networked environment as it is a multi-user system.

2.1.4 Software Interfaces

This subsection should specify the interface of the software product with other software applications. For example, operating system, front end tools, back end tools, the specification and version number of the software, etc. should also be specified. The software interfaces for the LMS are given as:

2.1.4 Software Interfaces

- (a) MS-Windows Operating System (NT/XP/Vista)
- (b) Microsoft Visual Basic 6.0 for designing front-end
- (c) MS SQL Server 2005 for back-end

2.1.5 Communication Interfaces

This subsection specifies the various interfaces to communication such as local network, protocol, etc. For example, whether the software product will use LAN, web-enabled services or stand-alone system. In the LMS, communication is via local area network (LAN).

2.1.6 Memory Constraints

Any constraints related to primary and secondary memories are specified. For example, for the LMS, we may specify “At least 512 MB RAM and 500 MB space of hard disk will be required to run the software”.

2.1.7 Operations

Various operations in the user’s organization are specified. This may include data processing support functions, backup and recovery operations.

2.1.8 Site Adaptation Requirements

Any specific requirements related to the hardware and software interfaces at site are to be specified in this subsection. In the LMS, the terminal at the client site will have to support the hardware and software interfaces specified in sections 2.1.3 and 2.1.4, respectively.

2.2 Product Functions

This section should list the summary of all functions that the software shall perform. Functions should be listed in a simple and clear language and must be understandable to the customer. The product functions for the LMS are given as:

2.2 Product Functions

The LMS will allow access only to the authorized users with specific roles (system administrator, library staff, DEO and student). Depending upon the user’s role, he/she will be able to access only specific modules of the system.

A summary of major functions that the LMS shall perform includes:

- A login facility for enabling only authorized access to the system.
- The system administrator/DEO will be able to add, modify, delete or view book, student, faculty/employee and login information.
- The system administrator/library staff will be able to issue, bring back, and reserve book.
- The system administrator/library staff will be able to query a book in order to check the availability of the book in the library.
- The system administrator/students/faculty/employee will be able to search a book from the library catalogue (author-wise, title-wise and publisher-wise).
- The system administrator/library staff will be able to generate various reports from the LMS.

2.3 User Characteristics

Characteristics of users in terms of their qualification, skills sets and experience should be specified in this subsection. The user characteristics for the LMS are given as:

2.3 User Characteristics

Qualification: At least matriculation and comfortable with English.

Experience: Should be well versed/informed about the processes of the university library.

Technical experience: Elementary knowledge of computers.

2.4 Constraints

We may list constraints that may limit the developer's implementation options. These constraints may include:

- (i) Regulatory policies
- (ii) Hardware limitations
- (iii) Interface to other applications
- (iv) Parallel operations
- (v) Audit functions
- (vi) Control functions
- (vii) Higher-order language requirements
- (viii) Protocols
- (ix) Reliability requirements
- (x) Safety, security and criticality of the application

The constraints for the LMS are given as:

2.4 Constraints

- The software does not maintain records of periodicals.
- There will be only one administrator.
- The delete operation is available to the administrator and DEO. To reduce the complexity of the system, there is no check on delete operation. Hence, the administrator/DEO should be very careful before deletion of any record and he/she will be responsible for data consistency.
- The user will not be allowed to update the primary key.

2.5 Assumptions and Dependencies

This subsection includes assumptions and dependencies for the implementation of the software product. These assumptions may be very important and may become input for the design and implementation of the software system.

2.5 Assumptions and Dependencies

- The acquisition section of the library will provide the list of the books purchased by the library.
- The academic section will provide the list of the admitted students (school-wise and programme-wise).
- The establishment section will provide the list of the faculty members in various schools of the university. It will also provide the list of employees working in various branches such as examination, academics, establishment and schools of the university.
- The login ID and password must be created by the system administrator and communicated to the concerned user confidentially to avoid unauthorized access to the system.

2.6 Apportioning of Requirements

This subsection lists those requirements that can be delayed and may be incorporated in the future version of the software system. For the LMS, there is no such requirement.

3. Specific Requirements

All requirements stated in section 2 must be written in detail and at a level sufficient for the designers to understand and design the system. The testers should test that the requirements are met by the system. All requirements should conform to the guidelines of good requirements as discussed in Section 3.6. Some of the characteristics of good requirements are correctness, completeness, unambiguity, consistency, verifiability, modifiability, traceability, etc.

In addition to the basic format given by IEEE 830-1998, we may also construct Section 3 according to any one of the following additional templates given in Section 3.7 of the SRS.

System Mode

Depending on the mode of operation, we may choose either of the two outlines given in Tables 3.9 and 3.10.

Table 3.9 Template of SRS section 3 organized by mode

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Functional requirements
3.2.1 Mode 1
3.2.1.1 Functional requirement 1.1
.
.
.
3.2.1. <i>n</i> Functional requirement 1. <i>n</i>
3.2.2 Mode 2
.
.
.
3.2. <i>m</i> Mode <i>m</i>
3.2. <i>m</i> .1 Functional requirement <i>m</i> .1
.
.
.
3.2. <i>m</i> . <i>n</i> Functional requirement <i>m</i> . <i>n</i>
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

Table 3.10 Another template of SRS section 3 organized by mode

3. Specific requirements
3.1 Functional requirements
3.1.1 Mode 1
3.1.1.1 External interfaces
3.1.1.1.1 User interfaces
3.1.1.1.2 Hardware interfaces
3.1.1.1.3 Software interfaces
3.1.1.1.4 Communications interfaces
3.1.1.2 Functional requirements
3.1.1.2.1 Functional requirement 1
.
.
.
3.1.1.2. <i>n</i> Functional requirement <i>n</i>
3.1.1.3 Performance
3.1.2 Mode 2
.
.
.
3.1. <i>m</i> Mode <i>m</i>
3.2 Design constraints
3.3 Software system attributes
3.4 Other requirements

User Class

Some systems provide different sets of functions to different classes of users. The outline is given in Table 3.11.

Table 3.11 Template of SRS section 3 organized by user class

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Functional requirements
3.2.1 User class 1
3.2.1.1 Functional requirement 1.1
.
.
.
3.2.1. <i>n</i> Functional requirement 1. <i>n</i>

(Contd.)

Table 3.11 Template of SRS section 3 organized by user class (*Contd.*)

3.2.2 User class 2
.
.
3.2. <i>m</i> User class <i>m</i>
3.2. <i>m</i> .1 Functional requirement <i>m</i> .1
.
.
3.2. <i>m</i> . <i>n</i> Functional requirement <i>m</i> . <i>n</i>
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

Objects

The outline of arranging section 3 by objects is given in Table 3.12.

Table 3.12 Template of SRS section 3 organized by object

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Classes/Objects
3.2.1 Class/Object 1
3.2.1.1 Attributes (direct or inherited)
3.2.1.1.1 Attribute 1
.
.
3.2.1.1. <i>n</i> Attribute <i>n</i>
3.2.1.2 Functions (services, methods, direct or inherited)
3.2.1.2.1 Functional requirement 1.1
.
.
3.2.1.2. <i>m</i> Functional requirement 1. <i>m</i>
3.2.1.3 Messages (communications received or sent)
3.2.2 Class/Object 2
.
.
3.2. <i>p</i> Class/Object <i>p</i>
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

Feature

A feature is an externally desired service by the system that may require a sequence of inputs to affect the desired result. Each feature is generally described as a sequence of stimulus response pairs. The outline is given in Table 3.13.

Table 3.13 Template of SRS section 3 organized by feature

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 System features
3.2.1 System feature 1
3.2.1.1 Introduction/Purpose of feature
3.2.1.2 Stimulus/Response sequence
3.2.1.3 Associated functional requirements
3.2.1.3.1 Functional requirement 1
.
.
.
3.2.1.3. <i>n</i> Functional requirement <i>n</i>
3.2.2 System feature 2
.
.
.
3.2. <i>m</i> System feature <i>m</i>
.
.
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

Stimulus

Some systems are best organized by describing their functions in terms of stimuli. The outline is given in Table 3.14.

Table 3.14 Template of SRS section 3 organized by stimulus

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces

(Contd.)

Table 3.14 Template of SRS section 3 organized by stimulus (*Contd.*)

3.2 Functional requirements
3.2.1 Stimulus 1
3.2.1.1 Functional requirement 1.1
.
.
.
3.2.1.n Functional requirement 1.n
3.2.2 Stimulus 2
.
.
.
3.2.m Stimulus m
3.2.m.1 Functional requirement m.1
.
.
3.2.m.n Functional requirement m.n
3.3 Performance requirements
3.4 Design constraints
3.5 Software system attributes
3.6 Other requirements

Response

Some systems are best organized by describing their functions in support of the generation of a response. The outline is the same as given in Table 3.14 (stimulus is replaced by response).

Functional Hierarchy

Functions are organized in terms of their hierarchy with respect to either common inputs, common outputs or common internal data access. Data flow diagrams and data dictionaries may be used to show the relationships amongst the functions and data. The outline is given in Table 3.15.

Table 3.15 Template of SRS section 3 organized by functional hierarchy

3. Specific requirements
3.1 External interface requirements
3.1.1 User interfaces
3.1.2 Hardware interfaces
3.1.3 Software interfaces
3.1.4 Communications interfaces
3.2 Functional requirements
3.2.1 Information flows
3.2.1.1 Data flow diagram 1
3.2.1.1.1 Data entities
3.2.1.1.2 Pertinent processes
3.2.1.1.3 Topology

(*Contd.*)

Table 3.15 Template of SRS section 3 organized by functional hierarchy (*Contd.*)

	3.2.1.2 Data flow diagram 2
	3.2.1.2.1 Data entities
	3.2.1.2.2 Pertinent processes
	3.2.1.2.3 Topology
	.
	.
	.
	3.2.1. <i>n</i> Data flow diagram <i>n</i>
	3.2.1. <i>n</i> .1 Data entities
	3.2.1. <i>n</i> .2 Pertinent processes
	3.2.1. <i>n</i> .3 Topology
3.2.2	Process descriptions
	3.2.2.1 Process 1
	3.2.2.1.1 Input data entities
	3.2.2.1.2 Algorithm or formula of process
	3.2.2.1.3 Affected data entities
	3.2.2.2 Process 2
	3.2.2.2.1 Input data entities
	3.2.2.2.2 Algorithm or formula of process
	3.2.2.2.3 Affected data entities
	.
	.
	.
	3.2.2. <i>m</i> Process <i>m</i>
	3.2.2. <i>m</i> .1 Input data entities
	3.2.2. <i>m</i> .2 Algorithm or formula of process
	3.2.2. <i>m</i> .3 Affected data entities
3.2.3	Data construct specifications
	3.2.3.1 Construct 1
	3.2.3.1.1 Record type
	3.2.3.1.2 Constituent fields
	3.2.3.2 Construct 2
	3.2.3.2.1 Record type
	3.2.3.2.2 Constituent fields
	.
	.
	.
	3.2.3. <i>p</i> Construct <i>p</i>
	3.2.3. <i>p</i> .1 Record type
	3.2.3. <i>p</i> .2 Constituent fields
3.2.4	Data dictionary
	3.2.4.1 Data element 1
	3.2.4.1.1 Name
	3.2.4.1.2 Representation
	3.2.4.1.3 Units/Format
	3.2.4.1.4 Precision/Accuracy
	3.2.4.1.5 Range
	3.2.4.2 Data element 2
	3.2.4.2.1 Name
	3.2.4.2.2 Representation

(Contd.)

Table 3.15 Template of SRS section 3 organized by functional hierarchy (*Contd.*)

	3.2.4.2.3 Units/Format
	3.2.4.2.4 Precision/Accuracy
	3.2.4.2.5 Range
.	.
.	.
3.2.4.q	Data element q
	3.2.4.q.1 Name
	3.2.4.q.2 Representation
	3.2.4.q.3 Units/Format
	3.2.4.q.4 Precision/Accuracy
	3.2.4.q.5 Range
3.3	Performance requirements
3.4	Design constraints
3.5	Software system attributes
3.6	Other requirements

3.1 External Interfaces

External interfaces provide mechanisms to enter data into the system. Sometimes, we may also specify the outputs from the software system. We may use different types of forms for the entry of the data. A few forms of the LMS are given that are used to enter data into the LMS. Formats for various fields are also defined.

The specific requirements for issuing book in the LMS following the basic format of SRS given in IEEE Std 830-1998 are given as follows:

3. Specific Requirements

This section contains the software requirements in detail along with the various forms to be developed.

Issue Book

This form will be accessible only to the system administrator and library staff. It will allow him/her to issue existing book(s) to the student(s).

Issue Status:		
Accession No.	Title	Expected

Reservation Status:			
Accession No.	Title	Reservation Date	Expected

Various fields available on this form will be:

- **Membership ID:** Numeric and will have value from 100 to 19999. The system shall not allow the user to enter non-numeric characters and out of range values.
- **Accession number:** Numeric and will have value from 10 to 199999. The system shall not allow the user to enter non-numeric characters and out of range values.
- **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed. The system shall not allow the user to enter special characters and out of range values.
- Issue status (the following member information will be displayed):
 - ◆ **Accession number:** Numeric and will have value from 10 to 199999. The system shall not allow the user to enter non-numeric characters and out of range values.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed. The system shall not allow the user to enter special characters and out of range values.
 - ◆ **Expected:** Date of return. It will be of mm/dd/yyyy format and will have 10 alphanumeric characters. The system shall not allow the user to enter any other format.
- Reservation status (the following fields will be displayed):
 - ◆ **Accession number:** Numeric and will have value from 10 to 199999. The system shall not allow the user to enter non-numeric characters and out of range values.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed. The system shall not allow the user to enter special characters and out of range values.
 - ◆ **Reservation date:** It will be of mm/dd/yyyy format. It will have 10 alphanumeric characters. The system shall not allow the user to enter any other format.
 - ◆ **Expected:** Date of return. It will be of mm/dd/yyyy format and will have 10 alphanumeric characters. The system shall not allow the user to enter any other format.

3.2 Functions

Functional requirements define the fundamental actions that must take place in the software when inputs are accepted, and processed, and outputs are accordingly generated. Generally, the most popular way to implement this process is to write use case description. The use case description addresses all actions when an actor interacts with the system for a specified purpose. We may also include validity checks on inputs, sequencing information about operations, and error handling/response to abnormal situations. A few use case descriptions of LMS are given below in the specified template. The validity check and error handling information is also provided.

MAINTAIN BOOK DETAILS

A. Use Case Description

Introduction

This use case documents the steps that the administrator/DEO must follow in order to maintain book details and add, update, delete and view book information.

Actors

Administrator

Data entry operator

Precondition

The administrator/DEO must be logged onto the system before this use case begins.

Postcondition

If the use case is successful, then the book information is added, updated, deleted or viewed. Otherwise, the system state is unchanged.

Flow of Events

Basic Flow

This use case starts when the administrator/DEO wishes to add/update/delete/view book information.

1. The system requests that the administrator/DEO specify the function he/she would like to perform (either add a book, update a book, delete a book or view a book).
2. Once the administrator/DEO provides the requested information, one of the subflows is executed.
 - If the administrator/DEO selects “Add a Book”, the **Add a Book** subflow is executed.
 - If the administrator/DEO selects “Update a Book”, the **Update a Book** subflow is executed.
 - If the administrator/DEO selects “Delete a Book”, the **Delete a Book** subflow is executed.
 - If the administrator/DEO selects “View a Book”, the **View a Book** subflow is executed.

Basic Flow 1: Add a Book

The system requests that the administrator/DEO enter the book information. This includes:

- Accession number (book barcode ID)
- Subject descriptor
- ISBN
- Title
- Language
- Author
- Publisher

Once the administrator/DEO provides the requested information, the book is added to the system.

Basic Flow 2: Update a Book

1. The system requests that the administrator/DEO enter the accession number.

2. The administrator/DEO enters the accession number.
3. The system retrieves and displays the book information.
4. The administrator/DEO makes the desired changes to the book information. This includes any of the information specified in the **Add a Book** subflow.
5. Once the administrator/DEO updates the necessary information, the system updates the book information with the updated information.

Basic Flow 3: Delete a Book

1. The system requests that the administrator/DEO specify the accession number.
2. The administrator/DEO enters the accession number. The system retrieves and displays the required information.
3. The system prompts the administrator/DEO to confirm the deletion of the book record.
4. The administrator/DEO verifies the deletion.
5. The system deletes the record.

Basic Flow 4: View a Book

1. The system requests that the administrator/DEO specify the accession number.
2. The system retrieves and displays the book information.

Alternative Flows**Alternative Flow 1: Invalid Entry**

If in the **Add a Book** or **Update a Book** flow, the actor enters invalid accession number/ISBN/title/author/publisher/language/subject descriptor or leaves the accession number/ISBN/title/author/publisher/language/subject descriptor empty, the system displays an appropriate error message. The actor returns to the basic flow and may reenter the invalid entry.

Alternative Flow 2: Book Already Exists

If in the **Add a Book** flow, a book with a specified accession number already exists, the system displays an error message. The administrator returns to the basic flow and may reenter the book.

Alternative Flow 3: Book Not Found

If in the **Update a Book** or **Delete a Book** or **View a Book** flow, the book information with the specified accession number does not exist, the system displays an error message. The administrator returns to the basic flow and may reenter the accession number.

Alternative Flow 4: Update Cancelled

If in the **Update a Book** flow, the administrator/DEO decides not to update the book, the update is cancelled and the **Basic Flow** is restarted at the beginning.

Alternative Flow 5: Delete Cancelled

If in the **Delete a Book** flow, the administrator/DEO decides not to delete the book, the delete is cancelled and the **Basic Flow** is restarted at the beginning.

Alternative Flow 6: Deletion Not Allowed

If in the **Delete a Book** flow, issue/return/reserve details of the book selected exist, then the system displays an error message. The administrator returns to the basic flow and may reenter the student membership number.

Alternative Flow 7: User Exits

This allows the user to exit at any time during the use case. The use case ends.

Special Requirements
None
Associated Use Cases
Login
<p>B. Validity Checks</p> <ul style="list-style-type: none"> (i) Only the administrator/DEO will be authorized to access the Book Details module. (ii) Every book will have a unique accession number. (iii) Accession number cannot be blank. (iv) Accession number can only have value from 10 to 199999 digits. (v) Subject descriptor cannot be blank. (vi) ISBN number cannot be blank. (vii) Length of ISBN number for any user can only be equal to 11 digits. (viii) ISBN number will not accept alphabets, special characters and blank spaces. (ix) Title cannot be blank. (x) Length of title can be of 3 to 100 characters. (xi) Title will only accept alphabetic characters, brackets, numeric digits and blank spaces. (xii) Language cannot be blank. (xiii) Author (first name and last name) cannot be blank. <ul style="list-style-type: none"> (a) Length of first name and last name can be of 3 to 100 characters. (b) First name and last name will not accept special characters and numeric digits. (xiv) Publisher cannot be blank. (xv) Length of first name and last name can be of 3 to 300 characters. (xvi) Publisher will not accept special characters and numeric digits.
<p>C. Sequencing Information</p> <p>None</p> <p>D. Error Handling/Response to Abnormal Situations</p> <p>If any of the validation flows does not hold true, an appropriate error message will be prompted to the user for doing the needful.</p>

MAINTAIN STUDENT MEMBERSHIP DETAILS
A. Use Case Description
Introduction
This use case documents the steps that the Administrator/DEO must follow in order to maintain student membership details. This includes adding, updating, deleting and viewing student information.
Actors
Administrator
DEO
Precondition
The administrator/DEO must be logged onto the system before this use case begins.

Postcondition

If the use case is successful, then student information is added/updated/deleted/viewed from the system. Otherwise, the system state is unchanged.

Flow of Events**Basic Flow**

This use case starts when the administrator/DEO wishes to add, update, delete or view student information from the system.

1. The system requests that the administrator/DEO specify the function he/she would like to perform (either add a student, update a student record, delete a student record).
2. Once the administrator/DEO provides the requested information, one of the subflows is executed.
 - If the administrator/DEO selects “Add a Student”, the **Add a Student** subflow is executed.
 - If the administrator/DEO selects “Update a Student”, the **Update a Student** subflow is executed.
 - If the administrator/DEO selects “Delete a Student”, the **Delete a Student** subflow is executed.
 - If the administrator/DEO selects “View a Student”, the **View a Student** subflow is executed.

Basic Flow 1: Add a Student

The system requests that the administrator/DEO enter the user information. This includes:

- Membership number
- Photograph
- Roll No.
- Name
- School
- Programme
- Father’s name
- Date of birth
- Address
- Telephone
- Email
- Membership date
- Valid up to
- Password

Once the administrator/DEO provides the requested information, the system checks that student membership number is unique. The student is added to the system.

Basic Flow 2: Update a Student

1. The system requests that the administrator/DEO enter the student’s membership number.
2. The administrator/DEO enters the student’s membership number.

3. The system retrieves and displays the student's information.
4. The administrator/DEO makes the desired changes to the student information. This includes any of the information specified in the **Add a Student** subflow.
5. Once the administrator/DEO updates the necessary information, the system updates the student record with the updated information.

Basic Flow 3: Delete a Student

1. The system requests that the administrator/DEO specify the membership number of the student.
2. The administrator/DEO enters the membership number. The system retrieves and displays the student information.
3. The system prompts the administrator/DEO to confirm the deletion of the student record.
4. The administrator/DEO verifies the deletion.
5. The system deletes the record.

Basic Flow 4: View a Student

1. The system requests that the administrator/DEO specify the membership number.
2. The system retrieves and displays the student information.

Alternative Flows

Alternative Flow 1: Invalid Entry

If in the **Add a Student** or **Update a Student** flow, the actor enters invalid photograph/Roll No./Name/School/Programme/Father's name/Date of birth/Address/Telephone/Email/Membership date/Valid up to/Password or leaves the photograph/Roll No./Name/School/Programme/Father's name/Date of birth/Address/Telephone/Email/Membership date/Valid up to/Password empty, the system displays an appropriate error message. The actor returns to the basic flow and may reenter the invalid entry.

Alternative Flow 2: Student Already Exists

If in the **Add a Student** flow, a student with a specified membership number already exists, the system displays an error message. The administrator returns to the basic flow and may reenter the student information.

Alternative Flow 3: Student Not Found

If in the **Update a Student** or **Delete a Student** or **View a Student** flow, the student information with the specified membership number does not exist, the system displays an error message. The administrator returns to the basic flow and may reenter the membership number.

Alternative Flow 4: Update Cancelled

If in the **Update a Student** flow, the administrator decides not to update the student, the update is cancelled and the **Basic Flow** is restarted at the beginning.

Alternative Flow 5: Delete Cancelled

If in the **Delete a Student** flow, the administrator decides not to delete the student, the delete is cancelled and the **Basic Flow** is restarted at the beginning.

Alternative Flow 6: Deletion Not Allowed

If in the **Delete a Student** flow, issue/return/reserve details of the student selected exist, then the system displays an error message. The administrator returns to the basic flow and may reenter the membership number.

Alternative Flow 7: User Exits

This allows the user to exit at any time during the use case. The use case ends.

Special Requirements

None

Associated Use Cases

Login

B. Validity Checks

- (i) Only the administrator/DEO will be authorized to access the Student Membership Details module.
- (ii) Every student will have a unique membership number.
- (iii) Membership number can only have value from 100 to 5999 digits.
- (iv) Membership number will not accept alphabets, special characters and blank spaces.
- (v) Every student will have a unique membership number.
- (vi) Roll no. cannot be blank.
- (vii) Length of Roll no. for any user can only be equal to 11 digits.
- (viii) Roll no. cannot contain alphabets, special characters and blank spaces.
- (ix) Student name cannot be blank.
- (x) Length of student name can be of 3 to 50 characters.
- (xi) Student name will only accept alphabetic characters and blank spaces.
- (xii) School name cannot be blank.
- (xiii) Programme name cannot be blank.
- (xiv) Father's name cannot be blank.
- (xv) Father's name cannot include special characters and digits, but blank spaces are allowed.
- (xvi) Father's name can have length 3 to 50 characters.
- (xvii) Date of birth cannot be blank.
- (xviii) Address cannot be blank.
- (xix) Address can have length up to 10 to 200 characters.
- (xx) Phone cannot be blank.
- (xxi) Phone cannot include alphabets, special characters and blank spaces.
- (xxii) Phone can be up to 11 digits.
- (xxiii) Email cannot be blank.
- (xxiv) Email can have up to 50 characters.
- (xxv) Email should contain @ and . characters
- (xxvi) Email cannot include blank spaces.
- (xxvii) Password cannot be blank (initially autogenerated of 8 digits).

- (xxviii) Password can have length from 4 to 15 characters.
 (xxix) Alphabets, digits and hyphen, underscore characters are allowed in password field. However, blank spaces are not allowed.

C. Sequencing Information

None

D. Error Handling/Response to Abnormal Situations

If any of the validations/sequencing flow does not hold true, an appropriate error message will be prompted to the administrator for doing the needful.

3.3 Performance Requirements

Static and dynamic numerical requirements are addressed. Static numerical requirements may include:

- (i) The number of terminals to be supported.
- (ii) The number of simultaneous users to be supported.
- (iii) Amount and type of information to be handled.

Dynamic requirements may include the amount of data to be processed within certain time periods for both normal and peak load conditions, response time for transactions etc. We may specify “75% of transactions shall be processed in less than 2 seconds”. Hence, response time for all operations shall be specified. Broadly, it covers those requirements that affect the performance of the software system. For the LMS, we may specify performance requirements as:

3.3 Performance Requirements

- (a) Should support at least 8 terminals.
- (b) Should support at least 7 users simultaneously.
- (c) Should run on 500 MHz, 512 MB RAM machine.
- (d) Responses should be within 2 seconds.

3.4 Logical Database Requirements

We may specify logical requirements for any information that is to be placed into a database. We may include:

- (i) Frequency of use
- (ii) Accessing capabilities
- (iii) Data entities and their relationships
- (iv) Integrity constraints
- (v) Data retention policy

The logical database requirements for the LMS are given in Appendix.

3.5 Design Constraints

If there are any specific design constraints due to some standards, hardware and networking limitations, then those constraints should be specified to help the designers. This may include any deviation from standard (if any) report format, data naming, accounting procedures, audit tracing, etc.

3.6 Software System Attributes

Non-functional requirements such as reliability, availability, security, maintainability, usability, etc. are specified in this subsection. Definitions of some quality attributes are given in Table 3.16.

Table 3.16 Software quality attributes

S. No.	Attribute	Description
1	Reliability	The extent to which a software product performs its intended functions without failure.
2	Correctness	The extent to which the software meets its specifications.
3	Consistency and precision	The extent to which the software is consistent and gives results with precision.
4	Robustness	The extent to which the software tolerates the unexpected problems.
5	Simplicity	The extent to which the software is simple in its operations.
6	Traceability	The extent to which an error is traceable in order to fix it.
7	Usability	The extent of effort required to learn, operate and understand the functions of the software.
8	Accuracy	Meeting specifications with precision.
9	Clarity and accuracy of documentation	The extent to which documents are clearly and accurately written.
10	Conformity of operational environment	The extent to which the software is in conformity of operational environment.
11	Completeness	The extent to which the software has specified functions.
12	Efficiency	The amount of computing resources and code required by the software to perform a function.
13	Testability	The effort required to test the software to ensure that it performs its intended functions.
14	Maintainability	The effort required to locate and fix an error during maintenance phase.
15	Modularity	It is the extent of ease to implement, test, debug and maintain the software.
16	Readability	The extent to which the software is readable in order to understand.
17	Adaptability	The extent to which the software is adaptable to new platforms and technologies.
18	Modifiability	The effort required to modify the software during maintenance phase.
19	Expandability	The extent to which the software is expandable without undesirable side effects.
20	Portability	The effort required to transfer a program from one platform to another platform.

The quality considerations in the LMS are given as follows:

3.6 Software System Attributes

Usability

The application will be user-friendly and easy to operate, and the functions will be easily understandable.

Reliability

The applications will be available to the students throughout the registration period and have a high degree of fault tolerance.

Security

The application will be password protected. Users will have to enter correct login ID and password to access the application.

Maintainability

The application will be designed in a maintainable manner. It will be easy to incorporate new requirements in the individual modules.

Portability

The application will be easily portable on any windows-based system that has SQL Server installed.

IEEE Std 830-1998 has given various ways to organize section 3 and depending on the project, customer's expectations and developers expertise, we may choose one way for the organization of the SRS document.

3.7 Organizing the Specific Requirements

The details are given in the beginning of section 3.

3.8 Additional Comments

We may choose any one organization and the selection is dependent on the developer's expertise, previous experience, customer's expectations, available technologies, etc. Sometimes, more than one organization may also be used for organizing section 3 of the SRS document.

4. Supporting Information

Supporting information makes the SRS easier to use. It may include table of contents, index, appendixes, etc.

3.8 Requirements Change Management

Requirements change management is a new area which has been rapidly adopted by the software industry. As we all know, changes in requirements are inevitable. Many times, we make changes due to factors beyond our control. Every change is difficult and requires a systematic way to handle. Developers keep on getting requests for changes in the requirement before and also after release of the software system. Sometimes, requests for addition of new requirements come, which are equally difficult to handle and incorporate in the present set of requirements. Managing

change includes activities such as establishing a baseline, determining which dependencies are important to trace, establishing traceability between different items and change control (Rational Software, 2002).

Requirements change management is a systematic process of understanding and controlling changes in requirements. Any change in the requirements is not an independent activity and may have impact on other requirements, which make the area much more difficult and challenging.

The various steps for change management are given in the next subsections.

3.8.1 Is Change Necessary?

After receiving a change request, a study is carried out to know whether a change is necessary or not. Hence, all changes must be considered by a committee and action is taken on the merits of every change. The committee may not accept every change request.

3.8.2 Establishment of Baseline

After the approval of a change by the committee, it is communicated to all the stakeholders. They may give their views about the change and the same may be considered by the committee. Important views are noted and an appropriate action is recommended. If the change is finally approved, it is implemented and tested. Baseline is the tested version of a set of requirements representing a conceptual milestone, and serves as the basis for further development. A particular version of the software becomes the baseline when a responsible group decides to designate it as such (Aggarwal and Singh, 2008). Baselining is nothing but labelling a set of requirements at specific versions and freezing them before proceeding to the next phase of development. History of changes must be maintained. All documents should also be updated after incorporation of any change.

3.8.3 Requirements Traceability

Requirements traceability is a well-proven technique that extends from initial requirements to use cases and from there to design, implementation and testing. The ability to keep track of these relationships is a key to produce high-quality software development. It is particularly important in mission-critical projects. Requirements traceability is essential to ensure the correctness of each step of software development process. In IEEE (1998), traceability is defined as:

The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another, for example, the degree to which the requirements and design of a given software component match.

The purpose of traceability is to:

- Understand the origin of the requirements.
- Manage the change.
- Assess the impact of change in any requirement on the software system.
- Verify that all the requirements have been implemented.
- Verify the functionality of the software.

The traceability relationship can be established between any two elements in the project. Figure 3.11 shows the traceability relationship between two elements X and Y.

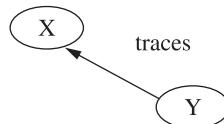


Figure 3.11 Requirements traceability.

A traceability relationship tells that if there is a change in one element (for example initial requirements), it may affect another element (for example use cases).

The use cases may be traced to the requirements in the IRD as shown in Figure 3.12.

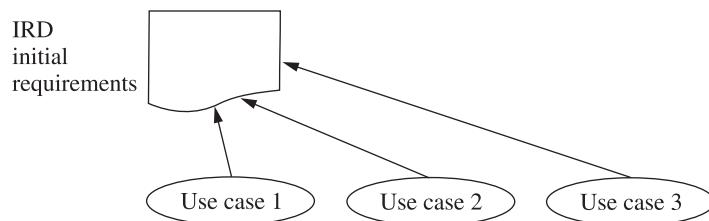


Figure 3.12 Use case traced to requirements.

The traceability relationship may also be represented by a traceability table or matrix. Table 3.17 shows the traceability matrix for representing relationships between initial requirements stated in IRD and use case. If an element X in a column fails to identify an element Y in a row, it means that extra functionality in the form of use case has been made that was not defined in IRD. Similarly, if an element X in a row does not correspond to an element Y in a column, it implies that no use case has been yet defined for initial requirements.

Table 3.17 Requirements traceability matrix

	Use case 1	Use case 2	Use case 3
Requirement 1			
Requirement 2			
Requirement 3			

For example, in the LMS, the IRD specifies:

The library staff shall be able to issue the books to each of its members.

The above requirement can be traced from use case “Issue book”:

This use case documents the steps that must be followed in order to get a book issued.

3.8.4 Change Control

A change in requirements is a regular activity. How do we manage and control such changes? There are two important issues; one is proper documentation of change and impact of the change. Change control activity is initiated with a change request form as given in Figure 3.13.

Change Request Form
(i) Title of the project:
(ii) Name of change request:
(iii) Date of change request:
(iv) Requested change description:
(v) Affected parts of source code:
(vi) Estimated cost of change:
(vii) Priority of change:
(viii) Assessment of change:
(ix) Date of submission of change request to change control authority (CCA):
(x) Decision of CCA with date:
(xi) Name of change implemented:
(xii) Date of implementation:
(xiii) Date of submission to QA:
(xiv) Decision of QA:
(xv) Submission to configuration team with date:
(xvi) Action of configuration management team:

Figure 3.13 Change request form.

The change request form documents the description of change. Its impact on other parts is also analysed and documented. Cost of change is estimated and priority is assigned. All these information is submitted to the change control authority (CCA) to take a decision about acceptance of change request. If decision of the CCA is in favour of accepting a change, then the CCA issues an approval of change. After this, the change is implemented. The revised software is handed over to the software quality assurance (SQA) team for revalidation and also to ensure that the change has not adversely affected other parts of the software. The modified software is handed over to the software configuration team and is incorporated in a new version of the system.

Review Questions

1. “The goal of requirements analysis is to develop a model of what the system will do.” Explain this statement with the help of the steps that a requirements analyst will follow throughout the analysis.
2. Who are the important stakeholders in software projects? Identify.
3. Discuss the key strategies in eliciting information about the user’s requirement.
4. Explain the importance of requirements. How many types of requirements are possible and why?

5. List out requirements elicitation techniques. Which one is most popular and why?
6. What is software prototyping? Explain with the help of an example.
7. A university wishes to develop a software system for the student result management of its M.Tech. programme. A problem statement is to be prepared for the software development company. The problem statement may give an overview of the existing system and broad expectations from the new software system.
8. What are the characteristics of a good software requirement specification (SRS) document?
9. Consider the problem of result preparation automation system of B.Tech. courses (or MCA program) of any university and design the following:
 - (i) Identification of use cases and actors
 - (ii) Use case diagram
10. Requirements analysis is unquestionably the most communication intensive step in the software engineering process. Why does the communication path frequently break down?
11. What is the purpose of extension and inclusion association between use cases? Explain with the help of an example.
12. Compare and contrast various requirements elicitation methods.
13. Which requirements elicitation is most popular? Explain.
14. Consider the following Transport Company Automation (TCA) software. A transport company requires to automate its various operations. The company has a fleet of vehicles. Currently, the company has the following vehicles:
Ambassadors: 10 Non-AC, 2 AC
Tata Sumo: 5 Non-AC, 5 AC
Maruti Omni: 10 Non-AC
Maruti Esteem: 10 AC
Mahindra Armada: 10 Non-AC

The company rents out vehicles to customers. When a customer requests for a car, the company lets them know what types of vehicles are available and the charges for each car. For every car, there is a per-hour charge and a per-kilometer charge. A car can be rented for a minimum of 4 hours. The amount chargeable to a customer is the maximum of (per-hour charge for the car times the number of hours used, and per-kilometer charge times the number of kilometers run) subject to a minimum amount decided by the charge for 4-hour use of the car. An AC vehicle of a particular category is charged 50% more than a non-AC vehicle of the same category. There is a charge of ₹ 150 for every night halt regardless of the type of the vehicle.

When a customer books a car, he has to deposit an advance amount. The customer also informs the company when he expects to return the car. When the car is returned, depending on the usage, either the customer is refunded some amount, or he has to pay additional amount to cover the cost incurred.

In addition to automating the above activities, the company wants to collect statistics about various types of vehicles such as average amount of money spent on repairs for the car, average demand, revenue earned by renting out the car, and fuel consumption of the car. Based on these statistics, the company may take a decision about which vehicles are more profitable. The statistics can also be used to decide the charge for different types of vehicles.

Draw the following using standard notations. If necessary, you can make suitable assumptions regarding the details of various features of TCA software, but you must clearly write down the assumptions you make.

- (i) Draw use case diagram for the TCA software.
- (ii) Write use case description of any two use cases.

15. What is software requirements specification (SRS)? List five desirable characteristics of a good SRS document. Discuss the relative advantages of formal requirement specifications. List the important issues which an SRS must address.
16. Write the use case description for Cash Withdrawal from an ATM machine
17. Discuss the role of use cases in object-oriented requirements analysis.
18. Consider the following bookshop automation system (BAS) software.

BAS should enable the shop clerk to enter the details of various books the shop deals with and change the inventory level of various books when new stocks arrive.

BAS should help the customers query whether a book is in stock. The users can query the availability to a book either by using the book's title or by using a partial name of the author. If a book is in stock, the exact number of copies available and the rack number in which the book is located should be displayed. If a book is not in the stock, the query for the books is used to increment a request field for the book. The manager can periodically view the request field of the books to roughly estimate the current demand for different out-of-stock books.

BAS should maintain the price of various books. As soon as a customer selects a book to purchase, the sales clerk would enter the ISBN number of the book. BAS should decrement the stock to reflect the book sale and generate the sales receipt for the book.

Upon request, BAS should generate sales statistics (viz. book name, publisher, ISBN number, number of copies sold and the sales revenue) for any period. BAS should enable the manager to view publisher-wise sales (e.g. sale of all books of any given publishing house) over a period.

A software system is required to be developed to maintain telephone directory and billing system. The software should be able to update/modify billing system and be able to maintain the directory of the subscribers.

- (i) Identify actors and use cases.
- (ii) Draw the use case diagram for the BAS software.
- (iii) Write the use case description of any two use cases.

19. The subscriber should be able to view the directory and search for a specific phone number according to the phone number, address and name.

A bill should be calculated on a monthly basis. Out of the total metered calls, 150 calls are free and rest all calls are called chargeable calls. The bill is calculated at ₹ 1.50 per call. The subscriber is given 10 days to pay the bill, if he fails to do so, he has to pay a fine of ₹ 500. The bill should show the status of the previous month's bill, that is, whether the last month bill was paid or not. The fine should be calculated if the bill is not paid within the due time (within 10 days from the date of issue of the bill). If the fine is paid within the due time, then no more action needs to be taken. But if the fine and the total payable amount is not paid within the due date, then that subscriber should be marked as not to be given any further services (only incoming is allowed). The subscriber can pay this amount at any time and reinitialize the services. If the total amount is not paid for 2 years, then the phone line should be disconnected.

A report is to be generated which analyses the total number of subscribers under an exchange area, revenue generated and total revenue generated for all the areas, on a monthly or yearly basis.

- (i) Identify actors and use cases.
- (ii) Draw the use case diagram for the above software.

20. Air ticket reservation (ATR) system is a software required to automate the reservation system for an airline at an airport or elsewhere. The primary task deals with the following functionalities:

- Maintenance of user account
- Maintenance of flight database, i.e. flight number, timing, number of seats, etc.
- Reservation of air tickets
- Cancelling of air tickets
- Enquiry about flight schedules, fares, timings, etc.
- Reports regarding day-to-day bookings, ticket availability, passengers database, etc.

Air ticket may be issued on demand for any flight to and from any destination for any date and in any class as desired by the passenger depending on the availability.

After any issue or cancellation of the ticket, the corresponding information has to be updated so that the ticket availability shows the status of the seats actually available/unavailable. Enquiries made by the passenger also need to be furnished disclosing the timing and availability of flight and air tickets in particular classes for the dates provided by the enquirer.

For the sale of an air ticket or for its cancellation, money will be paid or refunded and thus there will be a need for account maintenance, which may include ledger, sale book receipt, etc.

The system is expected to be able to help in the running of the air ticket reservation counter at an airport or elsewhere, thus helping in the overall functioning of any airlines government or private.

There will also be a detailed account of all the passenger list for all flights, thus aiding in cross checking any information that may be of use to the airline or any other authorities.

- (i) Identify actors and use cases.
- (ii) Draw the use case diagram for the ATR software.
- (iii) Write the use case description of reservation of airlines.

21. The proposed software product is the Hospital Patient Management System (HPMS). The system will be used to allocate beds to patients on a priority basis, and to assign doctors to patients in designated wards as need arises. Doctors will also use the system to keep track of the patients assigned to them. Nurses who are in direct contact with the patients will use the system to keep track of available beds, the patients in the different wards, and the types of medication required for each patient. The current system in use is a paper-based system. It is too slow and cannot provide updated lists of patients within a reasonable time frame. Doctors must make rounds to pick up patients' treatment cards in order to know whether they have cases to treat or not. The intentions of the system are to reduce over-time pay and increase the number of patients that can be treated accurately.
 - (i) Identify actors and use cases.
 - (ii) Draw the use case diagram for the above software.
 - (iii) Write the use case description of reservation of airlines.
22. Consider the case where Mr. A gives a call to Mr. B. Identify and discuss the use cases for the same.
23. Draw a use case diagram for e-shopping system.
24. A simple flight simulator is to be built/developed, using a bit-mapped display. Present a perspective view from the cockpit of a small airplane, periodically updated to reflect the motion of the plane. The world in which flights take place includes mountains, rivers, lakes, roads, bridges, a radio tower and of course a runway. Control inputs are from two joysticks. The left operates the radar and engine. The right one controls ailerons and elevator. Draw the use case diagram.
25. We can apply use case analysis as early as requirements analysis. Comment.
26. Draw a use case diagram for "Point-of-sale" terminal for the sale and payment of the customer.
27. Identify different use cases for a tea/coffee/soup vending machine.
28. What is a use case model? Why is the use case modelling useful in analysis?
29. Describe the use case-driven approach by Ivar Jacobson.
30. What is the significance of requirements change management in software development? Why is requirements traceability becoming important? Explain the change control process.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. The hardest part of software development is:

(a) Requirement gathering	(b) Software design
(c) Software implementation	(d) None of the above

- 2.** Requirements are described as:

 - (a) How of a system
 - (b) What of a system
 - (c) When of a system
 - (d) All of the above

3. Which is **not** a stakeholder?

 - (a) Customer
 - (b) User
 - (c) Developer
 - (d) Operating system

4. Which of the following is **not** a non-functional requirement?

 - (a) Correctness
 - (b) Functionality
 - (c) Reliability
 - (d) Portability

5. Requirements elicitation means:

 - (a) Requirements capturing
 - (b) Requirements prioritization
 - (c) Requirements management
 - (d) Requirements traceability

6. The best reason to select a requirements elicitation technique is:

 - (a) It is the only technique we know
 - (b) It is our favourite technique
 - (c) We believe that a particular technique is suitable for our project
 - (d) None of the above

7. The structured interview method includes the creation of:

 - (a) Use case diagram
 - (b) Data flow diagram
 - (c) Prototype
 - (d) Questionnaire

8. The size of a group in brainstorming session is:

 - (a) 15–30 persons
 - (b) 5–15 persons
 - (c) 1–5 persons
 - (d) >30 persons

9. The role of a facilitator is to:

 - (a) Handle conflicts
 - (b) Ensure smooth conduct of meeting
 - (c) Prepare detailed report
 - (d) All of the above

10. FAST stands for:

 - (a) Frequent application specification technique
 - (b) Facilitated application specification technique
 - (c) Facilitated approximate specification technique
 - (d) Facilitated application specification technology

11. Which one of the following is true?

 - (a) FAST is not a popular technique
 - (b) FAST session is conducted at customer's site
 - (c) FAST is more formal than brainstorming session
 - (d) FAST is a low-cost activity

12. Prototyping is:

 - (a) Less costly
 - (b) A process that may increase the cost of the system
 - (c) A complex version of the system
 - (d) A process of developing source code

13. A use case scenario is:
(a) An instance of a class
(c) An instance of an actor
(b) An instance of a use case
(d) A path through a process
14. A use case addresses:
(a) Functional requirements
(c) Both (a) and (b)
(b) Non-functional requirements
(d) None of the above
15. A use case diagram consists of:
(a) Actors and classes
(c) Classes and use cases
(b) Actors and objects
(d) Actors and use cases
16. Actors include anything that is:
(a) External to the system
(c) Functionality of the system
(b) Internal to the system
(d) Both (a) and (b)
17. Which one of the following is **not** an actor?
(a) External system
(c) Users
(b) Customers
(d) Keyboard
18. In a use case diagram, a use case is represented by:
(a) A triangle
(c) An oval
(b) A rectangle
(d) A square
19. In a use case diagram, actors are represented by:
(a) Triangles
(c) Ovals
(b) Stick figures
(d) Squares
20. The POPULAR template for writing use case description is given by:
(a) I. Jacobson
(c) B. Boehm
(b) V. Basili
(d) J. Rumbaugh
21. A requirement should be:
(a) Correct
(c) Verifiable
(b) Unambiguous
(d) All of the above
22. IRD stands for:
(a) Initial Request Document
(b) Initial Required Document
(c) Initial Requirements Document
(d) Interactive Requirements Document
23. The IEEE recommended practice for SRS is:
(a) IEEE std. 829–1998
(c) IEEE std. 830–1999
(b) IEEE std. 1026–1998
(d) IEEE std. 830–1998
24. The IEEE std. 830–1998 was approved on:
(a) August 25, 1998
(c) June 25, 1998
(b) July 30, 1998
(d) June 25, 1999
25. Which one is **not** a characteristic of a good requirement?
(a) Detailed
(c) Verifiable
(b) Complete
(d) Unambiguous

- 26.** Performance requirements do **not** include:
- (a) Number of terminals support
 - (b) Data entities and their relations
 - (c) Number of simultaneous users
 - (d) Type of information handled
- 27.** A popular object-oriented technique for representing requirements is:
- (a) Data flow diagram
 - (b) Flow chart
 - (c) Questionnaire
 - (d) Use case approach
- 28.** User characteristics exclude:
- (a) Qualification
 - (b) Technical knowledge
 - (c) Hardware limitations
 - (d) Experience
- 29.** The purpose of requirements traceability is to:
- (a) Understand origin of requirements
 - (b) Assess impact of change
 - (c) Identify requirements
 - (d) Verify functionalities
- 30.** CCA stands for:
- (a) Change committee authority
 - (b) Change control audit
 - (c) Change control arrangement
 - (d) Change control authority
- 31.** Which is **not** a component of a use case diagram?
- (a) Actor
 - (b) Use case
 - (c) Relationship between actor and use case
 - (d) Test case
- 32.** Which are **not** included in a use case template?
- (a) Actors
 - (b) Preconditions and postconditions
 - (c) Test cases
 - (d) Flow of events
- 33.** Every use case may have:
- (a) At least one actor
 - (b) At most one actor
 - (c) No actor
 - (d) None of the above
- 34.** A use case scenario may generate:
- (a) At most one test case
 - (b) At least one test case
 - (c) No test case
 - (d) None of the above
- 35.** Use cases and use case diagrams are used to define:
- (a) Complexity of a system
 - (b) Criticality of a system
 - (c) Stability of a system
 - (d) Behaviour of a system

Further Reading

Jacobson provides a classic introduction on use case approach in:

Jacobson, I.V., *Object Oriented Software Engineering: A Use Case Driven Approach*.
New Delhi: Pearson Education, 1999.

Cockburn provides guidance for writing and managing use cases. This may help to reduce common problem associated with use cases:

Cockburn, A., *Writing Effective Use Cases*. New Delhi: Pearson Education, 2001.

Hurlbut's paper provides a survey on approaches for formalizing and writing use cases:

Hurlbut, R., *A Survey of Approaches for Describing and Formalizing Use-Cases*, Technical Report 97-03, Department of Computer Science. Illinois Institute of Technology, USA, 1997.

Fournier has discussed the relationship between actor and system in:

Fournier, G., *Essential Software Testing—A Use Case Driven Approach*. New York CRC Press, 2009.

Other useful texts are available at:

Booch, G., Rumbaugh, J. and Jacobson, I.V., *The Unified Modeling Language User Guide*. Boston: Addison-Wesley, 1999.

Rational Requisite Pro, *User's Guide*, Rational Software Corporation, 2003.

Rational Rose User's Guide. IBM Corporation, 2003.

Tague, N.R., *The Quality Toolbox*. Milwaukee, WI: ASQ Quality Press, 2004.

The most current information about UML can be found at:

<http://www.rational.com>

<http://www.omg.org>

4

Object-Oriented Software Estimation

After the requirements are gathered, the customer may like to know the cost and time estimates of the project. In order to determine the cost and time of a project, estimates of size and effort are required. Providing accurate estimates for any software project is an important activity. Unfortunately, determining estimates in the early phases of software development is difficult due to lack of information available about the system at such an early stage. However, early estimates are highly desirable in order to determine the feasibility of a project or to bid for a tender. The importance of realistic estimates can be best explained by an example.

Suppose we want to construct our home and we have received many quotations mostly around ₹ 30 lakh. We select a builder who offers to do this job in 6 months for ₹ 25 lakh. The contract is signed and the builder begins the work. After about 2 months, the builder comes and demands ₹ 4 lakh extra and another 2 months for the same job. This is a real problem for us as we can neither arrange for more loan nor wait for 2 more months. After a great deal of discussion, we were forced to stop the construction. This was a great loss to us and to the builder as well. The real problem was that the builder made unrealistic or probably poor estimates. The above is a classic example of **lack of estimation capabilities of the builder**. Thus, providing realistic estimates is essential to project planning process. By using estimations, the project can be measured and controlled. In order to conduct effective software estimates, we must identify:

- Scope/boundaries of the project
- Size of the project
- Effort of the project
- Resources required in the project
- Risk involved in the project

In this chapter, the object-oriented methods for estimating size and effort are presented. These methods can be used in requirement and design phase of software development life cycle in order to estimate effort. The process of risk management is also presented so that uncertain and critical activities can be identified in the early phases of software development.

4.1 Need of Object-Oriented Software Estimation

Traditional software estimations include lines of source code and function point analysis for size estimation: COCOMO 81, COCOMO II and Putnam resource allocation model for cost estimation. However, as the paradigm is shifting towards object-oriented software, we wonder “Is object-oriented software estimation different than traditional software estimation?” We would say it is a similar activity but the key parameters (for example size) to do an estimation change in the case of object-oriented software. Object-oriented software engineering uses unified modelling language for creating models. As discussed in Chapter 3, one of the important mechanisms to depict the functionality of the system is use cases. The use case diagram may be used to predict the size and hence the effort of the software at an early stage of software development. Classes are also an important element for measuring size in object-oriented software. Figure 4.1 shows the framework for estimating an object-oriented software.

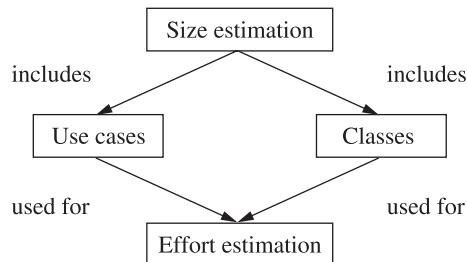


Figure 4.1 Framework of object-oriented software estimation.

The functionality of an object-oriented software can be depicted using use cases and these use cases are transformed using classes. Thus, the effort can be estimated using the size measures computed for object-oriented products.

4.2 Lorenz and Kidd Estimation Method

The Lorenz and Kidd method for estimation of size and effort is one of the earliest methods developed in 1994 (Lorenz and Kidd, 1994). The results of this method are based on the study of 5–8 projects developed in C++ and Smalltalk. Lorenz and Kidd provided two methods for estimation of a number of classes:

1. *Use of scenario scripts:* The number of scenario scripts (same as use cases) may be used to estimate size of classes.

$$\text{Number of classes} = 17 \times \text{number of scenario scripts}$$

This method for determining the number of classes can be used in early phases of software development life cycle, i.e. before the classes have been identified.

2. *Use of key and support classes:* After elaborating the design phase, the number of classes can be determined. Lorenz and Kidd differentiated between key classes and support classes. Key classes are specific to business applications and are ranked with higher priority by the customer. These classes also involve many scenarios. Support classes are common for many applications. These classes include user interface, back end classes and communications.

The following are used to determine the support classes:

Interface type	Multiplier
No GUI	2.0
Text-based user interface	2.25
Graphical user interface	2.5
Complex graphical user interface	3.0

Support classes can be calculated as:

$$\text{Support classes} = \text{Number of key classes} \times \text{Multiplier}$$

Finally, the total number of classes is obtained by adding key classes and support classes.

$$\text{Total number of classes} = \text{Key classes} + \text{Support classes}$$

According to Lorenz and Kidd, each class requires 10 to 20 person days for implementation. Thus, effort can be calculated as:

$$\text{Effort} = \text{Total number of classes} \times (10 \text{ to } 20 \text{ person days})$$

This method is easy and simple to understand.

EXAMPLE 4.1 An application consists of 15 scenario scripts and requires 15 person days to implement each class. Determine the effort of the given application.

Solution Number of classes = $17 \times \text{Scenario scripts}$

$$= 17 \times 15 = 255$$

$$\text{Effort} = 255 \times 15 = 3825 \text{ person days}$$

EXAMPLE 4.2 Consider the database application project with the following characteristics:

1. The application has 45 key classes
2. A graphical user interface is required

Calculate the effort to develop such a project given 20 person days.

Solution Number of key classes = 45

$$\begin{aligned} \text{Number of support classes} &= \text{Number of key classes} \times \text{Multiplier} \\ &= 45 \times 2.5 \\ &= 112.5 \end{aligned}$$

$$\begin{aligned} \text{Total number of classes} &= \text{Number of key classes} + \text{Number of support classes} \\ &= 112.5 + 45 \\ &= 157.5 \end{aligned}$$

$$\begin{aligned}\text{Effort} &= 157.5 \times 20 \\ &= 3150 \text{ person days}\end{aligned}$$

4.3 Use Case Points Method

The use case points method was developed by Gaustav Karner of Objectory (currently known as IBM Rational Software) in 1993 (Karner, 1993). The method is used for estimating size and effort of object-oriented projects using use cases. The method is an extension of function point analysis technique developed by Albrecht and Gaffney (1983). Karner's work on use case points was written in his diploma thesis titled "Metrics Objectory". The use case points method measures the functionality of the software based on the use case counts. The use case model is a very popular technique for requirements gathering and can be used at early phases of software development in order to provide estimations of the project. Figure 4.2 shows the steps followed

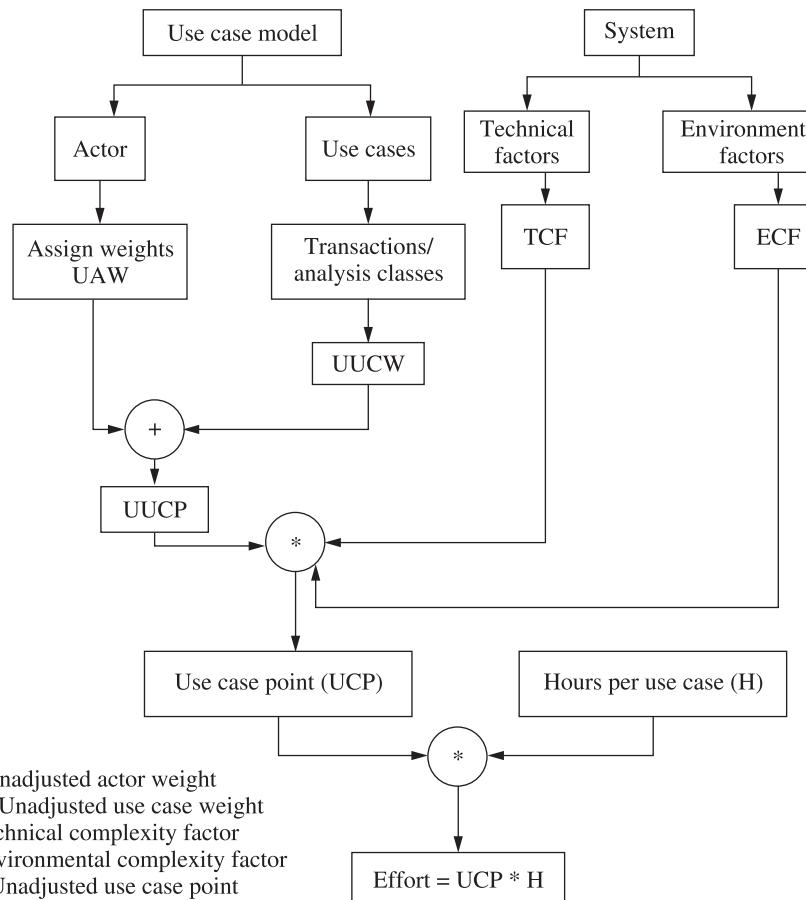


Figure 4.2 Use case points method.

to calculate use case points using the use case points method. Thus, the following steps are used in the use case points method:

1. ***Classification of actors and use cases:*** In this step, the complexity levels of the actors and use cases are identified and their weighted sum is computed.
2. ***Computing unadjusted use case points:*** The estimates of unadjusted use case points are made.
3. ***Calculating technical complexity factors:*** In this step, we identify the degree of influence of technical factors on the project.
4. ***Calculating environmental complexity factors:*** In this step, the environmental complexity factors are classified.
5. ***Calculating use case points:*** In the final step, the use case points are calculated on the basis of values obtained from steps 1, 2 and 3.

4.3.1 Classification of Actors and Use Cases

The first step involved in the calculation of use case points is the classification of actors and use cases. The actors are ranked according to their complexity, i.e. simple, average or complex. The criteria for classifying an actor as simple, average or complex are given in Table 4.1.

Table 4.1 Actors with weighting factors

Actor complexity	Description	Weighting factor
Simple	Represents another system with a defined application programming interface (API).	1
Average	Represents interaction with another system through a protocol such as TCP/IP or a human interaction with a line terminal.	2
Complex	Represents interaction through a graphical user interface.	3

The unadjusted actor weight (UAW) is the weighted sum of actors. Mathematically,

$$\text{UAW} = \sum_{i=1}^n a_i \times w_i$$

where n is the number of actors, a_i is the i th actor and w_i is the value of weighting factor of the i th actor.

The use cases are ranked according to their complexity: simple, average or complex. The criteria for classifying a use case are given in Table 4.2. There are two methods for classifying use case complexity:

1. A use case may be classified as simple, average or complex based on the number of transactions. A transaction can be defined as a collection of activities that are counted by counting the use case steps.
2. The other method to classify the use case is counting the analysis objects which are counted by determining the number of objects that we will need to implement a use case.

Table 4.2 Use cases with weighting factors

Use case complexity	Description	Weighting factor
Simple	Number of transactions ≤ 3 Analysis objects ≤ 5	5
Average	Number of transactions > 3 and < 7 Analysis objects > 5 and < 10	10
Complex	Number of transactions ≥ 7 Analysis objects ≥ 10	15

Each use case is multiplied by its corresponding weighting factors and this sum is added to get unadjusted use case weight (UUCW). Mathematically,

$$\text{UUCW} = \sum_{i=1}^m u_i \times w_i$$

where m is the number of use cases and u_i is the i th use case.

4.3.2 Computing Unadjusted Use Case Points

After classifying actors and use cases, the resultant unadjusted use case points (UUCP) are computed by adding UAW and UUCW as shown in mathematical form below:

$$\text{UUCP} = \text{UAW} + \text{UUCW}$$

The procedure for calculating UUCP is given in Table 4.3.

Table 4.3 Calculating unadjusted use case points

Units	Complexity count	Totals
Actor	<input type="text"/> × weighting factor <input type="text"/> × weighting factor	<input type="text"/>
Use cases	<input type="text"/> × weighting factor <input type="text"/> × weighting factor	<input type="text"/>
Total unadjusted use case points		<input type="text"/>

4.3.3 Calculating Technical Complexity Factors

Technical complexity factor (TCF) assesses the functionality of the software. TCFs are similar to the ones in function point analysis except some factors are added and some deleted. The criterion for technical factor is defined by Symons (1988) as:

A system requirement other than those concerned with information content intrinsic to and affecting the size of the task, but not arising from the project environment.

TCFs vary depending on the difficulty level of the system. Each factor is rated on the scale of 0 to 5 as shown in Figure 4.3. Table 4.4 shows the weights assigned to the contributing technical factors.

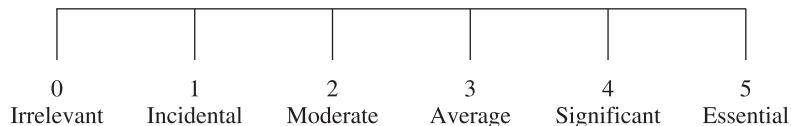


Figure 4.3 Measurement scale for TCF.

Table 4.4 Weighting factors for TCF

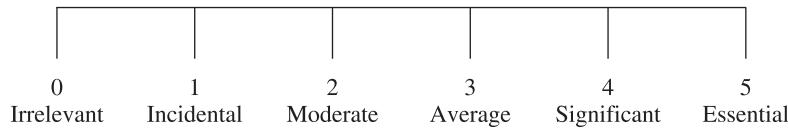
Factor (F_i)	Contributing factors	w_i
1	Distributed systems	2
2	Application performance, objectives in either response or throughput	1
3	End user efficiency	1
4	Complex internal processing	1
5	Reusability of source code	1
6	Installation ease	0.5
7	Ease of usability	0.5
8	Portability	2
9	Ease to change	1
10	Concurrency	1
11	Special security issues	1
12	Direct access for third parties	1
13	Special customer training provided	1

TCF is obtained by using the following relationship:

$$\text{TCF} = 0.6 + 0.01 \sum_{i=1}^{13} F_i \times w_i$$

4.3.4 Calculating Environmental Complexity Factors

Environmental complexity factor (ECF) helps in estimating the efficiency of the project. This factor is calculated based on the early estimations in the project based on the interviews carried out in objectory projects. Figure 4.4 classifies the ECF on the measurement scale from 0 to 5, 0 being irrelevant factor and 5 being essential factor. Table 4.5 shows the weights assigned to the contributing environmental factors.

**Figure 4.4 Measurement scale for ECF.****Table 4.5** Weighting factors for ECF

Factor (F_i)	Contributing factors	w_i
1	Familiarity with objectory/process	1.5
2	Application experience	0.5
3	Analyst capability	0.5
4	Object-oriented experience	1
5	Motivation	1
6	Part-time workers	-1
7	Programming language difficulty	-1
8	Stable requirements	2

The ECF is obtained by using the following relationship:

$$\text{ECF} = 1.4 + (-0.03) \sum_{i=1}^8 F_i \times w_i$$

4.3.5 Calculating Use Case Points

The final number of use case points is calculated by multiplying UUCP by TCF and ECF.

$$\text{UCP} = \text{UUCP} \times \text{TCF} \times \text{ECF}$$

Duration can be measured 20 person hours per use case point.

The use case points method can be used in early phases of software development in order to estimate the size and effort.

EXAMPLE 4.3 Consider an airline reservation system where the following information is available:

Number of actors: 05

Number of use cases: 10

Assume all complexity factors are average. Compute the use case points for the project.

Solution

$$\begin{aligned} \text{UAW} &= \sum_{i=1}^5 a_i \times w_i \\ &= 5 \times 2 = 10 \end{aligned}$$

$$\begin{aligned}
 \text{UUCW} &= \sum_{i=1}^{10} u_i \times w_i \\
 &= 10 \times 10 = 100 \\
 \text{UUCP} &= \text{UAW} + \text{UUCW} \\
 &= 10 + 100 = 110 \\
 \text{TCF} &= 0.6 + 0.01 \sum_{i=1}^{13} F_i \times w_i \\
 &= 0.6 + 0.01 \times (3 \times 2 + 3 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 0.5 \\
 &\quad + 3 \times 0.5 + 3 \times 2 + 3 \times 2 + 3 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 1) \\
 &= 0.6 + 0.01 \times 42 = 1.02 \\
 \text{ECF} &= 1.4 + (-0.03) \sum_{i=1}^8 F_i \times w_i \\
 &= 1.4 + (-0.03) \times (3 \times 1.5 + 3 \times 0.5 + 3 \times 0.5 + 3 \times 1 \\
 &\quad + 3 \times 1 + 3 \times -1 + 3 \times -1 + 3 \times 2) \\
 &= 0.995 \\
 \text{UCP} &= \text{UUCP} \times \text{TCF} \times \text{ECF} \\
 &= 110 \times 1.02 \times 0.995 \\
 &= 111.639
 \end{aligned}$$

EXAMPLE 4.4 The following information is available for an application:

2 simple actors, 2 average actors, 1 complex actor, 2 use cases with the number of transactions 3, 4 use cases with the number of transactions 5 and 2 use cases with the number of transactions 15. In addition to the above information, the system requires:

- (i) Significant user efficiency
- (ii) Essential ease to change
- (iii) Moderate concurrency
- (iv) Essential application experience
- (v) Significant object-oriented experience
- (vi) Essential stable requirements

Other technical and environmental complexity factors are treated as average. Compute the use case points for the project.

Solution

$$\begin{aligned}
 \text{UAW} &= \sum_{i=1}^5 a_i \times w_i \\
 &= 2 \times 1 = 2
 \end{aligned}$$

$$2 \times 2 = 4$$

$$1 \times 3 = 3$$

$$\text{UAW} = 2 + 4 + 3 = 9$$

$$\begin{aligned}\text{UUCW} &= \sum_{i=1}^8 u_i \times w_i \\ &= 2 \times 5 + 4 \times 10 + 2 \times 15 = 10 + 40 + 30 = 80\end{aligned}$$

$$\text{UUCP} = \text{UAW} + \text{UUCW}$$

$$= 9 + 80 = 89$$

$$\begin{aligned}\text{TCF} &= 0.6 + 0.01 \sum_{i=1}^{13} F_i \times w_i \\ &= 0.6 + 0.01 \times (3 \times 2 + 3 \times 1 + 4 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 0.5 \\ &\quad + 3 \times 0.5 + 3 \times 2 + 5 \times 1 + 2 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 1) \\ &= 0.6 + 0.01 \times 44 = 1.84\end{aligned}$$

$$\begin{aligned}\text{ECF} &= 1.4 + (-0.03) \sum_{i=1}^8 F_i \times w_i \\ &= 1.4 + -0.03 \times 3 \times 1.5 + 3 \times 0.5 + 3 \times 0.5 + 4 \times 1 \\ &\quad + 3 \times 1 + 3 \times -1 + 3 \times -1 + 5 \times 2 \\ &= 1.4 + (-0.03) \times 19.5 \\ &= 0.815\end{aligned}$$

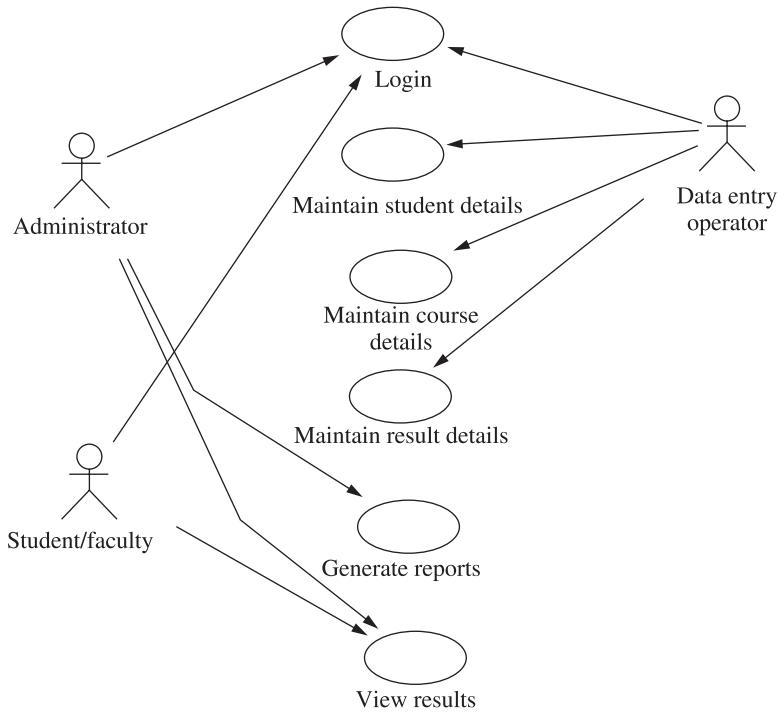
$$\begin{aligned}\text{UCP} &= \text{UUCP} \times \text{TCF} \times \text{ECF} \\ &= 89 \times 1.84 \times 0.815 \\ &= 133.46\end{aligned}$$

EXAMPLE 4.5 Consider Example 4.4 and calculate the effort for the given application.

Solution

$$\begin{aligned}\text{Effort} &= \text{UCP} \times 20 \\ &= 133.46 \times 20 \\ &= 2669.2 \text{ person hours}\end{aligned}$$

EXAMPLE 4.6 Consider the following use case model of result management system:



The number of transactions required for each use case is given as follows:

Use cases	Number of transactions
Maintain student details	7
Maintain course details	10
Maintain result details	5
Generate reports	3
View results	2
Login	4

Assume all complexity factors are average. Compute the use case points and effort for the project.

Solution

Use cases	Complexity
Maintain student details	15
Maintain course details	15
Maintain result details	10
Generate reports	5
View results	5
Login	10

(Contd.)

(Contd.)

Actors	Complexity
Data entry operator	2
Administrator	2
Student	1
Faculty	1
UUCP	66

$$\begin{aligned}
 TCF &= 0.6 + 0.01 \sum_{i=1}^{13} F_i \times w_i \\
 &= 0.6 + 0.01 \times (3 \times 2 + 3 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 0.5 + 3 \times 0.5 \\
 &\quad + 3 \times 2 + 3 \times 2 + 3 \times 1 + 3 \times 1 + 3 \times 1 + 3 \times 1) \\
 &= 0.6 + 0.01 \times 42 = 1.02
 \end{aligned}$$

$$\begin{aligned}
 ECF &= 1.4 + (-0.03) \sum_{i=1}^8 F_i \times w_i \\
 &= 1.4 + (-0.03) \times (3 \times 1.5 + 3 \times 0.5 + 3 \times 0.5 \\
 &\quad + 3 \times 1 + 3 \times 1 + 3 \times -1 + 3 \times -1 + 3 \times 2) \\
 &= 0.995
 \end{aligned}$$

$$\begin{aligned}
 UCP &= UUCP \times TCF \times ECF \\
 &= 66 \times 1.02 \times 0.995 \\
 &= 66.98
 \end{aligned}$$

$$\text{Effort} = 20 \times 66.98 = 1339.67 \text{ person hours}$$

4.4 Class Point Method

This method is used to provide the system level size estimation of the object-oriented software. The class point method was given by Costagliola and Tortora (2005). This method primarily focuses on classes for the estimation of size. The complexity of a class is analysed through determining the number of methods in a class, the number of attributes in a class and the interaction of a class with other classes. This method can be used for estimation during requirement and design phase of software development life cycle. The steps required for the estimation of size are given as follows:

1. Identification of classes
2. Determination of complexity of classes
3. Calculation of unadjusted class point
4. Calculation of technical complexity factor
5. Calculation of class point

4.4.1 Identification of Classes

In the class point method, four types of classes are identified. The types of system components are given in Table 4.6.

Table 4.6 Class type with description

Class type	Description	Example (library management system)
Problem domain type (PDT)	Represents real-world entities in the application domain.	Student, Book
Human interface type (HIT)	Satisfies the need for visualizing information and human-computer interaction.	LoginForm, BookDetailForm
Data management type (DMT)	Includes classes which incorporate data storage and retrieval.	Login
Task management type (TMT)	Includes classes which are responsible for tasks	BookDetailsController, StudentDtailsController

4.4.2 Classifying Class Complexity

In the class point method, two measures CP_1 and CP_2 are used. In CP_1 , the initial estimate of size is made and CP_2 provides detailed estimate of size. The following measures are used in the calculation of CP_1 and CP_2 measures:

1. *Number of external methods (NEM)*: It measures the size of class in terms of methods. It counts the number of public methods in a class.
2. *Number of service requested (NSR)*: It measures the interaction between system components. It counts the number of services requested by other classes.
3. *Number of attributes (NOA)*: It measures the complexity of a class. It counts the number of data members in a class.

In the CP_1 measure NEM and NSR are used to classify complexity of a class. Table 4.7 shows the complexity for CP_1 measure.

Table 4.7 Class complexity for CP_1

	0–4 NEM	5–8 NEM	≥ 9 NEM
0–1 NSR	Low	Low	Average
2–3 NSR	Low	Average	High
≥ 4 NSR	Average	High	High

In the CP_2 measure, NOA is also considered; thus, a detailed insight into the estimate of size is obtained. Tables 4.8 to 4.10 show the details for classifying class complexity for CP_2 measure on the basis of NEM, NSR and NOA.

Table 4.8 Class complexity for CP₂

		NOA		
		0–2 NSR	0–5	6–9
		0–4	Low	Low
NEM	5–8	Low	Average	High
	≥ 9	Average	High	High

Table 4.9 Class complexity for CP₂

		NOA		
		3–4 NSR	0–4	5–8
		0–3	Low	Low
NEM	4–7	Low	Average	High
	≥ 8	Average	High	High

Table 4.10 Class complexity for CP₂

		NOA		
		≥ 5 NSR	0–3	4–7
		0–2	Low	Low
NEM	3–6	Low	Average	High
	≥ 7	Average	High	High

4.4.3 Calculating Unadjusted Class Points

The total unadjusted class point (TUCP) is calculated by assigning complexity weights based on the classifications made as given in Table 4.11.

Table 4.11 Complexity weights for each class type

Class type	Description	Complexity weight		
		Low	Average	High
PDT	Problem domain type	3	6	10
HIT	Human interface type	4	7	12
DMT	Data management type	5	8	13
TMT	Task management type	4	6	9

After classifying the complexity level of classes, the TUCP is calculated as follows:

$$\text{TUCP} = \sum_{i=1}^4 \sum_{j=1}^3 w_{ij} \times x_{ij}$$

where w_{ij} is complexity weights j (j is low, average, high) assigned to class type i . x_{ij} is the number of classes of type i (i is type of class problem domain, human interface, data management and task management).

4.4.4 Calculating Technical Complexity Factor

The factors F_1-F_{18} are the degree of influence (DI) as shown in Figure 4.5. Each DI is rated on the scale of 0–5. DI is used to determine technical complexity factor (TCF). The TCF is determined by the following mathematical formula:

$$\text{TCF} = 0.55 + 0.01 \sum_{i=1}^{18} F_{ii}$$

F_i	Description	F_i	Description
1	Data communication	10	Reusability
2	Distributed functions	11	Ease of installation
3	Performance	12	Ease of operation
4	High used configuration	13	Multiple users
5	Transaction rate	14	Facilitation of change
6	Online data entry	15	Adaptability by user
7	End-user efficiency	16	Rapid prototyping
8	Online update	17	Multiuser interaction
9	Compiler processing	18	Multiple interfaces

Figure 4.5 Computing technical complexity factor.

4.4.5 Calculating Class Point and Effort

The final class point is calculated by multiplying total unadjusted class point values with technical factor. The procedure for calculating adjusted class point (CP) is given as:

$$\text{CP} = \text{TUCP} \times \text{TCF}$$

Costagliola and Tortora (2005) used data from 40 systems developed in Java language in order to predict the effort during two successive semesters of graduate courses of software engineering. They used ordinary least-square (OLS) regression analysis for deriving the effort model. The effort is defined in terms of person hours for both CP_1 and CP_2 measures as:

$$\text{Effort} = 0.843 \times \text{CP}_1 + 241.85$$

$$\text{Effort} = 0.912 \times \text{CP}_2 + 239.75$$

Two measures CP_1 and CP_2 have been described. The CP_1 measure can be used in early phases of software development and the CP_2 measure can be used when the number of attributes is available.

EXAMPLE 4.7 Consider a project with the following parameters:

Class type	Number of classes	NEM	NSR
PDT	4	6	2
		4	5
		10	15
		5	7
HIT	6	7	10
		5	8
		8	12
		6	9
		2	4
		4	2
DMT	3	2	3
		1	0
		3	4
TMT	2	4	8
		8	12

Assume all the technical complexity factors have average influence. Calculate class points.

Solution

Class type	Number of classes	NEM	NSR	Complexity	Weights
PDT	4	6	2	Average	6
		4	5	Average	6
		10	15	High	10
		5	7	High	10
HIT	6	7	10	High	12
		5	8	High	12
		8	12	High	12
		6	9	High	12
		2	4	Average	7
		4	2	Low	4
DMT	3	2	3	Low	5
		1	0	Low	5
		3	4	Average	8
TMT	2	4	8	Average	6
		8	12	High	9

The total unadjusted use case points are calculated as:

$$\begin{aligned}
 \text{TUCP} &= \sum_{i=1}^4 \sum_{j=1}^3 w_{ij} \times x_{ij} \\
 &= 6 \times 2 + 10 \times 2 + 12 \times 4 + 7 \times 1 + 4 \times 1 + 5 \times 2 + 8 \times 1 + 6 \times 1 + 9 \times 1 \\
 &= 12 + 20 + 48 + 7 + 4 + 10 + 8 + 6 + 9 \\
 &= 124 \\
 \text{TCF} &= 0.55 + 0.01 \sum_{i=1}^{18} F_{ii} \\
 &= 0.55 + 0.01(18 \times 3) \\
 &= 0.55 + 0.54 = 1.09 \\
 \text{CP}_1 &= 124 \times 1.09 = 135.16
 \end{aligned}$$

EXAMPLE 4.8 Consider a result management system with the following information:

Class name	Class type	NEM	NSR	NOA
Login	PDT	4	1	10
Student	PDT	3	3	12
Course	PDT	4	0	8
Marks	PDT	4	1	5
StudentForm	HIT	1	1	1
CourseForm	HIT	4	2	1
MarksForm	HIT	2	8	1
LoginDetail	DMT	2	1	0
StudentDetail	DMT	4	2	0
CourseDetail	DMT	5	3	0
MarksDetail	DMT	6	2	0
LoginChecker	TMT	2	0	0
MarksControl	TMT	2	2	0
CourseControl	TMT	3	3	0

In addition, the system requires

1. User adaptability is significant
2. Rapid prototyping has strong influence
3. Multiple interfaces are significant
4. Distributed functions are moderate

Assume all other factors as average. Calculate CP_1 and CP_2 using class point method and effort.

Solution

Class name	Class type	NEM	NSR	NOA	CP ₁	w ₁	CP ₂	w ₂
Login	PDT	4	1	10	Low	3	Average	6
Student	PDT	3	3	12	Low	3	Average	6
Course	PDT	4	0	8	Low	3	Average	6
Marks	PDT	4	1	5	Low	3	Low	3
StudentForm	HIT	1	1	1	Low	4	Low	4
CourseForm	HIT	4	2	1	Low	4	Low	4
MarksForm	HIT	2	8	1	Average	7	Low	4
LoginDetail	DMT	2	1	0	Low	5	Low	5
StudentDetail	DMT	4	2	0	Average	8	Low	5
CourseDetail	DMT	5	3	0	Average	8	Low	5
MarksDetail	DMT	6	2	0	Low	5	Low	5
LoginChecker	TMT	2	0	0	Low	4	Low	4
MarksControl	TMT	2	2	0	Low	4	Low	4
CourseControl	TMT	3	3	0	Low	4	Low	4
						65		65

$$\begin{aligned}
 TCF &= 0.55 + 0.01 \sum_{i=1}^{18} F_{ii} \\
 &= 0.55 + 0.01(14 \times 3 + 2 \times 4 + 1 \times 5 + 1 \times 2) \\
 &= 0.55 + 0.01 \times 57 = 1.12
 \end{aligned}$$

$$CP_1 = 65 \times 1.12 = 72.8$$

$$CP_2 = 65 \times 1.12 = 72.8$$

$$\text{Effort} = 0.843 \times CP_1 + 241.85 = 303.22 \text{ person hours}$$

$$\text{Effort} = 0.912 \times CP_2 + 239.75 = 306.14 \text{ person hours}$$

4.5 Object-Oriented Function Point

The traditional function point method can be mapped to obtain object points. The **object point sizing method** is also known as **object-oriented function point method** as it is very similar to the function point method. The object point method does not require much experience, requires less effort for computation and can be calculated quickly. It better suites for object-oriented systems and is easier to calculate as compared to the traditional function point method.

The object point method involves counting of classes and methods (services). All this information is obtained from the object model of object-oriented design. In this approach, the function point concepts are mapped to object-oriented concepts and the ambiguities present in the function point method are removed.

4.5.1 Relationship between Function Points and Object Points

A class is a template that encapsulates attributes and member functions into a single unit. Objects are data members of the class. Classes may be divided into **external and internal classes depending on their scope and boundary**. The relationship between concepts defined in function point and their corresponding concepts in object points are shown in Table 4.12. The internal logical files are mapped to internal classes, external interface files are mapped to external classes and external inquiries/inputs/outputs are mapped to services.

Table 4.12 Relationship between function point and object point

Function point concept	Object point concept
Internal logical files	Internal class
External interface files	External class
External inquiries	Services (methods)
External outputs	
External inputs	

The internal classes are the classes that reside inside the application boundaries and the external classes are the classes that reside **outside the application boundaries**. The services are the methods defined in the class. Figure 4.6 depicts the method to compute the object-oriented concepts in object point method. For each external/internal class, it is necessary to compute the number of data element types (DETs) and record element types (RETs). The DETs correspond to the total number of attributes of a class and the RETs correspond to the total number of subclasses of a class (descendants of a class) as shown in Figure 4.7. In inheritance hierarchy, the classes that inherit the properties of the base class while having their own properties are known as subclasses.

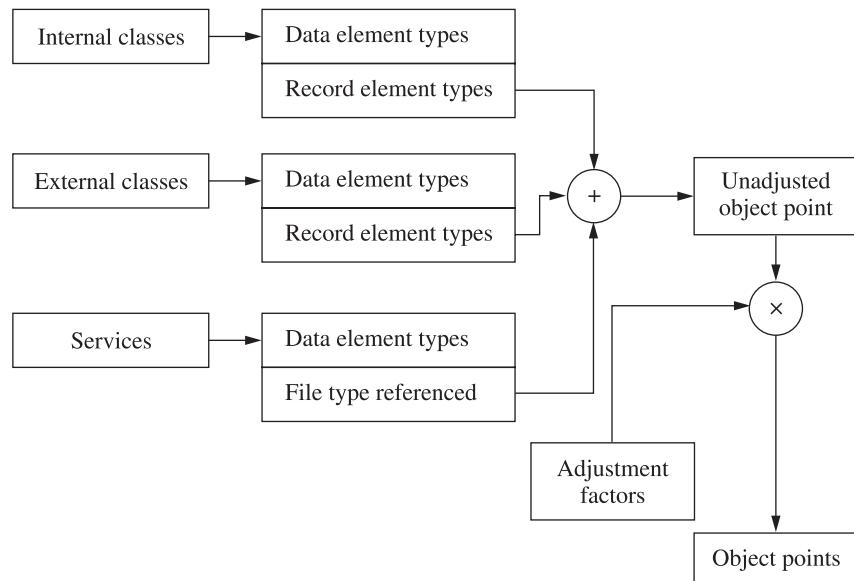


Figure 4.6 Steps for calculating object points.

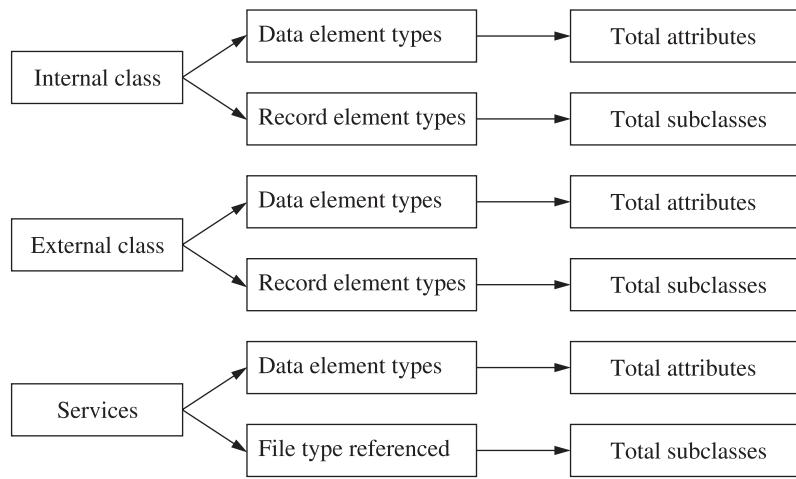


Figure 4.7 Calculation of DETs and RETs.

4.5.2 Counting Internal Classes, External Classes and Services

An internal class identifies the total number of DETs and RETs in the class. In the IFPUG Manual 4.1, DETs are defined as (IFPUG, 1994):

A unique user recognizable, non repeatable field.

DETs are counted by applying the following rules:

1. A DET is counted for each simple attribute (integer, string, real) defined in a class. For example, the book accession number (integer type) stored in a class is counted as one DET.
2. A DET is counted for each attribute required to communicate with another internal class or external class (association or aggregation relationship; see details in Chapter 5).

In the IFPUG Manual 4.1, RET is defined as (IFPUG, 1994):

A unique recognizable subgroups of data elements within an internal logical file or external interface file.

In object-oriented systems, these subgroups are known as subclasses or descendants. The RET is counted for each of a given class. For example, if employee is a base class and salary, contract based or hourly employee are three subclasses of class employee, then the RET count for employee class is 3. One of the following rules is applicable to a class while counting RETs:

1. Count an RET for each subclass of the internal/external class.
2. If there are no subclasses, count internal/external classes as one RET.

Each service in the class is examined. For each service, the number of DETs and file type referenced (FTR) needs to be counted. The counting of DETs and FTR involves simple and complex data types referenced by the methods. A simple data type is a compiler-defined data type and a complex data type is a user-defined data type. The following are the counting rules for DETs and FTR for each service:

1. A DET is counted for each simple data type referenced as arguments of the service or global variables referenced by the service.
2. An FTR is counted for each complex data type referenced as arguments of the service or returned by the service.

In Tables 4.13, 4.14 and 4.15, the complexity values for internal classes, external classes and services are shown.

Table 4.13 Complexity table for internal class

	1 to 19 DETs	20 to 50 DETs	51 or more DETs
0 to 1 RETs	Low	Low	Average
2 to 5 RETs	Low	Average	High
6 or more RETs	Average	High	High

Table 4.14 Complexity table for external class

	1 to 19 DETs	20 to 50 DETs	51 or more DETs
0 to 1 RETs	Low	Low	Average
2 to 5 RETs	Low	Average	High
6 or more RETs	Average	High	High

Table 4.15 Complexity table for services

	1 to 4 DETs	5 to 15 DETs	16 or more DETs
0 to 1 FTR	Low	Low	Average
2 FTR	Low	Average	High
3 or more FTR	Average	High	High

Consider that the example of the result calculation of a student is shown. The class model is given in Figure 4.8.

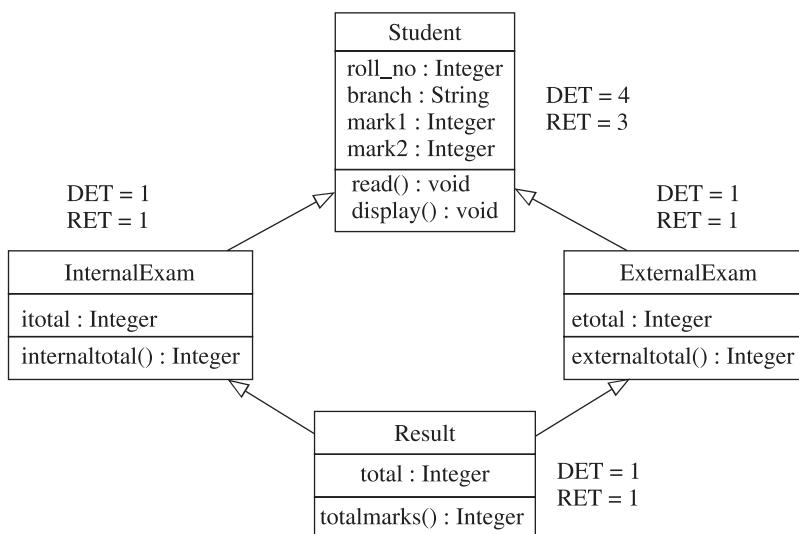


Figure 4.8 Class model of result management system.

In Table 4.16, the RET and DET are calculated for each class shown in Figure 4.8.

Table 4.16 Assigning complexity values in the given example

Class	DET	RET	Complexity	Complexity value
Student	4	3	Low	7
Internal exam	1	1	Low	7
External exam	1	1	Low	7
Result	1	1	Low	7

Table 4.17 shows the values of complexity types (high, average, low) for internal classes, external classes and services.

Table 4.17 Values of complexity

Function type	Low	Average	High
Internal class (IC)	7	10	15
External class (EC)	5	7	10
Services (S)	3	4	6

After classifying the internal classes, external classes and services, these are multiplied by their complexity values. Finally, the results are summed up using the following formulas:

$$IC_{total} = \sum_{i=1}^n IC_i$$

$$EC_{total} = \sum_{j=1}^m EC_j$$

$$S_{total} = \sum_{k=1}^o S_i$$

where n is the number of internal classes, m is the number of external classes and o is the number of services.

The classes in the given example are internal, hence the occurrences of internal classes are multiplied with their corresponding weights and summing all the resulting values, we get $IC = 4 \times 7 = 28$. The concrete services in the classes are 5. Assuming that the methods have average complexity (as their signature is not given in the example), the services are $5 \times 4 = 20$.

4.5.3 Calculating Unadjusted Object Points

The unadjusted object points (UOP) are obtained as follows:

$$UOP = IC_{total} + EC_{total} + S_{total}$$

In the example given in Figure 4.8, the $UOP = IC_{total} + S_{total} = 28 + 20 = 48$.

4.5.4 Adjustment Factors

The F_i ($i = 1$ to 14) are the degree of influence and are based on responses to questions noted in Figure 4.9. Each factor is rated on a scale of 0 to 5. The adjustment factor (AF) is calculated using the following mathematical formula:

$$AF = 0.65 + 0.01 \sum_{i=1}^{14} F_i$$

Finally the adjusted object points are calculated by multiplying UOP by AF as follows:

$$OP = UOP \times AF$$

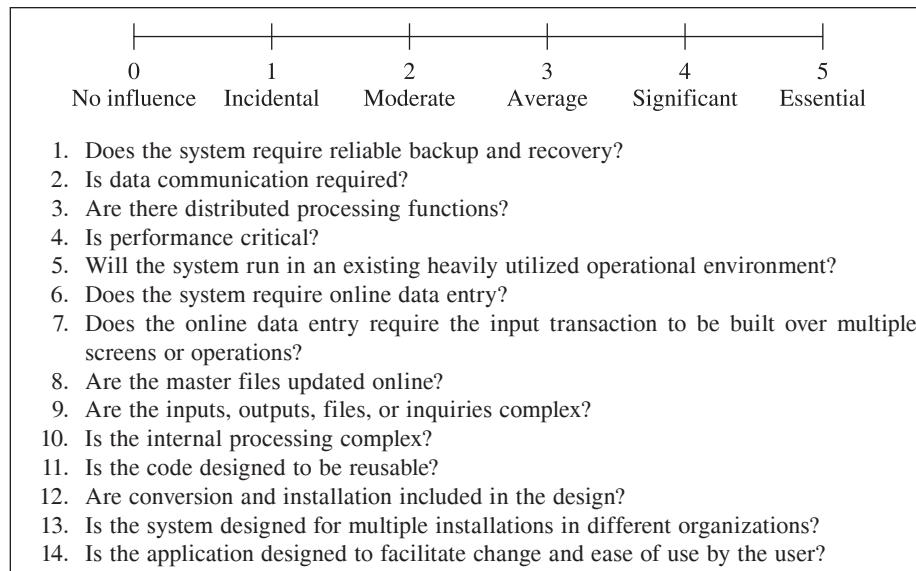


Figure 4.9 Adjustment factors.

EXAMPLE 4.9 Consider a project with the following data:

Internal class	DET	RET
C_1	4	2
C_2	5	7
C_3	3	1
C_4	2	1
External class	DET	RET
C_5	6	10

The above classes contain six methods with high complexity and the complexities of the adjustment factors are average. Compute object points.

Solution

$$\begin{aligned} IC_{\text{total}} &= 3 \times 7 + 1 \times 10 = 31 \\ EC_{\text{total}} &= 1 \times 7 = 7 \\ S_{\text{total}} &= 6 \times 6 = 36 \\ UOP &= IC_{\text{total}} + EC_{\text{total}} + S_{\text{total}} \\ &= 31 + 7 + 36 = 74 \end{aligned}$$

$$\begin{aligned} AF &= 0.65 + 0.01 \sum_{i=1}^{14} F_i \\ &= 0.65 + 0.01 (14 \times 3) \\ &= 0.65 + 0.42 = 1.07 \\ OP &= UOP \times AI \\ &= 74 \times 1.07 \\ &= 79.18 \end{aligned}$$

EXAMPLE 4.10 An application has the following data:

5 high internal classes, 2 average external classes and 6 average services. Assume the adjustment factor as significant. What are the unadjusted object points and object points?

Solution

$$\begin{aligned} IC_{\text{total}} &= 5 \times 15 = 75 \\ EC_{\text{total}} &= 2 \times 7 = 14 \\ S_{\text{total}} &= 6 \times 4 = 24 \\ UOP &= IC_{\text{total}} + EC_{\text{total}} + S_{\text{total}} \\ &= 75 + 14 + 24 = 113 \\ AF &= 0.65 + 0.01 \sum_{i=1}^{14} F_i \\ &= 0.65 + 0.01 (14 \times 4) \\ &= 0.65 + 0.56 = 1.21 \\ OP &= UOP \times AI \\ &= 113 \times 1.21 \\ &= 136.73 \end{aligned}$$

4.6 Risk Management

Identifying and managing risks in the early phases of software development life cycle can prevent software companies from facing major disasters. **Risk is defined as the probability of**

some unexpected happening in the future, a situation that can cause unwanted results. Risk management involves identifying and estimating the probability of risks with their order of severity.

4.6.1 What is Risk?

Risk is defined as the combined effect of probability of occurrence of an undesirable event and the impact of the occurrence of this event. Risk can delay the delivery of the software and over budget a project. Risky projects may not also meet specified quality levels. The risk may be defined as:

$$\text{Risk} = \text{Probability of occurrence of an undesired event} \times \text{Impact of occurrence of that event}$$

There are various types of risks that may cause threat to the completion of a project. These include technical risks, economical risks, deployment risks and environment risks. Technical risks deal with the feasibility and understanding of the problem. Do we actually know the problem? Is there any feasible solution to the problem? These risks must be analysed during the start of the project life cycle. Economical risks involve budget, time, personnel and quality risks. After identification, these threats require continuous monitoring. Deployment risks consist of mishandling of the software, inadequate user training and ineffective maintenance activities. No matter how much high quality software is produced it cannot run without environmental support. The security and safety of the workplace is a very important issue that should be addressed. The security issues involve catastrophic failures, unauthorized access and virus threats.

The risk may be rated according to urgency to address these risks. Initial identification of threat potential must lead to effective planning and analysis of risks. Risks may be rated as

1. *Urgent*: Risks that would cause high loss to the business.
2. *High*: Risks that would prevent the delivery of the software.
3. *Medium*: Risk may affect the company from meeting a milestone.
4. *Low*: Routine risks with little or no impact.

It is important to classify risks so that they can be analysed and prioritized based on their probabilities and impacts. Risks rated urgent should be addressed before the risks rated high, as they cause huge loss to the organization. Similarly, high-rated risks are more severe than medium rated-risks. Thus, the high-rated risks should be addressed before the medium-rated risks.

4.6.2 Framework for Managing Risks

Risk management is a key part of project planning activities and the specific risky areas are highlighted in the plan. The project plan is expected to highlight both probability of failure and impact of the failure and to describe the steps to be taken in order to reduce the risk. Risk management consists of the following steps:

1. Risk identification
2. Risk analysis and prioritization
3. Risk avoidance and mitigation
4. Risk monitoring

Steps 1–3 are carried out iteratively in order to identify, analyse, prioritize and reduce risks. The risks should be identified and appropriate plans must be made to reduce the high prioritized risks. After reducing the high prioritized risks, the impact of reducing these risks should be reanalysed and also risks that may have been introduced should be identified. For example, if a new technology is introduced in order to resolve risk for delay in the schedule of the software. However, the hiring of new experts might introduce new risks. After identifying new risks, the risk reduction and removal activities may be planned again.

4.6.3 Risk Identification

In risk identification, we consider what unexpected can happen? More specifically, we are looking for unusual events that may occur and prevent the delivery of the software on time, with specified quality and within budget. Risk identification primarily involves brainstorming activities and preparation of risk list.

Brainstorming is a group discussion technique in which stakeholders such as actual users, developers and managers may be brought together. These group discussions may lead to new ideas and promote creative thinking. Brainstorming sessions can be used to identify the potential problems and the possible solutions to these problems. Preparation of risk lists involves identification of generic risks that may have been found continuously in previous software projects. The top 10 risk items are given by B.W. Boehm and are summarized in Table 4.18 (Boehm, 1989).

Table 4.18 Risk checklist

S. No.	Risks	Risk reduction strategies
1	Short of personnel	<ul style="list-style-type: none"> • Hiring top professionals • Teamwork
2	Unrealistic cost and time estimates	<ul style="list-style-type: none"> • Use of more than one estimation technique • Analysis of historical data from past projects • Incremental methods • Standardization of methods
3	Software developed with incorrect functions	<ul style="list-style-type: none"> • Prototyping • Construction of early user manuals • Early user feedback
4	Software developed with incorrect user interface	<ul style="list-style-type: none"> • Prototyping • Continuous user involvement
5	Gold plating of requirements	<ul style="list-style-type: none"> • Identify important requirements • Prototyping • Incremental development
6	Requirements are changed late in software development life cycle	<ul style="list-style-type: none"> • Stringent change control processes • Incremental software development
7	Shortfalls in externally purchased components	<ul style="list-style-type: none"> • Conduct of verification activities such as walkthroughs and inspections • Quality assurance and management activities • Certifications

(Contd.)

Table 4.18 Risk checklist (*Contd.*)

S. No.	Risks	Risk reduction strategies
8	Shortfalls in externally performed activities	<ul style="list-style-type: none"> • Quality assurance activities • Prototyping
9	Shortfalls in real-time performance	<ul style="list-style-type: none"> • Prototyping • Simulation • Benchmarking • Technical analysis
10	Technical difficulty in development	<ul style="list-style-type: none"> • Training of person • Technical analysis • Hiring of experts

The stakeholders must go through the risk checklist given in Table 4.18 and identify the risks specific to their project.

4.6.4 Risk Analysis and Prioritization

There may be many risky areas in the project. These risks should be addressed according to their priority. Risk analysis and prioritization is a process consisting of the following steps:

1. Identifying the problems.
2. Assigning probability of occurrence value.
3. Assigning impact of occurrence value.
4. Assigning values obtained in steps 2 and 3 on a scale of 1 (low) to 10 (high).
5. Calculating risk exposure factor which is the product of probability of occurrence of a problem and impact of the problem on software operation.
6. Risks are ranked on the basis of the value of risk exposure.
7. A risk table (shown in Table 4.19) is prepared.

Table 4.19 Table for risk analysis

S. No.	Problem	Probability of occurrence of a problem	Impact of the problem	Risk exposure	Priority
	R ₁				
	R ₂				
	R ₃				

On the basis of the case study on “Library Management System” given in Chapter 3, the potential problems are identified. The risk exposure factor is calculated for each potential problem which is the multiplication of probability of occurrence of the problem and impact of that problem on software operation. The risk analysis and prioritization table is given in Table 4.20.

Table 4.20 Risk analysis table for library management system

S. No.	Problem	Probability of occurrence of a problem	Impact of the problem	Risk exposure	Priority
R ₁	Issue of incorrect password	2	2	4	10
R ₂	Incorrect entry in book details form	8	2	16	4
R ₃	Incorrect user interface of student details form	7	3	21	1
R ₄	Coding takes longer than expected	4	5	20	2
R ₅	Lack of expertise causes delay in the project	1	10	10	6
R ₆	Testing reveals lots of defects	1	9	9	7
R ₇	Design is not robust	2	7	14	5
R ₈	Documentation is in improper format	4	2	8	8
R ₉	Changes in requirements during implementation	2	9	18	3
R ₁₀	Requirements specification takes longer than expected	7	1	7	9

The potential problems ranked by the risk exposure factor are R₃, R₄, R₉, R₂, R₇, R₅, R₆, R₈, R₁₀, R₁. A risk matrix may be used to capture the potential risks. The probability of problem and impact of problem values may be classified as urgent, high, medium, and low based on their rankings. Tables 4.21 and 4.22 show the probability of occurrence of the problem and impact with their corresponding types of risks, respectively. The values obtained on the scale of 1–10 are changed on the scale of 0 to 1.

Table 4.21 Probability type

Probability of occurrence of problem	Type
0.8–1	Urgent
0.6–0.7	High
0.3–0.6	Medium
0.0–0.3	Low

Table 4.22 Impact type

Probability of impact	Type
0.5–1	Urgent
0.3–0.5	High
0.1–0.3	Medium
0.0–0.1	Low

In Figure 4.10, the risk matrix for the LMS is shown. The values of probability and impact are shown by their position of the risk in the matrix. The importance may be given either to probability or to impact depending upon the needs of software under development.

Urgent	R ₅ , R ₆ , R ₉ , R ₇	R ₄		
High		R ₈	R ₃	R ₂
Medium	R ₁		R ₁₀	
Low				

Low Medium High Urgent

Figure 4.10 Risk matrix.

Risk matrix is a popular tool for designing prioritization schemes. The probabilities of estimates on the scale of 1–10 are required in order to construct a risk matrix. The probability and impact values may be categorized as urgent, high, medium, and low. These two values are essential for prioritizing risks. After the risks are ranked, the urgent and high priority risks should be reduced first.

4.6.5 Risk Avoidance and Mitigation Strategies

The purpose of risk avoidance strategies is to altogether **eliminate the occurrence of risks**. One possible way to avoid risks is to reduce the scope of the projects by **removing unnecessary requirements**. When the number of requirements is reduced, the whole risk list will come down.

In mitigation strategies, plans are developed in order to **reduce the impact of the risk** when it occurs. For example, to reduce the impact of risk that a component A will not perform adequately, a set of tests may be used to ensure its proper functioning. If we want to reduce the risk that a design tool will not create the design of the application effectively, its features will be investigated before hand to ensure that the design is built successfully. Similarly, by taking regular backups of the data, the impact of database corruption may be reduced.

4.6.6 Risk Monitoring

The risks should be monitored on continuous basis by reevaluating the risks, the probability of occurrence of risks and the impact of the risk. This ensures that:

- **The identified risks have been reduced or resolved.**
- **The new risks are discovered.**
- **The impact and magnitude of the risk are reassured.**

The risk can be monitored by scheduling regular review meetings to evaluate risks. Some risks may move down the risk list, some may be eliminated from the list and some new risks may be identified and added to the list.

4.6.7 Estimating Risk Based on Schedule

If the activities of the project take longer than expected, they will most likely cause risk of delay in the completion of the project. The importance of identifying critical path activities and uncertainties is that these have high risk for causing delay in the projects. Critical path method (CPM) and program evaluation review technique (PERT) are two well-known methods for identifying critical activities and estimating uncertainties of meeting dates. Both of the techniques are used for visualizing the projects where activities are represented as arrows joining circles or nodes.

Critical Path Method

The CPM shows the activities that are crucial to the end date of the project. In the CPM, arrows represent activities that take time to execute and circles represent events. The events represented by the circles are divided into four quadrants:

Event number: It shows the event number, i.e. the order in which the activities must be executed in the project.

Earliest start date: The earliest date by which the event must occur.

Latest date: The latest date by which the event must occur.

Slack: Slack is a measure that calculates how much an activity may be delayed without affecting the finish date of the project.

The node for the CPM chart is shown in Figure 4.11.

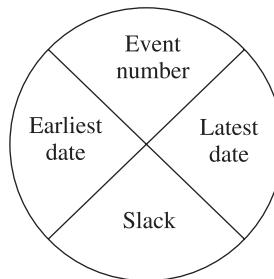


Figure 4.11 Node of CPM.

The estimated durations of activities given in Table 4.23 are used to illustrate the construction of networks by the CPM.

Table 4.23 An example application

Activity	Description	Required predecessor	Duration (months)
A	Market research	—	5
B	Product analysis	—	1
C	Product design	A	2
D	Product model	A	3
E	Sales brochure	A	2

(Contd.)

Table 4.23 An example application (*Contd.*)

Activity	Description	Required predecessor	Duration (months)
F	Cost analysis	C	3
G	Implementation	D	4
H	Testing	B, E	2
I	Training	H	1
J	Project report	F, G, I	1

Forward Pass

In the forward pass, the earliest date of the event is calculated. The forward pass of the example given in Table 4.23 is shown in Figure 4.12. Using Figure 4.12 and Table 4.24, the forward pass is carried out as follows:

1. Activities A and B may start immediately, so the earliest start date for both the events is 0.
2. Activity A will take 5 months to complete, hence the earliest end date is 5 for this activity.
3. Activity B will take 1 month to complete, hence the earliest end date is 1 for activity B.
4. Activity C must begin after the completion of activity A, hence the earliest start date for activity C is 5 months and the earliest end date is 7 months (duration of activity C is 2 months).
5. Activity D must also begin after the completion of activity A. The earliest start date is 5 months and the earliest end date is 8 months.
6. Similarly, the start date and the end date are calculated for all the other activities.

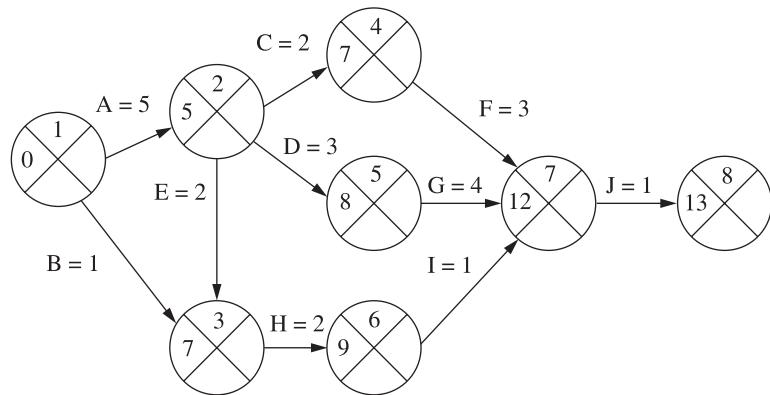
**Figure 4.12** Forward pass.

Table 4.24 Table after forward pass

Activity	Duration (months)	Earliest start date	Earliest end date
A	5	0	5
B	1	0	1
C	2	5	7
D	3	5	8
E	2	5	7
F	3	7	10
G	4	8	12
H	2	7	9
I	1	9	10
J	1	12	13

Backward Pass

Now, we follow the backward pass in order to compute the latest dates till when the project must be started and finished without any delay in the project. Figure 4.13 and Table 4.25 show the calculation of the latest date for each event of the project.

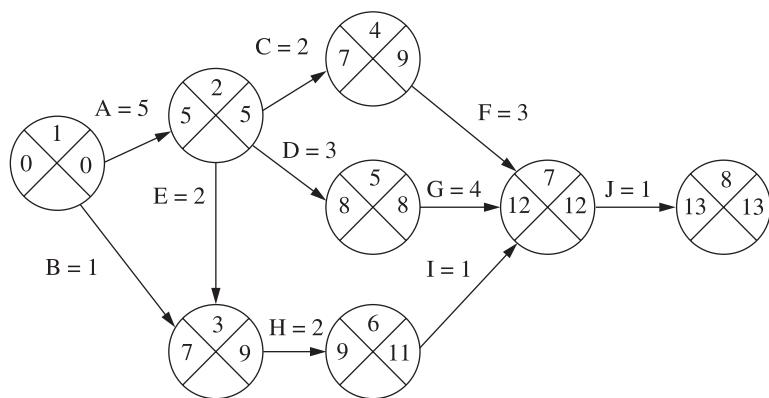


Figure 4.13 Backward pass.

The backward pass is carried out as follows:

1. The latest completion date for activity J is assumed to be 13 months.
 2. Activity J must therefore latest start at 12 ($13 - 1$) months and latest end at month 3.
 3. The latest end dates for activities F, G and I are at 12 months at which activity J must start. They have the latest start date months 9 ($12 - 3$), 8 ($12 - 4$) and 11 ($12 - 1$), respectively.
 4. Activity H must be started at 9 ($11 - 2$) month and completed by 11 month. Similarly activity E must begin at month 7 and end at month 9. Activity E must begin at month 7 and end at month 9.

5. Activities C and D must start at month 7 (9 – 2) and month 5 (8 – 3), respectively. They must end at months 9 and 8, respectively.
6. Activity B must be completed by month 9 (the latest start date for activity H). Activity A must finish at month 5 (the latest start date for activity D).
7. The latest start date for the project is zero.

The backward pass rule states that when there is more than one activity following an activity, then we take the earliest of the latest start date for these activities. For example, there are three activities C, D and E with latest start dates 7, 5 and 7, respectively. Thus, the latest finish date for activity A is 5.

Table 4.25 Table after backward pass

Activity	Duration (months)	Earliest start date	Earliest end date	Latest start date	Latest end date
A	5	0	5	0	5
B	1	0	1	8	9
C	2	5	7	7	9
D	3	5	8	5	8
E	2	5	7	7	9
F	3	7	10	9	12
G	4	8	12	8	12
H	2	7	9	9	11
I	1	9	10	11	12
J	1	12	13	12	13

Calculation of Slack and Identification of Critical Path

The slack of the events is calculated by computing the difference between the earliest date and the end date. The critical path is identified as the path with the slack value equal to zero. The critical path identifies those activities which are crucial and cannot be delayed. The final CPM chart with activities having slack value zero shown in dotted lines is given in Figure 4.14.

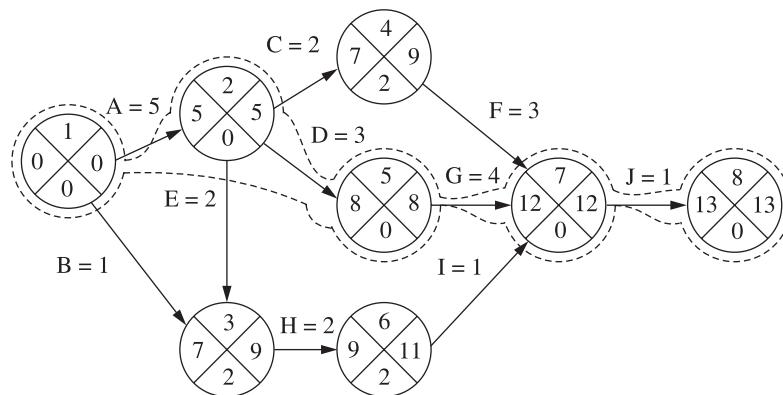


Figure 4.14 Final pass.

After identifying the critical path, the activities with slack value zero are identified to be risky and critical and are cause of concern.

EXAMPLE 4.11 Consider the description of a set of activities given below. Draw the CPM chart and identify high risky activities.

Activity	Description	Required predecessor	Duration (months)
A	Feasibility study	–	3
B	Requirement analysis	–	2
C	Requirement specification	–	4
D	Cost analysis	–	3
E	Product design	A, B	2
F	System model	E	4
G	Implementation	F	2
H	Unit level test cases	D	1
I	Testing	G, H	2
J	Training	C, I	4

Solution The forward pass is given in Figure 4.15 and the data is given in Table 4.26.

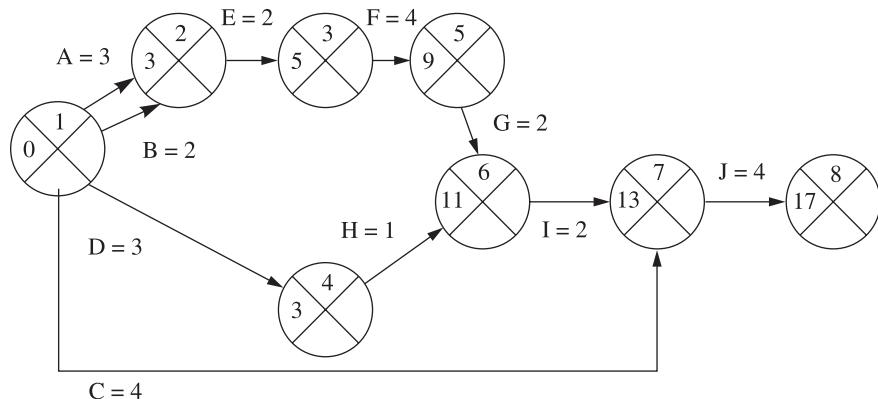


Figure 4.15 Forward pass.

Table 4.26 Table after forward pass

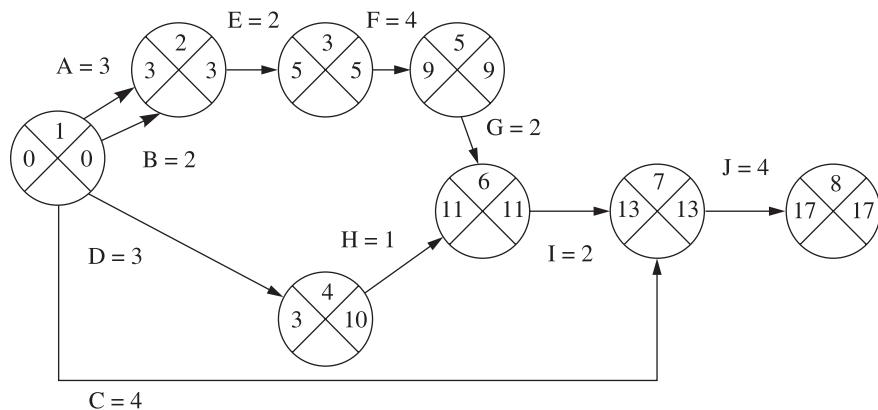
Activity	Duration (months)	Earliest start date	Earliest end date
A	3	0	3
B	2	0	2
C	4	0	4
D	3	0	3

(Contd.)

Table 4.26 Table after forward pass (*Contd.*)

Activity	Duration (months)	Earliest start date	Earliest end date
E	2	3	5
F	4	5	9
G	2	9	11
H	1	3	4
I	2	11	13
J	4	13	17

The backward pass is shown in Figure 4.16 and the final data is given in Table 4.27.

**Figure 4.16** Backward pass.**Table 4.27** Table after backward pass

Activity	Duration (months)	Earliest start date	Earliest end date	Latest start date	Latest end date
A	3	0	3	0	3
B	2	0	2	1	3
C	4	0	4	9	13
D	3	0	3	7	10
E	2	3	5	3	5
F	4	5	9	5	9
G	2	9	11	9	11
H	1	3	4	10	11
I	2	11	13	11	13
J	4	13	17	13	17

The final CPM chart is shown in Figure 4.17. The critical path is A-E-F-G-I-J.

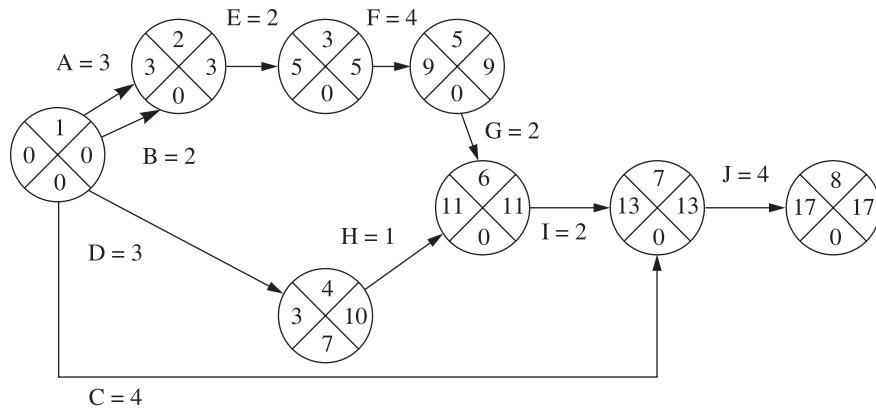


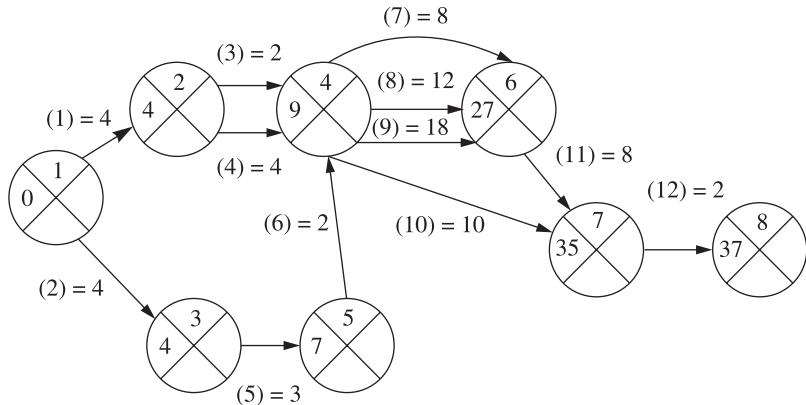
Figure 4.17 Final pass.

EXAMPLE 4.12 For a software project, consider the following information:

Activity No.	Activity name	Duration (weeks)	Immediate predecessor
1	Obtain requirements	4	—
2	Determine operations	4	—
3	Identify and define subsystems	2	1
4	Develop database	4	1
5	Make decision analysis	3	2
6	Identify risks, assumptions, constraints	2	5
7	Build module A	8	3, 4, 6
8	Build module B	12	3, 4, 6
9	Build module C	18	3, 4, 6
10	Write report	10	6
11	Testing and integration	8	7, 8, 9
12	Deployment	2	10, 11

Draw the CPM chart. Find out the project completion time and the critical path.

Solution The forward pass of the above problem is shown in Figure 4.18 and the table for the forward pass is shown in Table 4.28. The backward pass is shown in Figure 4.19 and the data for the backward pass is given in Table 4.29.

**Figure 4.18** Forward pass.**Table 4.28** Table for forward pass

Activity	Duration (months)	Earliest start date	Earliest end date
1	4	0	4
2	4	0	4
3	2	4	6
4	4	4	8
5	3	4	7
6	2	7	9
7	8	9	17
8	12	9	21
9	18	9	27
10	10	9	19
11	8	27	35
12	2	35	37

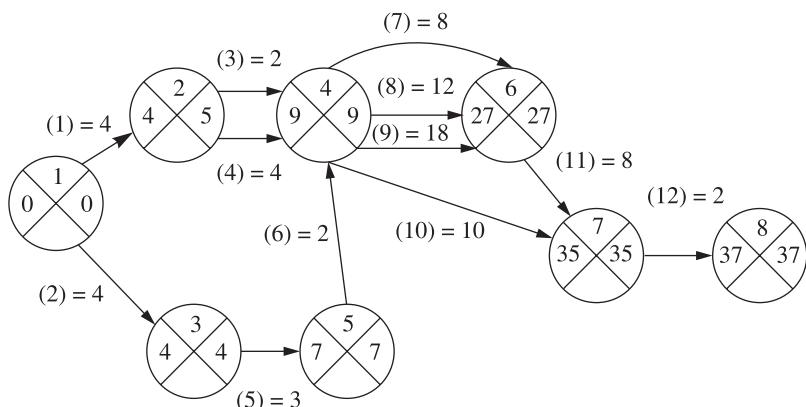
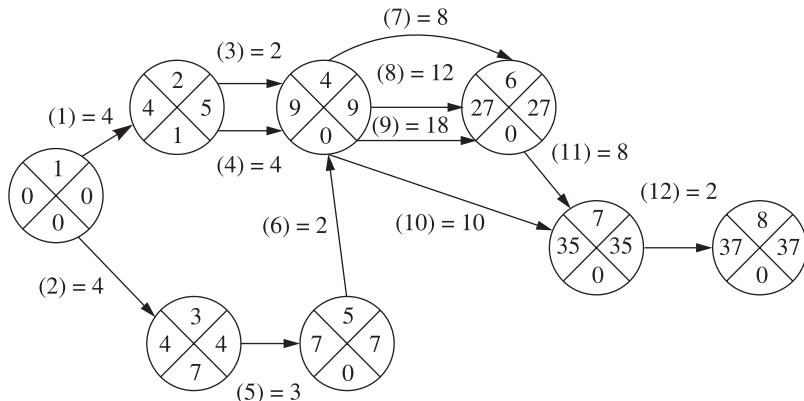
**Figure 4.19** Backward pass.

Table 4.29 Table after backward pass

Activity	Duration (months)	Earliest start date	Earliest end date	Latest start date	Latest end date
1	4	0	4	1	5
2	4	0	4	0	4
3	2	4	6	7	9
4	4	4	8	5	9
5	3	4	7	4	7
6	2	7	9	7	9
7	8	9	17	19	27
8	12	9	21	15	27
9	18	9	27	9	27
10	10	9	19	25	35
11	8	27	35	27	35
12	2	35	37	35	37

The final CPM chart is shown in Figure 4.20.

**Figure 4.20** Final pass.

Program Evaluation Review Technique (PERT)

PERT can be used to measure the uncertainty in activity durations. The PERT was developed for high-risk, expensive and large-sized project. Thus, the environment was similar of today's software project. The PERT is based on the CPM technique and requires the following estimates:

1. Most likely time (m): The expected time taken by a project in ordinary conditions.
2. Optimistic time (a): The expected shortest time taken by a project.
3. Pessimistic time (b): The worst possible time that a project may take.

The PERT calculates the total estimated time by using the above three estimates as

$$t = \frac{a + 4m + b}{6}$$

Table 4.30 shows the additional time estimates for project taken in Table 4.23.

Table 4.30 Table with time estimates

Activity	Optimistic (<i>a</i>)	Most likely (<i>m</i>)	Pessimistic (<i>b</i>)
A	4	5	7
B	0.5	1	2
C	1	2	2
D	2	3	3
E	1	2	3
F	1	3	4
G	3	4	6
H	1	2	3
I	0.5	1	2
J	1	1	1.5

The nodes in the PERT network are divided into four quadrants as shown in Figure 4.21.

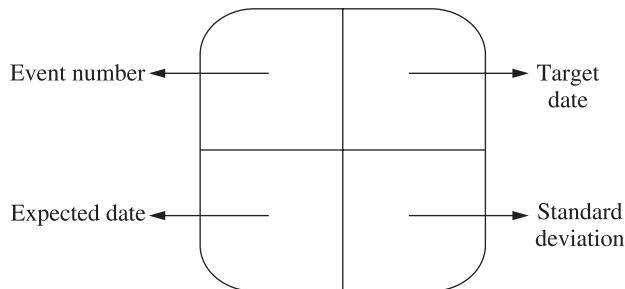


Figure 4.21 PERT node.

In the forward pass, we calculate the expected duration as shown in Figure 4.22. Table 4.31 shows the expected time of the project. Unlike the CPM technique, the PERT is based on the expected date instead of the earliest date by which the project must be completed. Thus, the PERT deals with the uncertainties involved in durations. The three estimators put emphasis on the fact that we are uncertain about the future happenings and thus the approximate estimate of the durations is taken.

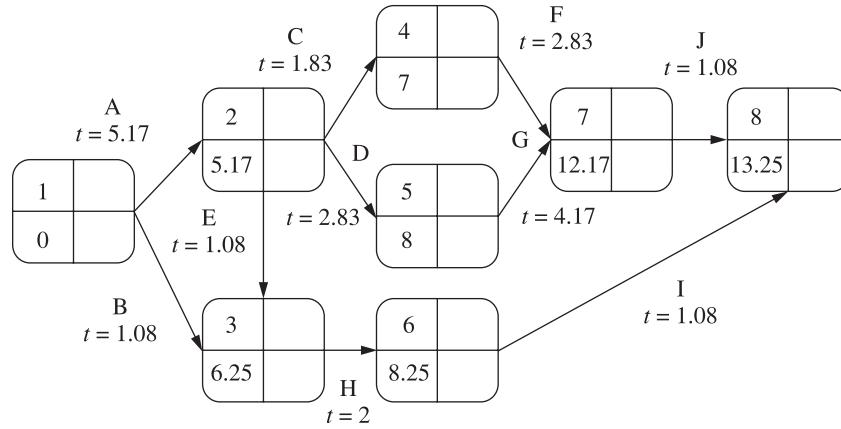


Figure 4.22 First pass of PERT.

Table 4.31 Expected time and standard deviation for PERT

Activity	<i>a</i>	<i>m</i>	<i>b</i>	<i>t</i>	SD
A	4	5	7	5.17	0.5
B	0.5	1	2	1.08	0.25
C	1	2	2	1.83	0.17
D	2	3	3	2.83	0.17
E	1	2	3	2	0.33
F	1	3	4	2.83	0.5
G	3	4	6	4.17	0.5
H	1	2	3	2	0.33
I	0.5	1	2	1.08	0.25
J	1	1	1.5	1.08	0.08

The degree of uncertainty can be measured by the calculation of standard deviation (SD) which is calculated by using the following formula:

$$SD = \frac{b - a}{6}$$

The SD is proportional to the difference between pessimistic and optimistic activity durations. This measure can be used to rank the activities by calculating the probability of uncertainty of meeting the target date. Figure 4.23 shows the SD of the project. Two standard deviations cannot be added; thus, they are calculated as follows:

$$\sqrt{(SD_1)^2 + (SD_2)^2}$$

The SD for event 2 is 0.5 as it depends only on activity 1. For event 3, there are two possibilities (A + E or B). Thus, the total SD is given by

$$\sqrt{(0.5)^2 + (0.33)^2} = \sqrt{0.25 + 0.1089} = 0.6$$

Similarly, the SD for each of the events is calculated.

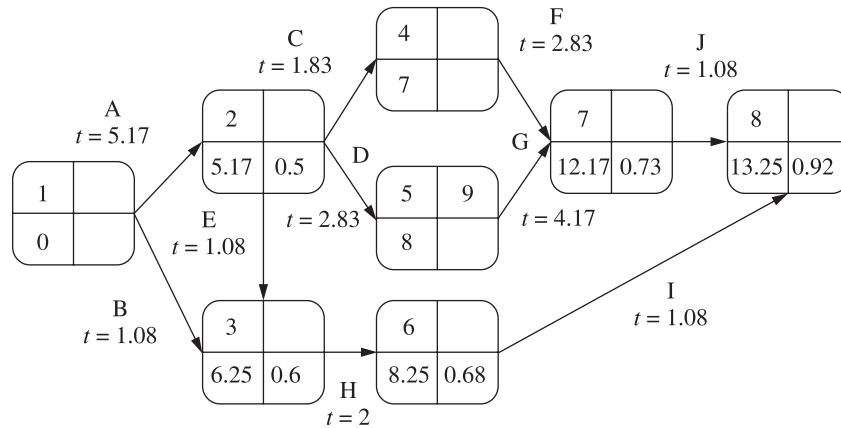


Figure 4.23 Standard deviation.

The PERT uses the following steps to compute the probability of meeting the target durations:

- Calculate the Z value for each event corresponding to its target date.
- Convert the Z values to probabilities of meeting or missing the targets.

The Z value for each event is calculated by using the following formula:

$$Z = \frac{TT - t}{SD}$$

where TT is target time.

If we want to complete the event 5 in 9 months, the Z value for event 5 is:

$$Z = \frac{9 - 8}{0.53} = 1.886$$

By seeing the table of normal distribution, we conclude that there is 0.89 probability of meeting the target date. Thus, there is 11% risk of not meeting the target date at the end of month 9. After calculating the probabilities of meeting the target dates, we can rank the risk by their risk percentage. Using these rankings, we can find out which activity is more uncertain and a cause of concern.

EXAMPLE 4.13 For the following software project, draw the PERT chart and identify the critical path.

Activity	Task description	Prerequisites	Optimistic duration	Most likely duration	Pessimistic duration
1	Identify stakeholders' needs and construct plans	–	1	2	3
2	Cost–benefit analysis	–	2	3	4
3	Construct module A	1	1	2	3
4	Construct module B	2	2	4	6
5	Construct module C	3	1	4	7
6	Testing	3	1	2	9
7	Integration	4, 5	3	4	11
8	Deployment	6, 7	1	2	3

Solution The PERT chart and its corresponding table are given in Figure 4.24 and Table 4.32, respectively.

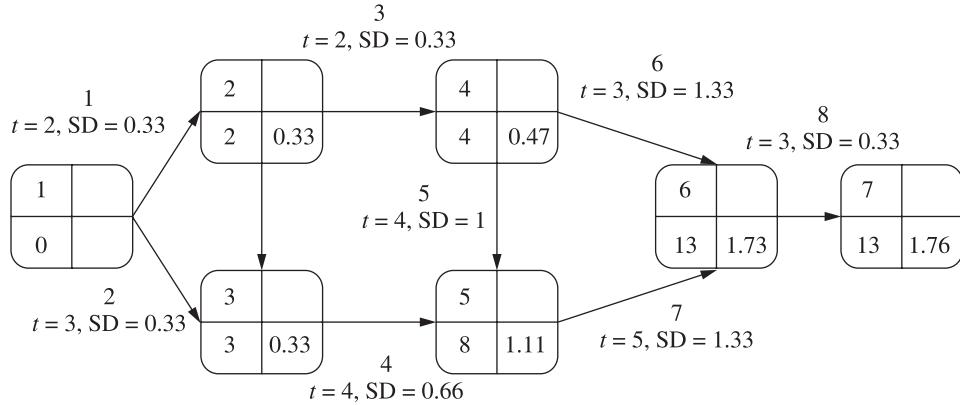


Figure 4.24 PERT chart.

Table 4.32 Expected time and standard deviation for PERT

Activity	a	m	b	t	SD
1	1	2	3	2	0.33
2	2	3	4	3	0.33
3	1	2	3	2	0.33
4	2	4	6	4	0.66
5	1	4	7	4	1
6	1	2	9	3	1.33
7	3	4	11	5	1.33
8	1	2	3	2	0.33

EXAMPLE 4.14 Consider the following data concerning the activities in a project (in weeks):

Activity	Immediate predecessor	a	m	b
A	—	2	4	6
B	—	6	8	10
C	A	1	5	15
D	C	1	5	9
E	B	6	8	10

- (a) Using the PERT, compute the expected time and the standard deviation required to complete each activity.
- (b) Draw a network diagram and find the critical path. What is the expected length of the critical path?
- (c) Assuming that the normal distribution applies, compute the probability that path A-C-D will be completed in at most 16 days.
- (d) How much time must be allowed to achieve a 95% confidence of timely completion?

Solution The PERT chart and its corresponding table are given in Figure 4.25 and Table 4.33, respectively.

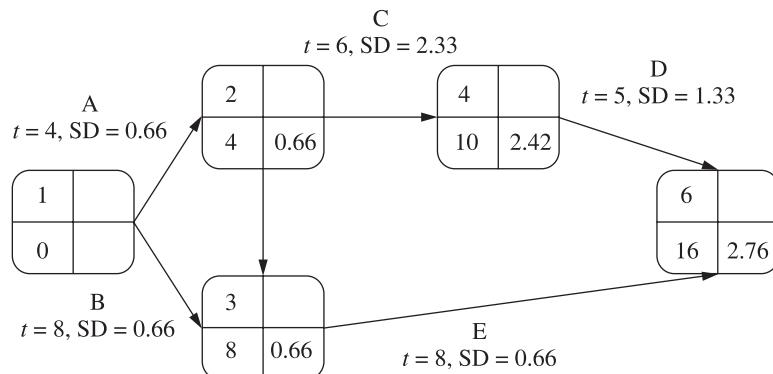


Figure 4.25 PERT chart.

Table 4.33 Expected time and standard deviation for PERT

Activity	a	m	b	t	SD
A	2	4	6	4	0.66
B	6	8	10	8	0.66
C	1	5	15	6	2.33
D	1	5	9	5	1.33
E	6	8	10	8	0.66

- (a) Expected time = 16, SD = 2.76
- (b) Critical path B, E
- (c) $Z = \frac{TT - t}{SD} = \frac{16 - 15}{2.76} = 0.3623$
- Risk = 0%
- (d) To achieve 95% confidence, risk factor = 5%

$$\begin{aligned} Z &= \frac{TT - t}{SD} \\ \frac{TT - 16}{2.76} &= 1.75 \\ TT &= 2.76 \times 1.75 + 16 = 20.83 \end{aligned}$$

Review Questions

1. What is the need for object-oriented software estimation techniques?
2. Describe any two object-oriented size estimation techniques.
3. An application consists of 20 scenario scripts and requires 17 person days to implement. Determine the effort of the given application.
4. Describe the steps in the use case points method to calculate effort. Why is the use case points method becoming acceptable in the industry?
5. Compute the value of the use case points for a project with the following information domain characteristics:

Number of actors: 3

Number of use case: 8

Assume all complexity factors to be average.

6. What metrics can be used to estimate the size of an object-oriented software product? How is the use case points metric advantageous over the function point metric? Explain.
7. Describe the use case points method with a suitable example.
8. Describe the class point method in detail.
9. Explain the type of classes identified in the class point method. Identify the type of classes for result management system.
10. What is the purpose of CP₁ and CP₂ measures? In which software development life cycle phase can they be calculated?

11. Consider a project with the following parameters:

Class type	Number of classes	NEM	NSR	NOA
PDT	3	6	2	4
		4	5	5
		10	15	3
		5	7	2
		7	10	1
HIT	5	5	8	7
		8	12	8
		6	9	4
		2	4	5
		4	2	3
DMT	4	2	3	2
		1	0	2
		3	4	4
TMT	3	4	8	1
		8	12	

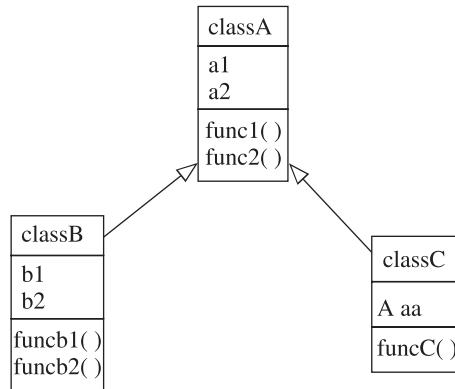
Assume all the technical complexity factors have average influence. Calculate CP₁, CP₂ and effort using class point method.

12. Compare and contrast function point method and object-oriented function point method.
13. Explain all the element types that are used to count object-oriented concepts in the object-oriented function point method.
14. Describe the steps for calculating object points.
15. Establish the relationship between function point method and object-oriented function point method.
16. Discuss the method for calculating object-oriented concepts in object point method.
17. Assume that a project has five classes with DET and RET values given below:

Internal class	DET	RET
C ₁	4	2
C ₂	5	7
C ₃	3	1
C ₄	2	1
External class	DET	RET
C ₅	6	10

The above classes contain four methods with high complexity and the complexities of the adjustments factors are average. Compute object points.

18. Consider the following class model. Determine DET and RET for each class.



19. Consider the example given in Question 18. Calculate unadjusted object points. Assume all the services to be of average complexity.
20. What is risk? Describe the various risk ratings.
21. Describe the framework for managing risks.
22. What are the risk management activities? How can risks be prioritized?
23. Consider the result management system. Analyse and prioritize the various kinds of risks associated with such a project.
24. Discuss the risk list given by Boehm.
25. What is risk exposure? Describe the method to prioritize risks.
26. Explain the steps in constructing a CPM network model with an example. Explain the importance of critical path.
27. How can PERT be used in risk analysis? How can we identify uncertainties in activities using the PERT chart?
28. Explain with examples the importance of PERT and CPM charts.
29. Draw the CPM network for the project specification given in the table below. Explain each step of the network in detail. Identify the critical path.

Activity	Duration (weeks)	Precedent
A	8	
B	4	
C	6	
D	8	A
E	3	C
F	2	
G	4	C, E
H	5	B, F

30. Draw the CPM network for the project specification given in the table below. Explain each step of the network in detail. Show the critical path on your CPM chart. List the tasks that are on the critical path.

Activity	Activity name	Duration (weeks)	Immediate predecessor
1	Obtain requirements	4	–
2	Analyse operations	4	–
3	Define subsystems	2	1
4	Develop database	4	1
5	Make decision analysis	3	2
6	Identify constraints	2	5
7	Build module 1	8	3, 4, 6
8	Build module 2	12	3, 4, 6
9	Build module 3	18	3, 4, 6
10	Write report	10	6
11	Testing and integration	8	7, 8, 9
12	Implementation	2	10, 11

31. What do you mean by risks in software projects? How does risk management tackle these risks? What procedure is usually followed?
32. What do you understand by a “critical path” in a project schedule? Bring out the importance of software project estimation in the context of software project management.
33. How can PERT be used to evaluate the effects of uncertainty? Explain with an example.
34. Consider the following data for a given project. Draw the PERT chart using the given information. For the due date 20, 24 and 25, calculate the percentage of risks.

Activity	Predecessor	Optimistic	Most likely	Pessimistic
A	–	4	5	9
B	–	3	6	15
C	A	3	4	5
D	A	2	7	9
E	B, C	3	5	7
F	D	6	8	13
G	D, E	1	3	11

35. You are given the following data concerning the activities in a project (numbers refer to weeks):

Activity	Immediate predecessor	a_j	m_j	b_j
A	—	5	7	15
B	-	6	15	18
C	A	6	10	14
D	B, C	3	5	7
E	B, C	4	12	14
F	D	4	5	6

- (a) Using the standard PERT approximation formulas, compute the expected time and the variance required to complete each activity.
- (b) Draw a network diagram and find the critical path by inspection. What is the expected length of the critical path?
- (c) Assuming that the normal distribution applies, compute the probability that the critical path will take between 25 and 32 days to complete.
- (d) How much time must be allowed to achieve a 95% confidence of timely completion?

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. After the requirements have been gathered, the customer may like to estimate:
 - (a) Cost
 - (b) Coupling
 - (c) Inheritance
 - (d) Polymorphism
2. In object-oriented systems, size can be estimated by:
 - (a) Use cases
 - (b) Objects
 - (c) Classes
 - (d) All of the above
3. Which one is **not** an object-oriented software estimation method?
 - (a) Use case point
 - (b) Lorenz and Kidd
 - (c) Function point
 - (d) Class point
4. The use case points method was developed by:
 - (a) B. Boehm
 - (b) V. Basili
 - (c) G. Karner
 - (d) A. Albrecht
5. Use case points can be calculated by:
 - (a) UUCP \times TCF \times ECF
 - (b) UUCP \times TCF
 - (c) UCP \times H
 - (d) UUCP \times ECF
6. The use case points method is based on:
 - (a) Classes
 - (b) Objects
 - (c) Scenarios
 - (d) Use cases
7. The use case points method can be used in:
 - (a) Early phases
 - (b) Later phases
 - (c) Effort estimation
 - (d) Both (a) and (c)

8. TCF stands for:
- (a) Technological complexity factor
 - (b) Technical complexity factor
 - (c) Technical class factor
 - (d) Total complexity factor
9. The class point method was developed by:
- (a) G. Karner
 - (b) G. Costagliola
 - (c) B. Beizer
 - (d) B. Boehm
10. The class point method involves identification of types of cases. How many types of classes are identified?
- (a) 3
 - (b) 2
 - (c) 4
 - (d) 5
11. In the class point method, how many measures can be calculated?
- (a) 1
 - (b) 2
 - (c) 3
 - (d) 4
12. In the class point method, the number of technical complexity factors is:
- (a) 10
 - (b) 14
 - (c) 20
 - (d) 18
13. In the calculation of CP₁, which measures are used?
- (a) NEM and NOA
 - (b) NOA and NSR
 - (c) NEM and NSR
 - (d) NEM, NSR and NOA
14. In the calculation of CP₂, which measures are used?
- (a) NEM and NOA
 - (b) NOA and NSR
 - (c) NEM and NSR
 - (d) NEM, NSR and NOA
15. In the CP₁:
- (a) Initial estimate of size is made
 - (b) Detailed estimate of size is made
 - (c) The number of attributes is determined
 - (d) Both (a) and (c)
16. Object-oriented function point can be mapped to:
- (a) Use case points method
 - (b) COCOMO model
 - (c) Putnam resource allocation model
 - (d) Function point method
17. In services, which measures are calculated?
- (a) DET and RET
 - (b) DET
 - (c) RET and FTR
 - (d) DET and FTR
18. In the object point method, external classes can be mapped to _____ in the function point method.
- (a) External interface files
 - (b) External inquiries
 - (c) External outputs
 - (d) External inputs
19. A DET is counted for:
- (a) Simple attributes
 - (b) Class
 - (c) Method
 - (d) Method argument

Further Reading

The report by David N. Card focuses on object-oriented software measurement which can be found in:

Card, D., Emam, K. and Scalzo, B., Measurement of Object-oriented Software Development Projects, <http://sern.ucalgary.ca/courses/SENG/693/F00/readings/OO%20Measurement%20-%20March%202001.pdf>, 2001.

The following research papers provide a framework on estimating use case points:

Banerjee, G., Use Case Points: An Estimation Approach, http://www.bfpug.com.br/Artigos/UCP/Banerjee-UCP_An_Estimation_Approach.pdf, 2001

Roy, K. Clemons, Project estimation with use case points. *CROSSTALK The Journal of Defense Software Engineering*, February 2006.

Ferens, D., Fischman, L., Fitzpatrick, T., Galorath, D., and Tarbet, D., *Automated Software Project Size Estimation via Use Case Points*. El Segundo: Galorath Incorporated, 2002.

A detailed explanation of use case point method with example case study may be obtained from:

Kirsten, R., Estimating Object-Oriented Software Projects with Use Cases. Master of Science, Thesis 7th November 2001.

Other useful research papers may be:

Minkiewicz, A., Measuring object-oriented software with predictive object points. Proc. ASM'97—Applications in Software Measurement, Atlanta, 1997.

Moser, S., Measure and Estimation of Software and Software Processes. PhD Thesis, University of Berne, Switzerland, 1996.

Antoniol, G., Lokan, C., Caldiera, G., and Fiutem, R., A function point-like measure for object-oriented software. *Empirical Software Engineering*, 4(3): 263–287, 1999.

An excellent tutorial on risk management is given by Boehm in:

Boehm, B., *IEEE Tutorial on Software Risk Management*. New York: IEEE Computer Society Press, 1989.

Chapter 6 of the book by Galorath and Evans discusses the measures for estimating object-oriented projects:

Galorath, D.D. and Evans, M.W., *Software Sizing, Estimation, and Risk Management*. Philadelphia, PA: Auerbach Publications, 2006.

Hughes and Cotterell provide an introduction to risk management in Chapter 7 of the book:

Hughes, B. and Cotterell, M., *Software Project Management*. New Delhi: Tata-McGraw Hill, 2009.

5

Object-Oriented Analysis

After requirements have been captured, the analysis of the identified requirements begins. The process of analysis involves construction of logical structure of a system that can be maintained during the software development life cycle. Object-oriented analysis (OOA) identifies and defines the real-world objects that are involved in interaction with the system. It also identifies the objects that can be used to provide alternative solutions to the problem. The real-world objects are defined in terms of classes, attributes and operations. These objects, thus, can be mapped into objects in object-oriented languages easily. The OOA constructs a robust and ideal model that is free from the details of implementation environment. Hence, the OOA is the process of defining the problems in terms of real-world objects. It gives a high level of understanding of possible solution to the requirements identified.

The aim of this chapter is to extract classes and attributes. The types of classes are identified and relationships amongst the identified classes are discovered. Finally, the operations and attributes in the classes are defined. All these concepts are explained along with a real-life case study of LMS.

5.1 Structured Analysis versus Object-Oriented Analysis

Structured analysis (SA) involves converting the requirements into specifications. Pressman (2005) defines structured analysis as:

Structured analysis (SA) takes a distinct input-process-output view of requirements. Data are considered separately from the processes that transform the data. System behavior, although important, tends to play a secondary role in structured analysis. The structured analysis approach makes heavy use of functional decomposition (partitioning of the data flow diagram).

The SA involves top-down decomposition of the system into modules. It centres around the creation of process model using data flow diagram (DFD) and design of data model using

entity-relationship (ER) model. The DFD depicts the flow of data through the system. It shows the processes that need to be designed and finally developed. The focus of the DFD is on functions rather than on data. However, the data can be modelled using ER models. The disadvantages of function-oriented and data-oriented techniques are that the functions are given more importance than the data and the data and functions are modelled separately. Data and functions are collective activities, i.e. the data can be modified through functions.

In the OOA phase, the class modelling techniques are used which combine the data and functions into a single class. Thus, data and functions are given equal importance. The attributes in a class can only be accessed by the operations in the classes which prevent the accidental modification of data (attributes). The data are known as attributes and the functions are called operations. These are easily modifiable and maintainable, and can be directly refined in object-oriented design. In the OOA phase, relationships between classes are depicted through visual diagrams. The purpose of such diagrams is to:

- understand, extract and define the customer's requirements.
- serve as a basis for development of the software. Based on these diagrams, it would be easier to develop the database schema of the system and partially identify its operations.

5.2 Identification of Classes

A class is a collection of objects, with common attributes and operations. The objects with common attributes and operations have similar relationship to the class. Hence, a class is a template that groups attributes and operations together, and is used to create objects. Identification of classes is not a simple task. In the rational unified process, classes are categorized into three types: entity class, interface class and control class. These three types of classes allow the analyst to separate the functionality of the system and simplify the identification process. The OOA process is iterative. Hence, we keep on modifying the list and structure of classes, as the analysis and design progresses.

5.2.1 Entity Classes

Entity classes include those classes that are going to persist longer into the system. These are the classes which have to be stored and maintained for a longer period, sometimes for the lifetime of the system. Figure 5.1 shows the notation for representing entity classes.

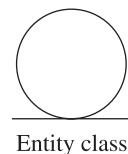


Figure 5.1 Notation for an entity class.

Entity classes contain information needed to accomplish a given task. These may be identified by extracting the information that is required to complete a task from the use case description. Entity classes are also called *domain classes*. Consider the 'issue book' use case of

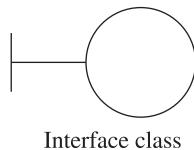
the LMS case study given in Chapter 3. The purpose of the use case is to allow the members (student/faculty/employee) to issue book from the university library. The use case requires the information of member, book and transaction. Hence, we identify three entity classes:

- Member
- Book
- Transaction

The instances of entity classes are known as entity objects. Entity classes may not be specific to one use case. For example, in the LMS, student class may be included in ‘issue book’ use case as well as in ‘maintain student details’ use case. The attributes and operations of an entity class may be identified by the actor of the use case. Entity classes represent the key attributes of the system that is to be developed. For instance, in an ATM system, customer and account are some entity classes and in a result management system, students, marks and courses are entity classes. The major source of information for extracting entity classes is the data fields of interface screens described in the SRS document.

5.2.2 Interface Classes

Interface classes handle the interaction in the system. They provide interface between the actor (human being or external system) and the system. These classes are known as *boundary classes*. These classes are used to model windows, buttons, communication protocols, etc. Usually, the life of the interface classes is only as long as the use case exists. Figure 5.2 shows the notation for representing interface classes.



Interface class

Figure 5.2 Notation for an interface class.

The interaction of actor with each use case scenario is examined to extract the interface classes. The interface classes represent the number of interfaces required by an actor to interact with all the paths in the use cases. For example, in the LMS, in order to issue a book, an interface is required. The actor library staff interacts with ‘issue book’ use case. This means the student/faculty/employee must be able to interact with something to issue book. A class containing all the options to enable the actor ‘library staff’ issue book will satisfy the given requirement. This class is called IssueBookInterface. BarcodeReader class will also be required to get the bar code details of the book and the library member. Unlike entity and control classes, interface classes are dependent on the surroundings of the system. If there is a change in the interface, then the interface class should change but the entity and control classes will remain unaffected.

5.2.3 Control Classes

Control classes are responsible for coordinating and managing entity and interface classes. The functionality that could not be placed in either of entity and interface classes may be placed in

a control class. The control class is used to put together things so that a use case is completed. Control classes represent the dynamics of the system and handle the tasks and sequence of events. Figure 5.3 shows the notation for representing control classes.

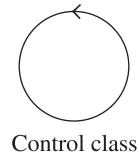


Figure 5.3 Notation for a control class.

A control object (an instance of control class) is created when the use case starts and it ends when the use case finishes. Control classes may be identified from the use cases. One control class may be assigned to one use case. These classes are used to manage and coordinate the flow of events needed to accomplish the functionality of a given use case. If the control class is doing more than coordinating the sequence of events in the use case, then it must be reconsidered. For example, in the LMS, the data entry operator adds a student. The information about the procedure of addition of the student must be available with student class rather than StudentManager class. Now we will identify control classes for ‘issue book’ use case in the LMS. A control class can be added to handle the flow of events for ‘issue book’ use case. This class is called IssueBookController.

Some use cases in the system (specifically data manipulation use cases) may function without any control class. Examples of control classes include TransactionManager, LoginController, SecurityManager, ExceptionHandler. The control classes allow the entity classes to be more independent so that they can be used across multiple use cases. Control classes have the following properties:

1. They are independent of their surroundings.
2. They are used to control the flow of events.
3. The changes required in the control class are due to the changes in the entity classes.
4. They may not perform in the same manner each time they are executed.

Thus, the classes identified for ‘issue book’ use case in the LMS are summarized in Table 5.1 and their graphical notations are shown in Figure 5.4.

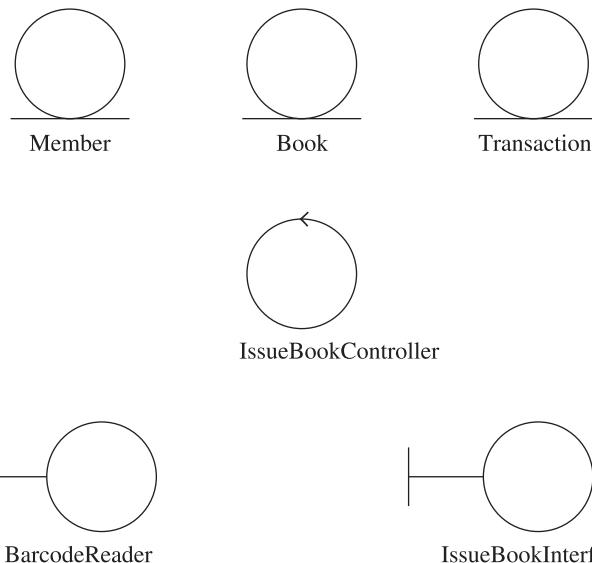
Table 5.1 Summary of classes in ‘issue book’ use case

Class name	Class type	Description
Member	Entity	This class is used to represent the information of the members (student/faculty/employee) in the university.
Book	Entity	This class is used to represent the information of the books in the university library.
Transaction	Entity	This class is used to represent the information of the transactions to the members in the library.

(Contd.)

Table 5.1 Summary of classes in ‘issue book’ use case (*Contd.*)

Class name	Class type	Description
IssueBookInterface	Interface	This class provides interface between the actor and the system.
BarcodeReader	Interface	This class reads bar code of books and members of the library.
IssueBookController	Control	This class manages and controls the operations in the ‘issue book’ use case.

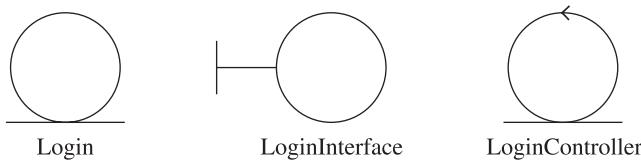
**Figure 5.4** Classes for ‘issue book’ use case.

EXAMPLE 5.1 Consider the case study of LMS, given in Chapter 3, Section 3.1. Identify entity classes, interface classes and control classes for ‘login’ use case.

Solution In the login use case, one entity class ‘login’ containing the login details will be required. The summary of classes identified is given in Table 5.2 and these are diagrammatically depicted in Figure 5.5.

Table 5.2 Summary of classes in ‘login’ use case

Class name	Class type	Description
Login	Entity	This class is used to represent the login information of the operators of the system.
LoginInterface	Interface	This class provides login interface between the actor and the system.
LoginController	Control	This class manages and controls the operations in the ‘login’ use case.

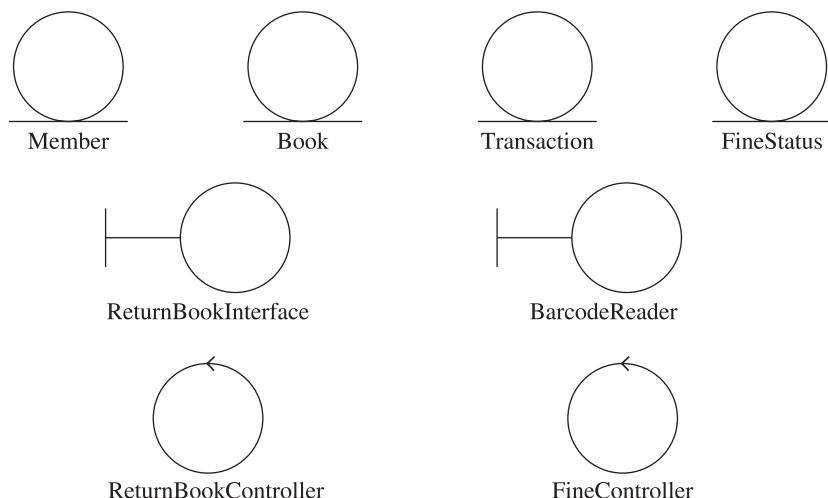
**Figure 5.5** Classes for ‘login’ use case.

EXAMPLE 5.2 Consider the case study of LMS, given in Chapter 3, Section 3.1. Identify entity classes, interface classes and control classes for ‘return book’ use case.

Solution In the return book use case, three entity classes, two control classes and one interface class will be required. The summary of classes identified is given in Table 5.3 and they are diagrammatically depicted in Figure 5.6.

Table 5.3 Summary of classes in ‘return book’ use case

Class name	Class type	Description
BarcodeReader	Interface	This class reads bar code of books and members of the library.
Member	Entity	This class is used to represent the members of the library.
Book	Entity	This class is used to represent the information of the books in the university library.
Transaction	Entity	This class is used to store the information of the transactions to the members in the library.
ReturnBookInterface	Interface	This class provides return book interface to the actor of the system.
ReturnBookController	Control	This class manages and controls the operations in the ‘return book’ use case.
FineCalculator	Control	This class manages and controls the operations in the ‘fine calculation’ use case.
FineStatus	Entity	This class stores the fine amount and the status of the fine.

**Figure 5.6** Classes for ‘return book’ use case.

5.3 Identification of Relationships

If we are constructing a class room of a college, it will consist of walls, windows, doors, fans, tube lights, projector, black board, etc. All of these things are related to each other. Walls are connected to each other. Doors are fixed inside the walls to provide way to people to enter the room. Windows are fixed for light or air purpose. Fans and tube lights are fixed on the ceilings of the wall to provide cooling and reduce darkness. Thus, all these things together form a class room. These things have different kinds of relationships amongst them. Similarly in OOA, the classes can connect to each other having different kinds of relationships. These relationships provide collaboration amongst classes.

A system consists of classes and objects. The system model depicts the communication amongst classes. The instances of a class communicate through sending messages. For example, the book is issued after the issue book message is received by the issue book object. Thus, a message is sent from one object to another. A relationship provides a channel through which instances of classes communicate with each other. They represent connection amongst instances of classes. There are five types of relationships amongst objects: association, aggregation, composition, dependency and generalization. Relationships along with their notations are summarized in Table 5.4.

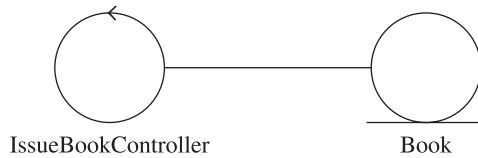
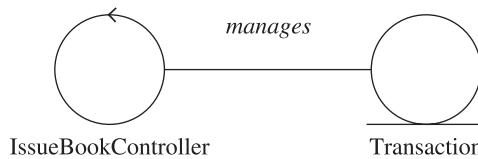
Table 5.4 Relationships among objects

Relationship	Description	Notation
Association	It provides structural connections between instances of classes.	———
Aggregation	It provides whole-part kind of relationship between classes.	◇ —————
Composition	It provides strong aggregation between classes.	◆ —————
Dependency	It signifies that changes in one class affect the other class.	----->
Generalization	It signifies parent-child relationship amongst classes.	————→

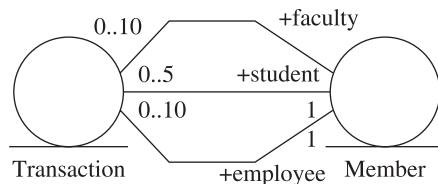
5.3.1 Association

Association is a **structural connection** between classes. This type of relationship is **mostly bidirectional**. In another term, the association relationship provides a link between different objects of the classes in the relationship. For example, there is a connection between IssueBookController class and Book class. This implies that the objects of IssueBookController class are linked with the objects of Book class. **Binary associations are used to connect exactly two classes**. The association relationship is represented by a solid line between the participating classes as shown in Figure 5.7.

An association relationship may be associated with a name. The name describes the type of relationship between two classes. For example, IssueBookController class manages Transaction class, as shown in Figure 5.8.

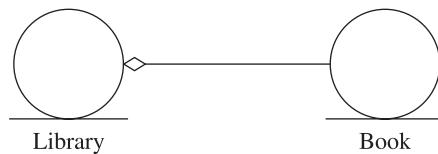
**Figure 5.7** Association relationship.**Figure 5.8** Association names.

When two classes are in association relationship, a role may also be associated between them. A role describes the function a class plays in an association. For example, a member may play the role of a student, faculty or employee when it is associated with Transaction class (see Figure 5.9). On the end of each association shown in Figure 5.9, multiplicity (refer to Section 5.3.3) is specified.

**Figure 5.9** Association roles.

5.3.2 Aggregation

The aggregation relationship models the **whole-part type of relationships**. It depicts that a class **is a part of another class**. Some operations in the whole class may be applied to the other class. Aggregation represents '**has-a**' relationship. The notation used for aggregation relationship is a line with a diamond at the end, i.e. to the class denoting the whole. For example, in the LMS, a book may be issued to a student/faculty/employee. The relationship between the Book class and the Library class is aggregation which means that the Book class is a part of the Library class as shown in Figure 5.10.

**Figure 5.10** Aggregation relationship between library and book.

5.3.3 Multiplicity

We must specify for an object that **how many objects are related at the other end of an association**. Multiplicity represents the number of instances of classes related to one instance of another class. It depicts how many objects of a class are related to one object of another class at a given time. Multiplicity can be specified in association and aggregation relationships. Multiplicity is expressed by a range of values as shown in Figure 5.11.

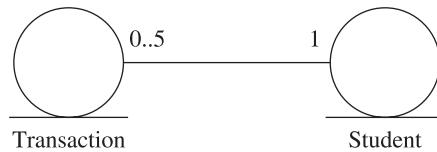


Figure 5.11 Multiplicity indicator of transaction and student association.

When the multiplicity is stated at one end of an association, it specifies the number of objects that must be there for each object of the class at the opposite end. Multiplicity can be shown as many ($0..*$), one or more ($1..*$) and exactly one (1). Hence, multiplicity is denoted as:

Minimum_value..maximum_value

Some commonly used multiplicity indicators are given as follows:

1	one
$0..*$	many
$1..*$	one or more
0..1	zero or one
3..5	specific range (3, 4 or 5)

The relationship shown in Figure 5.11 can be read in the following ways:

1. One transaction object is related to exactly one student. For example, B101 book is related to student Ram (a student object).
2. One student object may be related to zero to five transaction objects. For example, Ram (a student object) is related to B101, B105 and B111 (the student has three books issued). Since the range of multiplicity is zero to five, a minimum of zero and maximum of five books may be issued to a student.

Control classes usually depict one-to-one multiplicity (Boggs and Boggs, 2002). As the application is executed, only one IssueBookController object may be required.

5.3.4 Composition

Composition relationship represents a **strong form of the whole-part relationship**. When two classes are having composition type of relationship, the **part class belongs to only the whole class**. For example in a university, the departments belong to only the specific university. The notation used for composition relationship is a line with a filled diamond at the end, i.e. to the class denoting the whole as shown in Figure 5.12.

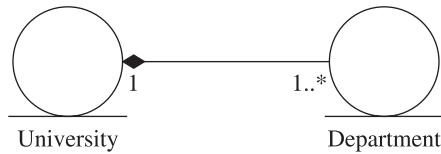


Figure 5.12 Composition relationship.

5.3.5 Dependency

The class referring another class is represented through dependency relationship. Hence, the change in the class being referenced affects the class using it. The dependency relationship is graphically depicted by a directed dashed line.

Dependency relationship is unidirectional. It exists when the instance of a class is either passed as a parameter variable in a method or used as a local variable in a method of another class.

When a system is designed, the aim is to minimize dependency relationships amongst classes. It increases complexity and decreases maintainability of the system. For example, a university maintains the record of programmes running in a school. We may consider a SchoolSchedule class which is dependent on programmes of the university as it uses the object of programme class as parameter to its member ‘add’. Thus, the SchoolSchedule class is dependent on programme class as shown in Figure 5.13.



add(prog : Programme)

Figure 5.13 Dependency relationship.

5.3.6 Generalization

Generalization is a relationship between the **parent class and the child class**. This relationship depicts inheritance relationships. Generalization relation is also known as ‘is-a’ relationship. In this relationship, the child inherits all the properties of its parents but vice versa is not true. The child also consists of its own attributes and operations. When the operation of a child has the same name and signature as of its parents, it is known to **override the operation of the parent class**. This concept is known as **polymorphism**. The graphical notation of generalization relationship is shown as a solid line with an open arrow head that points towards the parent class.

A class can have zero or more parents. When a child inherits features from more than one class, it is known as multiple inheritance. Thus, in multiple inheritance a class has more than one parent class.

For example, in the LMS, Student, Faculty and Employee are all members of a library. They may have their own attributes but have some attributes in common which are placed in the base

class Member. Figure 5.14 depicts the generalization relationship. Here, Student, Faculty and Employee entity classes are children of the Member class.

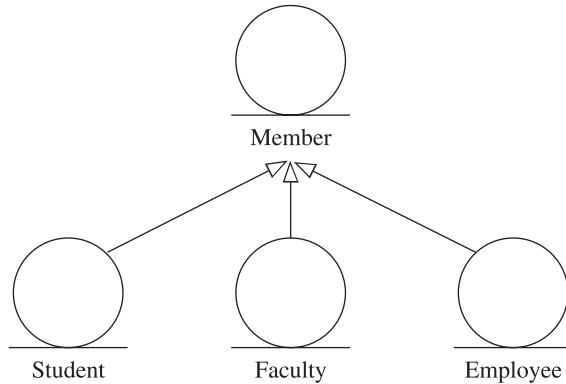


Figure 5.14 Inheritance relationship.

5.3.7 Modelling Relationships

When generalization and dependency relationships are modelled, the classes are not at equal level of importance. In the dependency relationship, one class is using the other class, but the other class has no knowledge of it. Similarly in the case of inheritance relationship, the base class does not have any idea about its derived classes. However, when association relationships are modelled, the two participating classes are at equal level of importance. Aggregation is a type of association that depicts whole–part relationships, inheritance depicts ‘is-a’ relationship and dependency depicts ‘using’ relationship. In order to identify structural relationships, the following may be seen:

- If there is an association between pairs of classes, the association relationship is used.
- If one object of a class passes messages to the other object other than through parameters of the method, then the association relationship is used.
- If a class is a whole and other classes are its parts, then an aggregation relationship is modelled. In other words, when a class is made up of other classes, the aggregation type of relationship is used.

The structural relationships can be identified by the use case descriptions explained in Chapter 3. Figure 5.15 models association relationships between classes involved in ‘issue book’ use case of the LMS and Table 5.5 shows their summary. The portion from the use case description of issue book use case is given as follows:

Basic Flow

1. Student/faculty/employee membership number is read through the bar code reader.
2. The system displays information about the student/faculty/employee.
3. Book information is read through the bar code reader.
4. The book is issued for the specified number of days and the return date of the book is calculated.
5. The book and student/faculty/employee information is saved into the database.

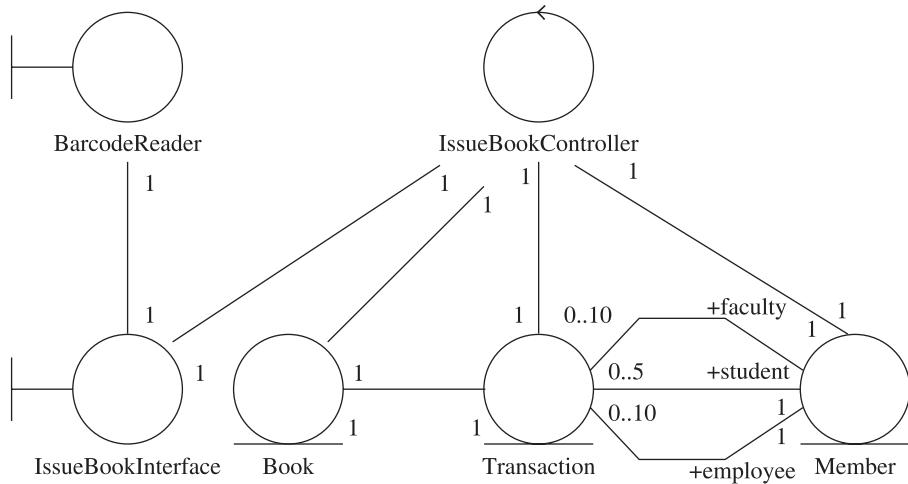


Figure 5.15 Relationships between classes in ‘issue book’ use case.

Table 5.5 Summary of relationships shown in Figure 5.14

Sending class	Receiving class	Relationship
BarcodeReader	IssueBookInterface	Bidirectional Association
IssueBookInterface	IssueBookController	Bidirectional Association
IssueBookController	Book	Bidirectional Association
IssueBookController	Transaction	Bidirectional Association
IssueBookController	Member	Bidirectional Association
Book	Transaction	Bidirectional Association
Member	Transaction	Bidirectional Association

Hence, all the relationships identified in ‘issue book’ use case are bidirectional association.

EXAMPLE 5.3 Consider Example 5.1 and identify the relationships amongst classes for ‘login’ use case.

Solution The relationships amongst various classes for ‘login’ use case are shown in Figure 5.16

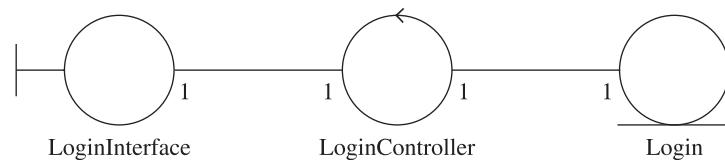


Figure 5.16 Relationships between classes in ‘login’ use case.

EXAMPLE 5.4 Consider Example 5.2 and identify relationships amongst classes for ‘return book’ use case.

Solution The relationships amongst classes in ‘return book’ use case are shown in Figure 5.17 and these relationships are summarized in Table 5.6.

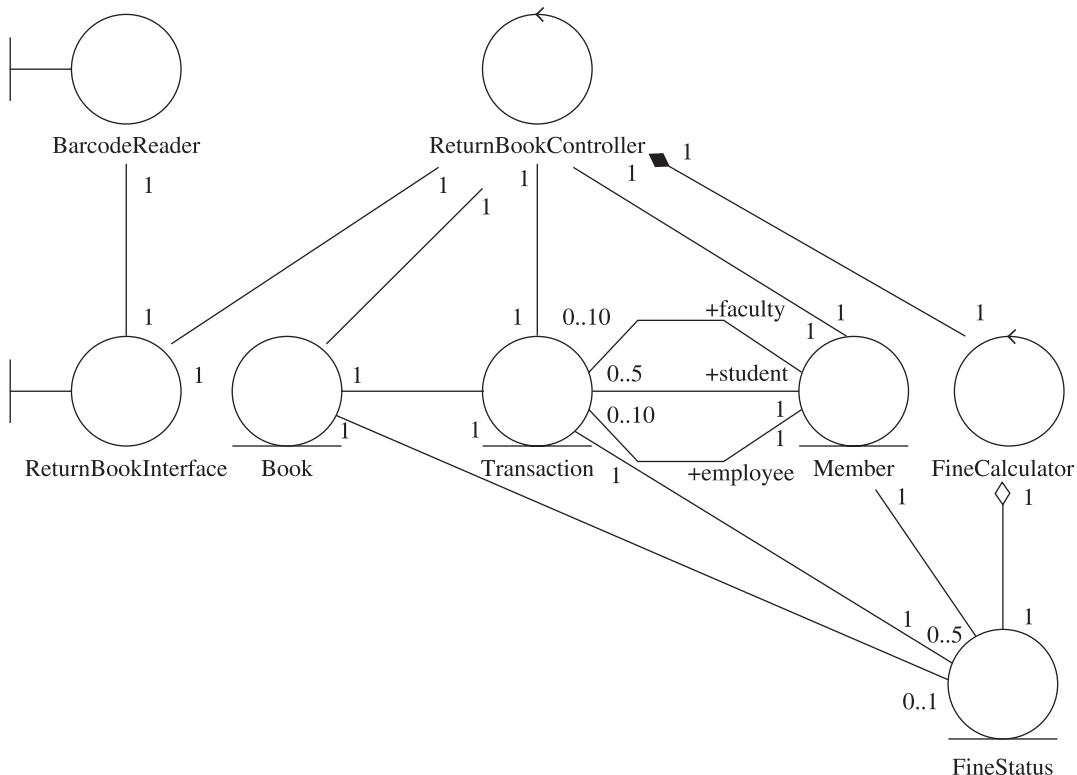


Figure 5.17 Relationships between classes in ‘return book’ use case.

Table 5.6 Summary of relationships shown in Figure 5.17

Sending class	Receiving class	Relationship
BarcodeReader	ReturnBookInterface	Bidirectional Association
ReturnBookInterface	ReturnBookController	Bidirectional Association
ReturnBookController	Book	Bidirectional Association
ReturnBookController	Transaction	Bidirectional Association
ReturnBookController	Member	Bidirectional Association
Book	Transaction	Bidirectional Association
Member	Transaction	Bidirectional Association
FineCalculator	ReturnBookController	Composition
FineCalculator	FineStatus	Aggregation

5.4 Identifying State and Behaviour

Classes **encapsulate both attributes and operations** that operate on those attributes, into a single unit. An attribute of a class represents data definition and is used to store information in the objects. For example, in a member class memberID, name, dateOfBirth, phone, email, date of validity of membership, etc. are the attributes.

An operation must be well defined with a single goal. For example, the member class must be able to add, delete, update and view the details of a member. These requirements are represented by four operations—addMember, updateMember, deleteMember and viewMember. All the instances of the class must be able to perform all these four operations.

In the LMS case study, we begin each attribute and operation with a lowercase letter and if the attribute or operation consists of multiple words, then the first letter of each word is capitalized. The operations of a class must be cohesive and a class should not contain unnecessary attributes and operations. If an attribute or operation is not useful by an object, then it must not be included in the class. For example, a student may contain an attribute programme but the student is only concerned with single programme. The programmes must be managed by a different school class. The procedure for identification of attributes and operations is described in subsequent sections.

5.4.1 Attributes

Attributes represent the information to be stored. The entity objects may contain many attributes. Attributes can be identified from the use case description. For example, add a book subflow of Maintain Book Details use case is given as follows:

Basic Flow 1: Add a book

The system requests that the administrator/DEO enter the book information. This includes:

- Book bar code ID
- Accession number
- Subject descriptor
- ISBN
- Title
- Language
- Author
- Publisher

Once the administrator/DEO provides the requested information, the book is added to the system.

This subflow shows that the Book entity class must contain the attributes such as book bar code ID, accession number, subject descriptor, ISBN, title, language, author, and publisher. This information needs to be stored in an entity class Book which is required to keep the details of the number of books in a library. Hence, the attributes can be discovered by extracting the elements that are needed from the use case description. They can also be identified by the class description and from the expertise of the analyst. In addition to the attributes identified by use case description of subflow ‘add a book’, one additional attribute that keeps the record of the

issue status of the book must be kept. Scope and domain expertise are used to determine this type of attribute. This attribute may not be alive in the use case but is required to keep track of the issue status of the book. The UML representation of an entity class Book is shown in Figure 5.18 along with its attributes. Each attribute consists of a data type such as integer, real or string as shown in Figure 5.18. An attribute may also be a complex data type.

<<entity>> Book
bookBarcodeID : Integer accessionNo : Integer subjectDescriptor : String ISBN : Long bookTitle : String language : String authorName : String publisher : String issueStatus : Boolean

Figure 5.18 Attributes in a Book class.

The attributes in the Book class can also be identified from section 3.1.1 user interfaces of the SRS. The portion of the details of the book form from the SRS is given as follows:

Various fields available on this form will be:

- **Book bar code ID:** Numeric
- **Accession number:** Numeric and will have value from 10 to 99999.
- **Subject descriptor:** Will display all the subject descriptors.
- **ISBN:** Numeric
- **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
- **Language:** Will display all the languages available.
- **Author:**
 - ◆ **First name:** Alphanumeric of length 3 to 50 characters. Special characters are not allowed. Numeric data will not be allowed.
 - ◆ **Last name:** Alphanumeric of length 3 to 50 characters. Special characters are not allowed. Numeric data will not be allowed.
- **Publisher:** Alphanumeric of length 3 to 300 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.

The above portion of the SRS shows the description of the fields of user interface ‘book form’. These fields need to be stored, hence should be converted as attributes of Book class in the OOA phase.

Hence, there are four methods to identify the attributes in a class:

1. The information to be stored can be extracted from the use case description.
2. The fields in the user interfaces given in the SRS document can be used to extract the attributes.

3. The database schema, if designed, can be used to retrieve the attributes. The fields in the tables can be directly mapped to attributes in classes. The attributes of the table can be mapped directly as attributes in entity classes. These attributes will be stored in the relational database in future.
4. The domain expertise of the analyst can be used to extract the attributes.

Entity classes will contain maximum number of attributes; however, control classes may contain a very few attributes. The class with no operation should be converted to an attribute. All the attributes identified in the entity classes will be later stored in the database.

5.4.2 Operations

It is usually difficult to identify the operations that a class will contain. The operations of a class can be identified by the flow of events given in the use case description. The instance of a class can only be modified through the operations. Interaction diagrams described in Chapter 6 can be used to identify the operations. However, operations can also be identified in the OOA phase. These operations may be changed according to the needs in the design phase. Everything connected to the class should be placed in it. The operations that must be added in an entity class must include:

1. Storing, updating and retrieving information
2. Behaviour that is included in entity class.

For example, the Book class consists of four operations for storing, updating, deleting and retrieving information: addBook(), updateBook(), deleteBook() and viewBook(). In order to retrieve the status of the book, getIssueStatus() operation is also identified. An operation may be identified when an object sends a message to another object. The updated description of the class Book with operations is shown in Figure 5.19.

<<entity>> Book	
	bookBarcodeID : Integer accessionNo : Integer subjectDescriptor : String ISBN : Long bookTitle : String language : String authorName : String publisher : String issueStatus : Boolean
	addBook() deleteBook() updateBook() viewBook() getIssueStatus()

Figure 5.19 Operations in a Book class.

5.4.3 Example: Issue Book in LMS

The attributes and operations may be identified by the steps given in the flow of events of the use case description. The attributes are identified by reading nouns from flow of events of ‘issue book’ use case given in the use case description. The following line specifies that date of issue and date of return need to be stored. Hence, these attributes are added to Transaction class. The information about a specific book issued to a specific student is also required.

The book is issued for the specified number of days and the return date of the book is calculated and stamped on the book.

The below line shows that issue status and number of books issued to a student need to be saved in the database. Hence, issueStatus is identified as an attribute of Book class and noIssuedBooks is identified in Member class.

The book and student/faculty/employee information is saved into the database.

Operations are identified using messages which are sent across the instances of the classes. For example, in issue book use case, BarcodeReader object sends the following messages to the IssueBookInterface object:

1. Accept book bar code ID
2. Accept member bar code ID

So the above-mentioned messages become the operations of IssueBookInterface object as shown in Figure 5.20.



Figure 5.20 Identifying operations in IssueBookInterface class.

Similarly, IssueBookInterface sends a message ‘issue book’ to IssueBookController and therefore issueBook becomes an operation of the IssueBookController as shown in Figure 5.21.



Figure 5.21 Identifying operations in IssueBookController class.

The complete set of steps is given below and the portion of the class diagram is shown in Figure 5.22.

1. The bar code reader is associated with the interface screen of issue book. The interface screen accepts the member and book information from the bar code reader.

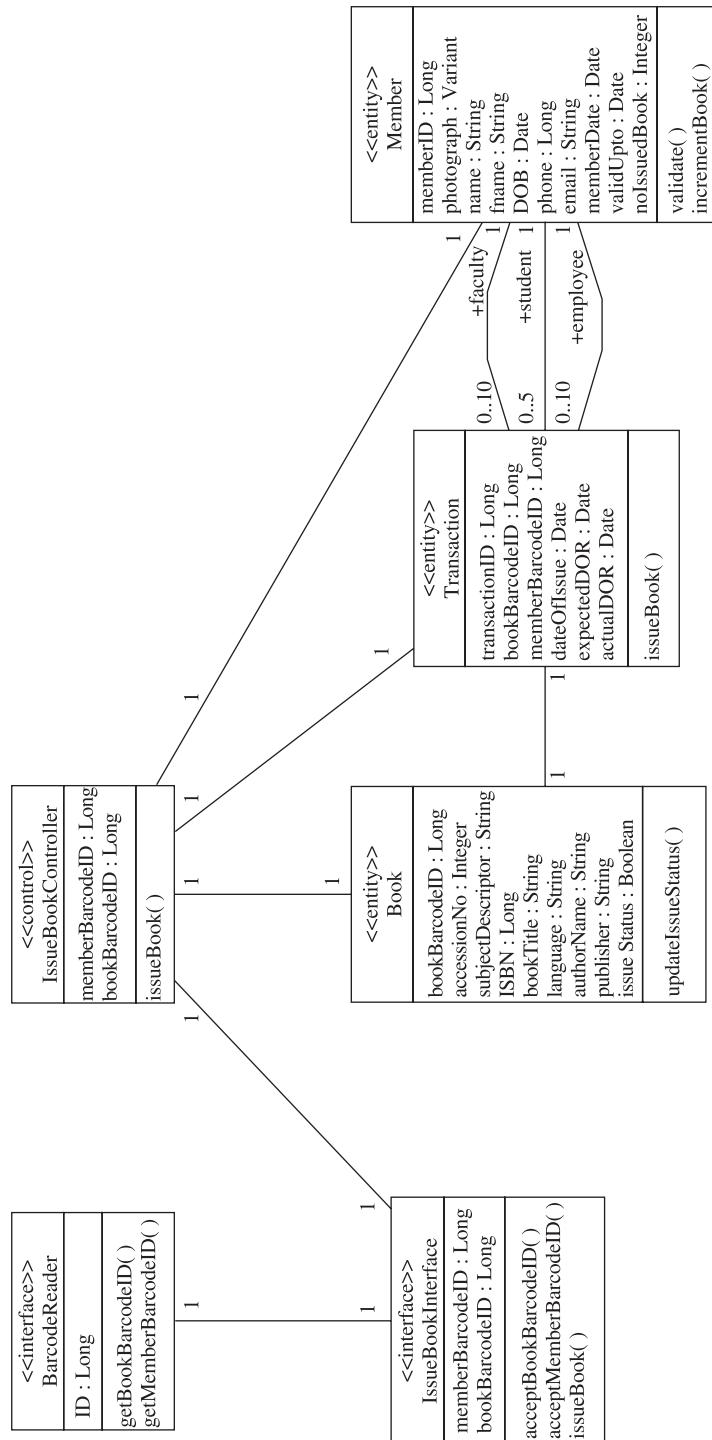


Figure 5.22 Portion of class diagram for ‘issue book’ in LMS.

2. The interface screen sends the issue book message to the controller and the controller transfers the control to the member object in order to validate whether the account of the member is full or not.
3. If the account is not full, then the controller tells the book object to update the status of the book as issued, tells the member object to increment the number of issued books and the issue information is saved in the Transaction object.
4. As we can see in Figure 5.22 that there are three associations between Member and Transaction classes. One of these associations specifies that a member who is a student can get zero to five books issued. The other two associations specify that the member (either a faculty or an employee) can get zero to 10 books issued.

The identified operations will represent the use case under consideration. When other use cases are considered, we may have to add additional operations as per the requirements of that use case. For example, in the Book class when the maintain book details use case is considered, four additional operations such as addBook, deleteBook, updateBook and viewBook may be included.

In UML, we design a class diagram which depicts the set of classes (along with their respective attributes and operations) and their relationships. The class diagram helps us to visualize and model the static aspects of the system. The class diagram models the interaction amongst classes in the system. In Figure 5.22, the portion of the class diagram for ‘issue book’ use case is shown. The class diagram consists of only those operations specific to the ‘issue book’ use case.

EXAMPLE 5.5 Consider Example 5.3 and identify the attributes and operations amongst classes.

Solution Figure 5.23 shows the operations and attributes in the three identified classes.

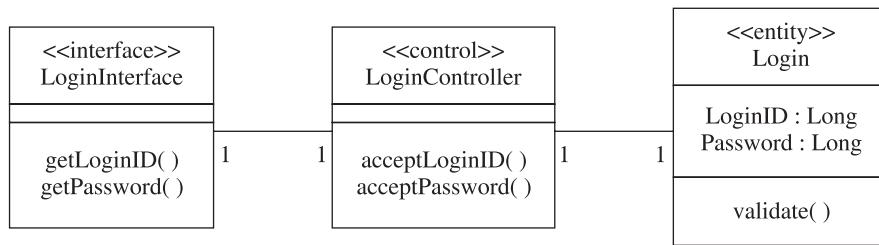


Figure 5.23 Portion of class diagram for ‘login’ in LMS.

EXAMPLE 5.6 Consider Example 5.4 and identify the attributes and operations amongst classes.

Solution The portion of class model for the ‘issue book’ use case is shown in Figure 5.24. In this figure, the classes shown in Figure 5.22 are revised with additional operations. The Transaction class and the Member class are further refined when the return book use case is considered. The Transaction class consists of expectedDOR and actualDOR as attributes. A new class FineStatus is added that keeps record of the fine and the status of the fine, if any, that is paid by a student. The FineCalculator control class temporarily calculates the fine, stores it in the FineStatus class and sends it to the Transaction class.

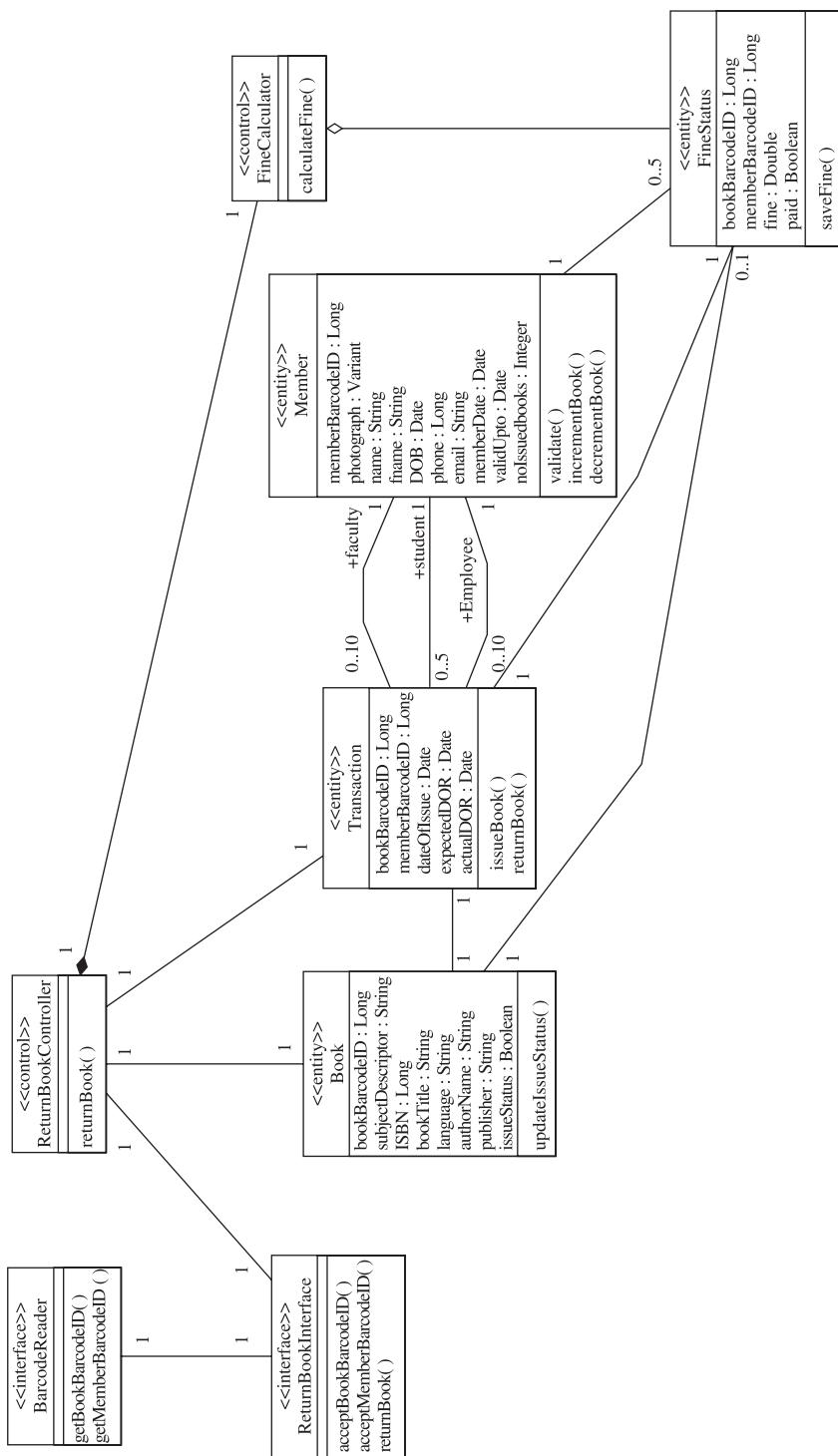


Figure 5.24 Portion of class diagram for 'return book' in the LMS (where already identified operations in the 'issue book' use case are shown).

5.5 Case Study: LMS

In this section, the types of classes for all the use cases in the LMS case study are identified and these classes are summarized in Table 5.7.

Table 5.7 Summary of classes in LMS

Use case	Entity class(es)	Interface class	Control class
Login	Login	LoginInterface	LoginController
Maintain login details	Login	MaintainLoginInterface	MaintainLoginController
Maintain student details	Member Student	StudentInterface	StudentController
Maintain employee details	Member Employee	EmployeeInterface	EmployeeController
Maintain faculty details	Member Faculty	FacultyInterface	FacultyController
Maintain book details	Book	BookInterface	BookController
Issue book	Book Member Transaction	BarcodeReader IssueBookInterface	IssueBookController
Return book	Book Member Transaction FineStatus	BarcodeReader ReturnBookInterface	ReturnBookController FineCalculator
Reserve book	Book Member	ReserveBookInterface	ReserveBookController
Query book	Book	QueryBookInterface	QueryBookController
Generate reports	Book Member	ReportInterface	ReportGenerator

5.6 Moving Towards Object-Oriented Design

The OOA is a medium between designers and customers in order to extract and define requirements. In the OOA phase, user's requirements captured in terms of use cases are converted into entity, interface and control classes along with their relationships. Unlike the traditional design phase, which decomposes the system into modules, in object orientation the classes are created itself in the OOA phase. The OOA is a continuous refining process that makes improvements in analysis classes at each step. An analyst must ensure that:

1. A class satisfies a requirement extracted from the use case description.
2. A class is as far as possible independent of other classes.
3. Relationships between classes are identified carefully.

4. The attributes and operations of a class are identified correctly.
5. The multiplicity and roles of classes are identified carefully and are meaningful.
6. The class diagram is consistent with the given specification and readable.

It is easy to convert the OOA to an object-oriented design (OOD) as compared to the conversion of specification to design in the case of structured analysis. In object-oriented design, the classes (along with attributes and operations) and their relationships, identified in the OOA are refined taking into account the implementation environment. The operations identified in the OOA phase are further refined and added keeping in mind the messages sent between objects in interaction diagrams. The parameters passed to the operations are also identified in the OOD phase. Further, the detailed version of a class diagram is produced in the OOD phase.

Review Questions

1. Compare and contrast object-oriented analysis with structured analysis for the purpose of software development.
2. Explain in detail the various types of relationships between classes with the help of suitable examples.
3. (a) Describe the various approaches to identify classes.
(b) What are interface objects? How will you identify the interface objects?
4. Draw the class diagram and identify at least 10 relationships between the following:
 - (a) Program
 - (b) Class
 - (c) Variable
 - (d) Function
 - (e) Constant
 - (f) Argument
 - (g) Expression
 - (h) Keyword
 - (i) Statement
5. “The goal of analysis model is to develop a model of what the system will do.” Explain this statement with the help of the steps that an analyst will follow throughout the OOA.
6. Consider the employee schema (emp-ID, emp-name, phone, email, street and city). Give the class representation along with the attribute types.
7. Consider the Hospital Management System with the following requirements:
 - The system should handle the in-patient and out-patient information through the receptionist.
 - Doctors can view the patient history and give their prescriptions.
 - There should be an additional system to provide the desired information.Identify entity, interface and control classes along with their relationships.
8. Describe the basic activities of the OOA and explain how the use case modelling is useful in the OOA.

9. Differentiate between aggregation and composition relationships.
10. Suppose a class acts as an actor in the use case model. How will you represent it in the class diagram?
11. How is a class diagram different from an ER diagram?
12. Draw a class diagram for the case study given below.

A directory may contain many other directories and may optionally be contained in other directory. Each directory is used by only one user who is the owner and many users are authorized to use that directory. The identified classes must have their respective attributes and operations. Make necessary assumptions, if any.
13. Differentiate between the following:
 - (a) Aggregation and generalization
 - (b) Aggregation and composition
 - (c) Association and dependency
14. Explain multiplicity and its types with examples.
15. Explain the constituents of a class in UML with the help of an example.
16. An instrument is to be installed to control x elevators in a complex with y floors. The problem domain concerns with the logic required to move elevators between floors according to the following constraints:
 - (i) Each elevator has one button for each floor. Thus, there are y buttons in an elevator. These buttons illuminate and based on this illumination the elevator stops on a particular floor. The illumination is stopped when the elevator visits a respective floor.
 - (ii) There are two buttons in each floor except the first floor and the top floor. One button is used to request an up-elevator and the second button is used to request a down-elevator. When these buttons are pressed, they illuminate and when the elevator visits the floor, the illumination is cancelled.
 - (iii) If there is no request for the elevator, it remains closed and its doors remain shut.

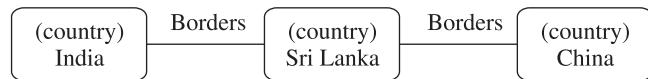
Given this problem statement, identify classes and draw the class diagram to show the relationships amongst the classes. You can make suitable assumptions regarding the details of various features of the elevator but you must clearly write down the assumptions you make.
17. Consider the following case study of Medicine Shop Automation (MSA) software to be used by the owner of a retail shop of medicine.
 - The medicine shop consists of various types of medicines purchased from different vendors. These medicines are kept in numbered racks in the shop. The software should be able to maintain the vendor name and address.
 - The inventory is maintained by the owner of the shop. The shop owner faces a problem of placing order as soon as the number of medicines starts reducing in the inventory below a threshold limit. The medicine shop owner wants to maintain inventory of medicines for about one week for each of the medicine. The limiting

value must be calculated by the software by analysing the average sales per one week for each medicine.

- The software must be able to generate the items ordered at the end of each day. The items must consist of the number of items to be ordered along with the vendor name.
- The software must be able to maintain the description about each item with item no., name, vendor no., quantity, expiry date, etc.
- The software should be able to maintain the items sold to each customer, and should print a cash receipt. At the end of the day, the software should be able to generate a list of expired items.

For implementing the MSA software, identify the classes and their interrelationships and represent them using the class diagram.

- 18.** Prepare a class diagram from the following instance diagram:



- 19.** Explain what you understand by the following types of relations among classes with examples:
- Association
 - Aggregation
 - Generalization
- 20.** The identification of classes along with their attributes and operations is the hardest part of the object-oriented analysis and design. Discuss with the help of an example.
- 21.** Consider the following Bookshop Automation System (BAS) software:
- The BAS should enable the shop clerk to enter the details of the various books the shop deals with and change the inventory level of the various books when new stocks arrive.
 - The BAS should help the customers query whether a book is in stock. The users can query the availability of a book either by using the books title or by using a partial name of the author. If a book is in stock, the exact number of copies available and the rack number in which the book is located should be displayed. If a book is not in the stock, the query for the books is used to increment a request field for the book. The manager can periodically view the request field of the books to roughly estimate the current demand for different out-of-stock books.
 - The BAS should maintain the price of various books. As soon as a customer selects a book to purchase, the sales clerk would enter the ISBN number of the book. The BAS should decrement the stock to reflect the book sale and generate the sales receipt for the book.
 - Upon request, the BAS should generate sales statistics (viz. book name, publisher, ISBN number, number of copies sold and the sales revenue) for any period. The

BAS should enable the manager to view publisher-wise sales (e.g. sale of all books of any given publishing house) over a period.

For implementing the BAS software, identify the classes and their interrelationships and represent them using the class diagram.

22. (a) List and explain the different types of relationships used in the OOA phase along with their notations.
(b) List and explain the various uses of a class diagram.
23. What is an entity class? Explain how entity classes can be used in designing database structure.
24. Explain the various types of classes along with their notations.
25. List and explain the ways for identification of attributes and operations in a class.
26. The car rental agency has multiple offices/branches. The customer visits the agency for enquiry and takes a test ride, then selects the car by signing the terms and conditions form. The customer can also book the car through telephone, email and sms. The agency checks the availability of the car and informs the status to the customer. The driver facility is also available to the customer if required, by paying additional charges. Identify classes (along with attributes and operations) and their relationships, and draw the class diagram for a car rental application.
27. ‘Pragati Apartments’ is a company that gives apartments on lease to the customers. This company owns multi-storeyed buildings in various locations in Delhi. Each building consists of a number of apartments that are classified into the following types:
 - Type 1: 4 bed roomed apartment with 3 attached bathrooms and 1 guest room
 - Type 2: 3 bed roomed apartment with 2 attached bathrooms
 - Type 3: 2 bed roomed apartment with 1 attached bathroom
 - Type 4: 1 bed roomed apartment

Each apartment consists of a dining area, a kitchen and a parking space for one vehicle. Each building and each apartment have unique identification numbers. At a given time, an apartment may be available, unavailable or reserved.

The customer signs a lease agreement consisting of terms and conditions and expiry date of the agreement.

The receptionist must be able to provide information about the availability of an apartment. She must also provide information about location, deposit amount, number of rooms and other relevant information about the desired apartment to the customer. If the apartment is not available, the customer must be able to reserve the apartment and he will be in the waiting list. The applications are served on a first come, first served basis.

- (a) Identify classes for the above system.
(b) Identify the pair of classes where a composition relationship exists. Draw the relationship using UML notations and indicate the multiplicity.
28. What are the activities involved in the OOA phase? How is the OOA different from the structured analysis? Provide guidelines for an analyst during the OOA phase.

29. Create three classes linked by associations to represent a student taking courses in a school. Specify appropriate multiplicity as well as association names for the association. If there is more than one alternative, explain the advantages and disadvantages of each.
30. For a university, course and its professor association, indicate whether it should be an ordinary association, an aggregation or a composition.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. Structured approach involves:

(a) Top-down approach	(b) Bottom-up approach
(c) Sandwitch approach	(d) None of the above
2. The class modelling technique:

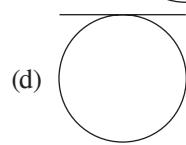
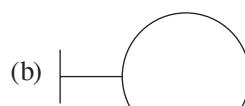
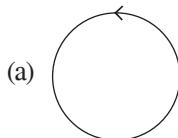
(a) Separates data and functions	(b) Combines data and functions
(c) Gives more importance to data	(d) Gives more importance to functions
3. Classes can be categorized into:

(a) Entity, boundary, monitor	(b) Entity, control, abstract
(c) Virtual, abstract, template	(d) Entity, control, interface
4. The purpose of entity classes is to:

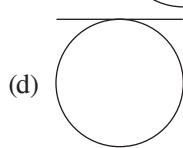
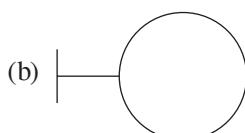
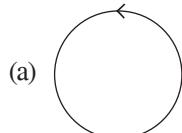
(a) Store temporary variables	(b) Store persistent data
(c) Store control information	(d) None of the above
5. Interface classes are used to:

(a) Handle control information in the system
(b) Handle interactions in the system
(c) Handle information that will persist longer in the system
(d) All of the above
6. Control classes are responsible for:

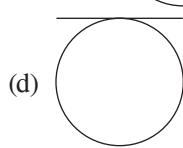
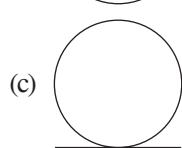
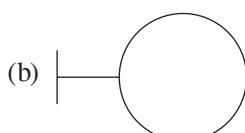
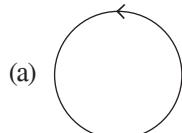
(a) Storing persistent data
(b) Handling interactions in the system
(c) Coordinating and managing entity and interface classes
(d) None of the above
7. The notation used to represent an entity class is:



8. The notation used to represent an interface class is:



9. The notation used to represent a control class is:



10. Which of the following is **not** true?

- (a) Control classes are independent of their surroundings
- (b) Control classes are used to control the flow of events
- (c) Control classes are not affected by the changes in the entity classes
- (d) None of the above

11. Which type of relationship is modelled by the aggregation relationship?

- | | |
|-------------|------------------|
| (a) Is-a | (b) Has-a |
| (c) Type-of | (d) Parent-child |

12. Which type of relationship is modelled by the generalization relationship?

- | | |
|-------------|-------------|
| (a) Is-a | (b) Has-a |
| (c) Type-of | (d) Part-of |

13. In which of the following relationships, multiplicity can be specified?

- (i) Aggregation
 - (ii) Generalization
 - (iii) Dependency
 - (iv) Association
- | | |
|------------------|------------------------|
| (a) (i) and (ii) | (b) (i) and (iii) |
| (c) (i) and (iv) | (d) (i), (ii) and (iv) |

14. Dependency relationship is depicted by:

- | | |
|------------------------------------|--|
| (a) A directed dashed line | (b) A solid line |
| (c) A line with diamond at the end | (d) A solid line with an open arrow head |

- 15.** Aggregation relationship is depicted by:
- (a) A directed dashed line
 - (b) A solid line
 - (c) A line with diamond at the end
 - (d) A solid line with an open arrow head
- 16.** If a class is a whole and other classes are its parts, then which type of relationship should be modelled?
- (a) Association
 - (b) Dependency
 - (c) Aggregation
 - (d) Generalization
- 17.** Which type of class may contain the maximum number of attributes?
- (a) Control
 - (b) Abstract
 - (c) Interface
 - (d) Entity
- 18.** The extent to which different classes are dependent upon each other is called:
- (a) Cohesion
 - (b) Coupling
 - (c) Inheritance
 - (d) Modularity
- 19.** The attributes can be identified from:
- (a) Use case description
 - (b) SRS document
 - (c) Database schema
 - (d) All of the above
- 20.** An analyst must ensure:
- (a) A class is independent from others
 - (b) All classes identified can be traced back to the requirements
 - (c) Class diagram is consistent and readable
 - (d) All of the above

Further Reading

Classic books that define object-oriented analysis:

Coad, P. and Yourdon, E., *Object-Oriented Analysis*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.

Martin, J. and Odell, J., *Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

A comparison of object-oriented analysis methods is available in the below research paper:

Champeaux, D. and Faure, P., A comparative study of object-oriented analysis methods. *Journal of Object-Oriented Programming*, 5(1), March/April 1992.

An introduction to UML is available in the following books:

Miles, R. and Hamilton, K., *Learning UML 2.0*. Sebastopol, CA: O'Reilly Media, 2006.

Arlon, J. and Neustadt, I., *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design*. Boston, MA: Addison-Wesley, 2005.

Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*, 2nd ed. Reading, MA: Addison-Wesley, 2004.

Booch, G., Jacobson, I. and Rumbaugh, J., *The Unified Modeling Language (Version 2.2)*. The Object Management Group, 2008.

Pilone, D. and Pitman, N., *UML 2.0 in a Nutshell*, 2nd ed. Sebastopol, CA: O'Reilly Media, 2005.

Roques, P., *UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions*. Hoboken, NJ: John Wiley & Sons, 2004.

Webster lists 240 references including a larger set of literature on object-oriented analysis in his research paper:

Webster, S., An annotated bibliography for object-oriented analysis and design. *Information and Software Technology*, 36(9): 569–582, September 1994.

6

Object-Oriented Design

After the OOA phase, further refinement of classes begins with object-oriented design (OOD) phase. In the OOD phase, the detailed class diagram is constructed and the interaction diagrams are created. The classes along with their attributes and operations, and their relationships, are refined keeping in mind the implementation environment including programming languages, databases and communication protocols. The operations in classes identified in the OOA phase are also added, deleted and updated with respect to messages in interaction diagrams.

In this chapter, the procedure used to create interaction diagrams is described. The notations used in these diagrams are given using the UML. The detailed set of operations of classes is identified from the interaction diagrams. Use cases descriptions are refined. Further, the procedure to generate test cases from use cases is explained.

6.1 What is Done in Object-Oriented Design?

In the OOD phase, we define in detail the interfaces of the object and the operation semantics of the object. Why can't we directly implement the analysis object? What is the need of design phase? There are many reasons to have the OOD phase:

1. In the analysis phase, an ideal model is created with the aim to build a maintainable model. The classes identified are not designed as per the implementation environment. Thus, the analysis classes need to be transformed taking in view the implementation environment.
2. The objects along with their relationships need to be refined. The operations must be refined and updated according to the messages sent between objects.
3. The classes along with their relationships must be verified so that a well-focused and detailed class model can be built.

As shown in Figure 6.1, the analysis classes are refined to design objects. The steps involved in the OOD are given as follows:

1. **Creation of interaction diagrams for each scenario**
 - (a) Creation of **sequence** diagram
 - (i) Identification of actors, objects and messages in a scenario of a use case
 - (ii) Identification of object's lifeline, focus of control and sequence of interaction
 - (iii) Refinement of sequence diagram
 - (b) Creation of **collaboration** diagram
 - (i) Identification of object, links and messages
 - (ii) Identification of structural relationship between objects
 - (iii) Identification of sequence of messages
2. Refinement of classes and relationships identified in the OOA phase
3. Identification of operations from the sequence diagram
4. Construction of a detailed class diagram
5. Development of a detailed design
6. Creation of software design document
7. Generation of test cases from use cases

Each of these steps is explained in detail in the subsequent sections.

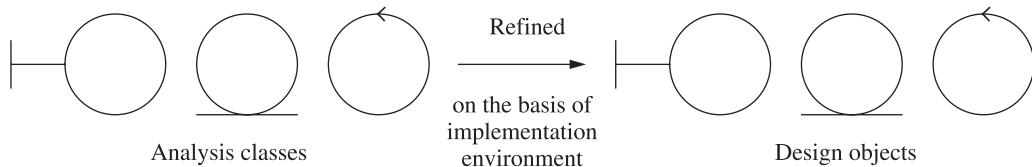


Figure 6.1 Refining analysis to design.

6.2 Interaction Diagrams

An interaction diagram depicts the communication between a set of objects through messages passing among themselves. The interaction diagram models the dynamic aspect of the system. For each use case, an interaction diagram of one particular flow (scenario) of the use case is constructed. The interaction between objects starts by sending messages to each other. Thus, the messages including their parameters are defined. The use cases were identified in the requirement analysis phase. The classes along with their relationships were identified in the OOA phase. Now in the design phase, the interaction amongst objects is depicted for each use case or use case scenario. This may cause addition, deletion or change in the operations identified in the OOA phase. The objects involved in the use case have been already identified in the OOA phase. Some new objects if introduced in the design phase due to consideration of implementation environment will also be included in the interaction diagrams.

In the UML, interaction diagrams are of two types: sequence diagram and collaboration diagram. A sequence diagram shows the interaction amongst objects with respect to time. The objects appear on the x -axis and the time lines are shown in the y -axis. A collaboration diagram depicts the sequence in which objects send or receive messages amongst themselves.

6.3 Sequence Diagrams

In sequence diagrams, the messages send amongst objects are time ordered. The sequence diagrams depict the focus of control. Each sequence diagram represents one of the scenarios of the use case. A separate sequence diagram may be created for an alternate flow in a use case.

6.3.1 Objects, Lifeline and Focus of Control

An object is an instance of a class and is depicted by:

Objectname:classname

The object is shown in a rectangle and is underlined as shown in Figure 6.2. The dashed line appearing on the vertical axis is known as the lifeline of an object. The lifeline of the object depicts the amount of time the object is alive in the use case. The lifeline of the object begins as soon as the actor sends some message to the object. The lifeline of the object ends when it is no longer required for any further interaction in the sequence diagram. The order of the vertical lines is not significant and must be chosen carefully to increase the clarity of the sequence of events. The object names may not be known in the initial phases of software development. Unnamed objects can be created without specifying the object names as shown in Figure 6.3.

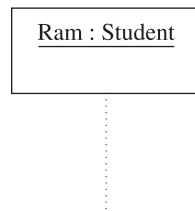


Figure 6.2 Object with timeline in a sequence diagram.

There is a focus of control which is represented by a rectangle. The focus of control shows the time period for which the object is under interaction or performing a specific event. The rectangle starts with the beginning of a set of procedures, or actions and ends when the procedure is completed. In most of the times, an object will be alive for an entire interaction. Figure 6.3 shows an object with focus of control.

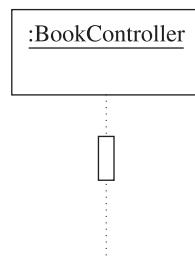


Figure 6.3 Focus of control in a sequence diagram.

6.3.2 Messages

Messages are an important element in a sequence diagram. Messages represent interaction between objects, in which the sending object asks the receiving object to perform some operation.

Messages are represented by arrows from the sending objects to the receiving object as shown in Figure 6.4. In this figure, a message is sent from the object of BookController class to the object of Book class.

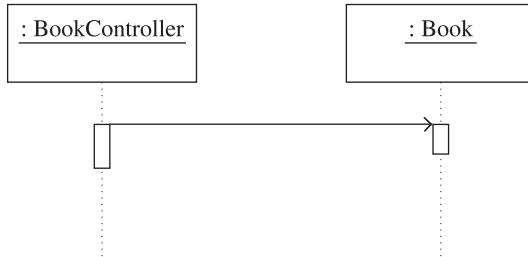


Figure 6.4 Messages in a sequence diagram.

There are various types of messages in the UML which are given as follows:

1. *Simple messages:* A simple message is used to represent interaction between objects that may not be a procedure call. The simple message is represented in Figure 6.5.

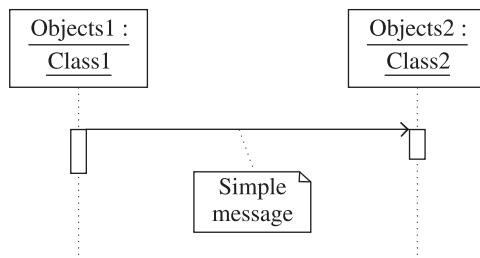


Figure 6.5 Simple message in a sequence diagram.

2. *Synchronous messages:* When a synchronous type of message is sent, the sending object waits to receive a response from the receiving object. The synchronous message is represented in Figure 6.6.

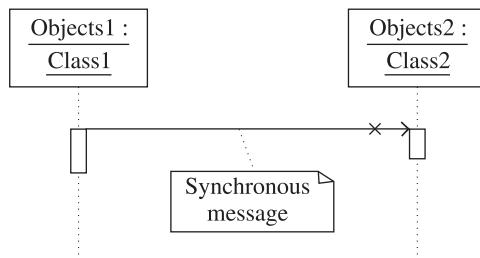


Figure 6.6 Synchronous message in a sequence diagram.

3. *Asynchronous messages:* When an asynchronous type of message is sent, the sending object does not wait to receive a response from the receiving object. The asynchronous message is represented in Figure 6.7.

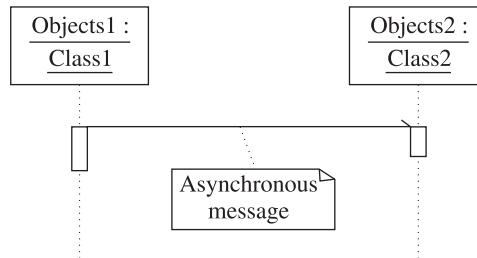


Figure 6.7 Asynchronous message in a sequence diagram.

4. *Procedure call:* The outer sequence resumes after the completion of the inner sequence of the procedure. The procedure call message is represented in Figure 6.8.

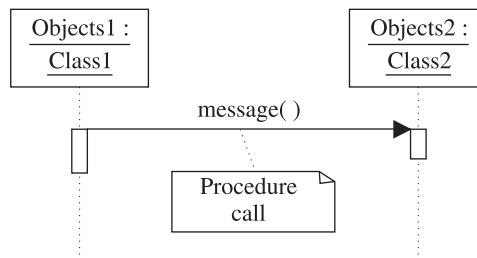


Figure 6.8 Procedure call message in a sequence diagram.

5. *Return message:* If some response value is given by a receiving object in response to some message obtained by the sending object, this is called a *return message*. The procedure call message is represented in Figure 6.9. It is not necessary to always return a value from a message. However, if an operation is returning something relevant, then that value may be shown in the diagram.

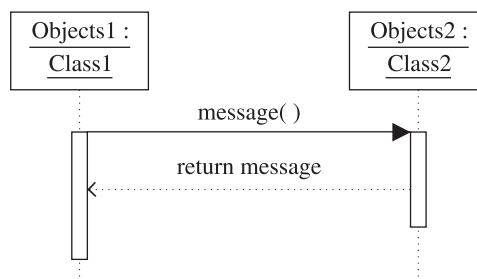


Figure 6.9 Return message in a sequence diagram.

A message can also be sent by an object to itself as shown in Figure 6.10. This type of message is known as *reflexive message*.

A new object can be created anytime during the sequence diagram. The new object can be created when required and is placed below the calling object. The calling object then sends a message of creation to the new object. An object may be destroyed during the interaction between objects. The destruction of an object is represented by a big X, which signifies the end

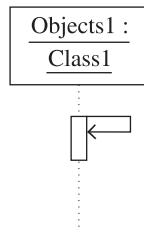


Figure 6.10 Reflexive message in a sequence diagram.

of the life of an object. The messages give rise to operations. The message by an object becomes operation of the receiving object. In Figure 6.11, Object1 creates Object2 by sending create message and Object2 is destroyed after interaction is completed.

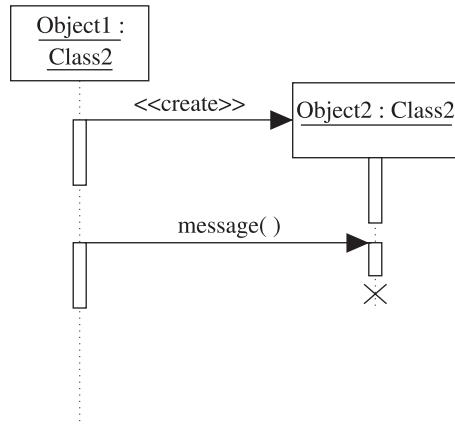


Figure 6.11 Creating and destroying an object.

6.3.3 Creating Sequence Diagrams

In order to construct a sequence diagram, we must first identify the objects participating in the use case. The objects in the OOD can be identified quickly by the classes identified in the OOA phase. As the work for creating use case diagrams, writing use case description and identifying classes with their relationships is done in earlier phases, a good architecture and design of the software can be created at an early stage. Since the work of identifying classes for the issue book use case was done earlier in the OOA phase, the objects can easily be identified. The objects participating in the issue book use case are created for the classes such as BarcodeReader, IssueBookInterface, IssueBookController, Member, Transaction and Book. If required, new objects may be introduced keeping the implementation environment in mind.

After identifying the participating objects, the external entity that will initiate the sequence diagram is identified. The sequence diagram begins when some message is sent by an external entity. The first lifeline in a sequence diagram is of the external entity (actor in the use case) that starts the sequence diagram. In the case of the issue book use case, an external entity may be Administrator, Librarian, or LibraryStaff. The skeleton of the basic flow scenario of the issue

book use case is shown in Figure 6.12. To the left end in the sequence diagram, we may specify the sequence of events using textual messages. This will increase the clarity of the diagrams and will help the programmers in implementation.

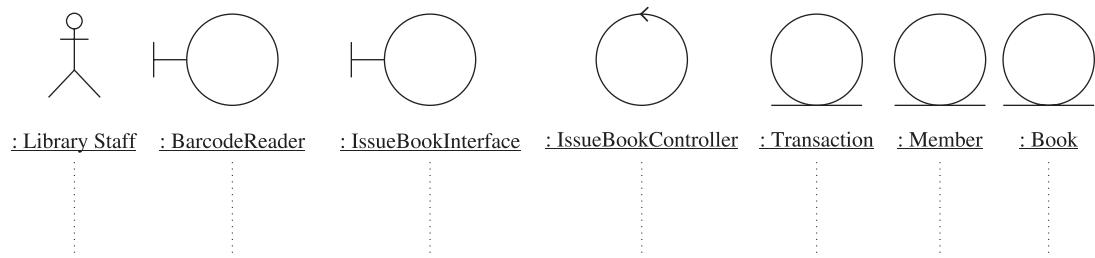


Figure 6.12 Skeleton of sequence diagram for issue book use case.

The sequence diagram is shown in Figure 6.13. On the left-hand side of the diagram, textual messages are shown to increase the readability and understandability of the diagram. The actor Library staff initiates the sequence diagram for basic flow of the ‘issue book’ use case by sending the first message ‘read barcode of the book’. As shown in Figure 6.13, six classes participate in the interaction following the sequence of events given above. The sequence diagram for the basic flow scenario of the ‘issue book’ use case will follow the following steps:

1. The bar code of the book is read through the bar code reader.
2. The issue book interface object accepts the book bar code.
3. The member bar code is read through the bar code reader.
4. The issue book interface accepts the member bar code.
5. The issue book message is sent to the controller object.
6. After validating the valid membership and the number of books in the member account, the controller sends the message to the transaction object so that the book is issued.
7. The information is finally saved to member and book objects.

There are two types of control structures—centralized control structure (fork shaped) and decentralized control structure (stair-case shaped). Figure 6.14 shows the centralized control structure and Figure 6.15 shows the decentralized control structure. In the centralized control structure, the controlling objects are responsible for managing the flow of sending and receiving messages. These controlling objects decide when which object will be activated and the order in which messages will be sent. In the decentralized control structure, the participating objects directly communicate with other objects without any controlling object.

The sequence diagram with decentralized control structure and operations for the basic flow scenario issue book use case is shown in Figure 6.16. The revised sequence diagram with centralized control structure is shown in Figure 6.17. In this diagram, we have increased the functionality of IssueBookController object. In the sequence diagram shown in Figure 6.17, everything is managed and handled by IssueBookController object. This object controls the functionality of the sequence diagram. This provides a centralized control structure for issuing a book and saving the changes to the book and member object. The centralized control structure is better, because if there is a change in sequence, only IssueBookController object will be affected.

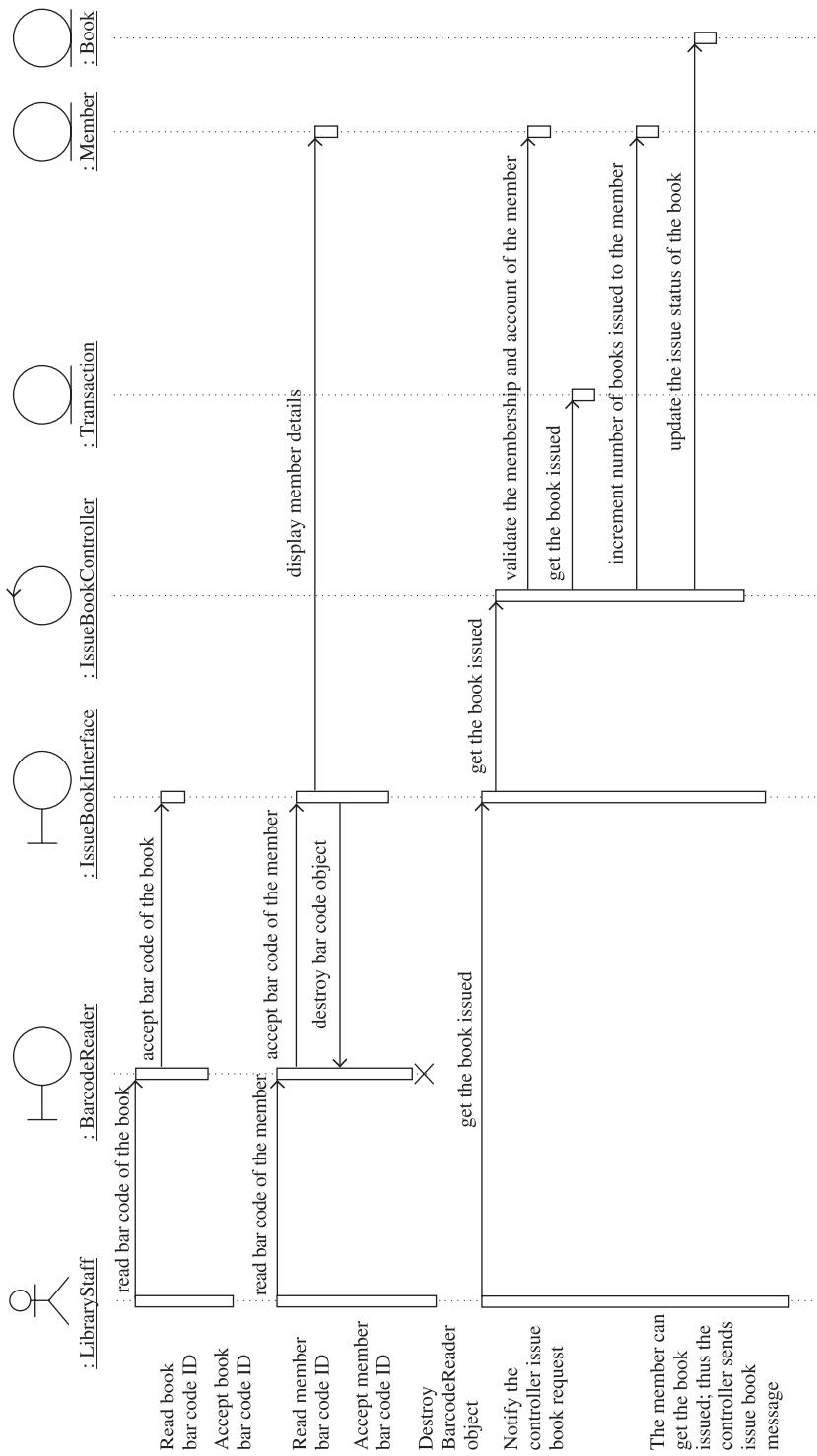
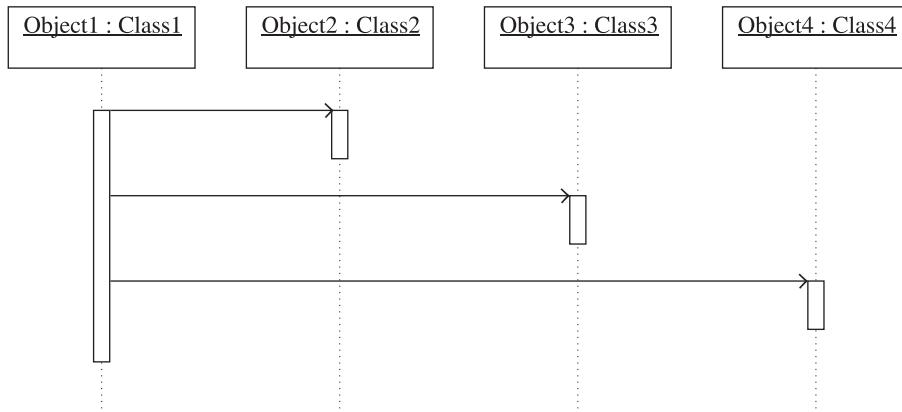
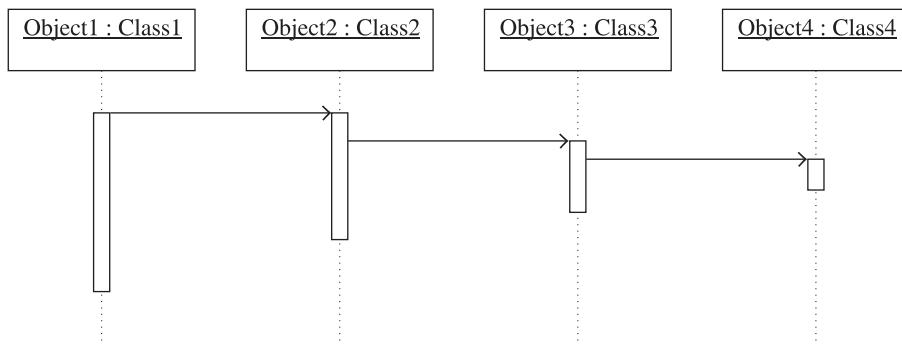


Figure 6.13 Sequence diagram for basic flow scenario of issue book.

**Figure 6.14** Centralized control structure.**Figure 6.15** Decentralized control structure.

Thus, a reusable, stable and updatable structure can be created. In addition, the centralized control structure will always allow to add new operations in the sequence diagrams. Another advantage of the centralized control structure is that the parts of the functionality can be easily reused. A decentralized control structure is preferable when the messages are strongly coupled with each other (Jacobson et al., 1997).

In Figures 6.16 and 6.17, the first lifeline in the sequence diagram is the lifeline of the actor LibraryStaff. The LibraryStaff initiates the diagram by invoking operation ‘read bar code’ of the object of the BarcodeReader class. The BarcodeReader receives destroy message as soon as the book and member ID are read from it. The big X signifies that the object is destroyed and is only alive till the lifeline exists as shown in Figure 6.17.

We may create a separate sequence diagram for each scenario in the use case. The use case description is divided into two parts—one containing the basic flow and the other containing alternative flows. A basic flow in the use case description depicts the most important functionality that is going to occur most of the times in the use case. In Figure 6.17, we have modelled the basic flow of the ‘issue book’ use case. An alternative flow usually depicts the error conditions that may occur a few times during the execution of the use case. Thus, all the possible alternative

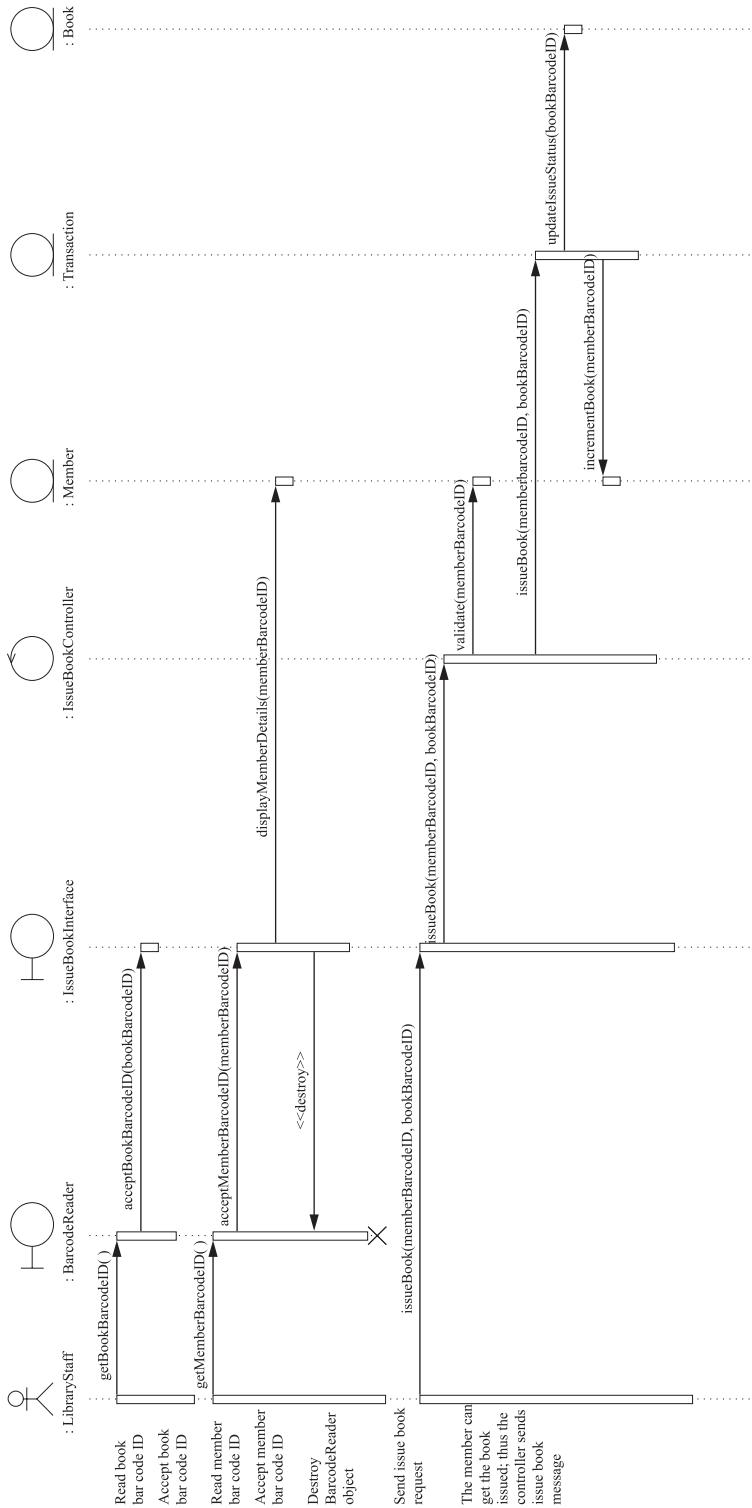


Figure 6.16 Sequence diagram for basic flow scenario of issue book.

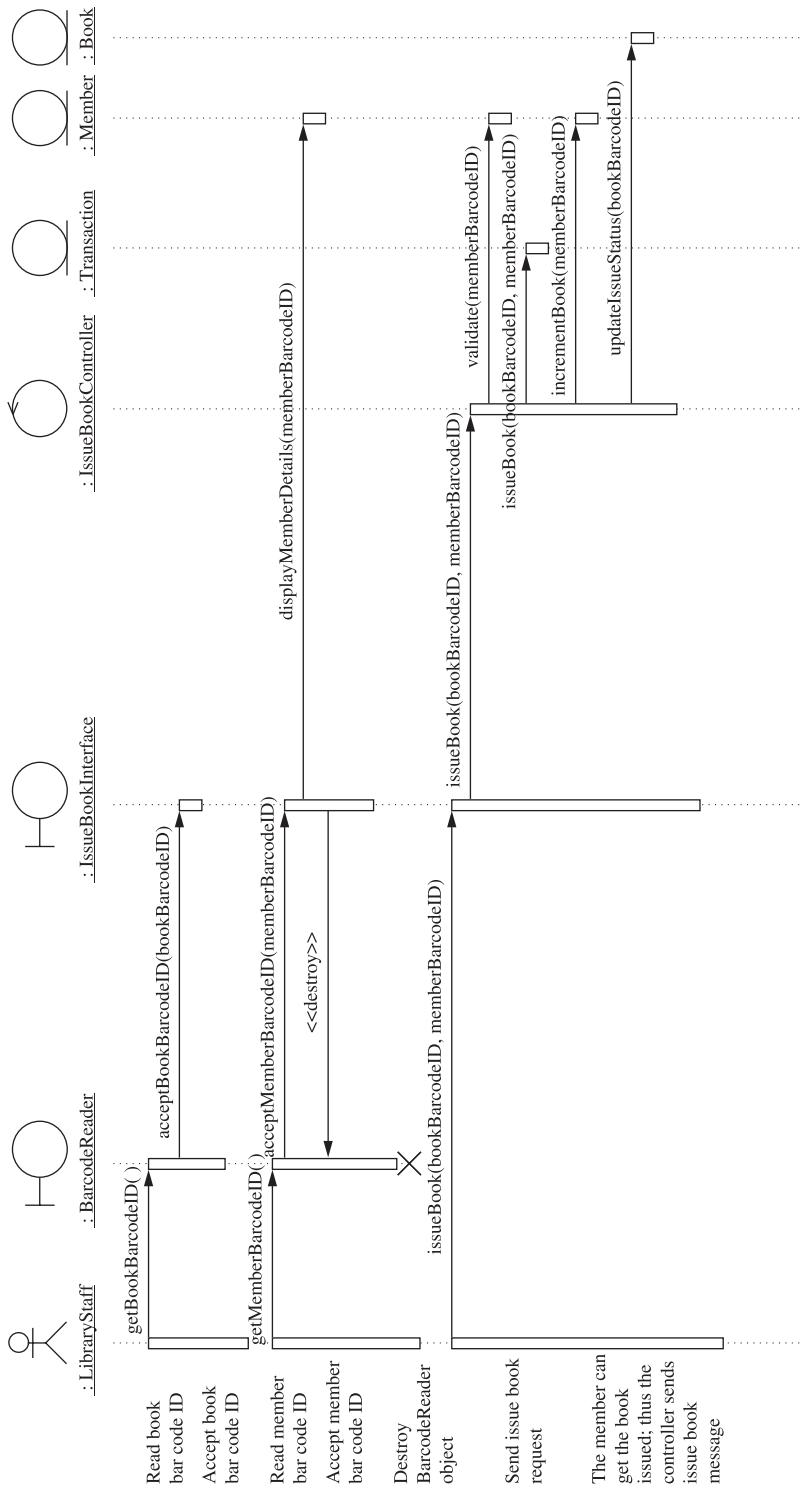


Figure 6.17 Revised sequence diagram with centralized control structure for basic flow scenario of issue book.

cases should be modelled as this will help in constructing a complete and robust design. The alternative cases in the issue book use case are shown below:

Alternative Flow 1: Unauthorized member

If the system doesn't validate the member's (student/faculty/employee) membership number (due to membership expiry), then an error message is flagged and the use case returns to the beginning of the basic flow.

Alternative Flow 2: Account is full

If the student/faculty/employee has requested a book in Account and it is full, i.e. he/she has already maximum number of books issued on his/her name, then the request for issue is denied and the use case ends.

Alternative Flow 3: Invalid entry

If in the issue book flow, the actor enters an invalid barcodeID and/or memberID or leaves the barcodeID and/or memberID empty, the system displays an error message. The actor returns to the beginning of the basic flow.

Alternative Flow 4: User exits

This allows the user to exit at any time during the use case. The use case ends.

Figures 6.18 and 6.19 show the sequence diagrams for alternative cases 1 and 2 in the 'issue book' use case. The membership of the member is not valid, thus a return message 'invalid membership' is returned by the object of Member class as shown in Figure 6.18. Similarly, the error message 'Account full' is returned to the actor LibraryStaff as the account of the member has already maximum number of books issued. The sequence diagram for alternative flows 3 and 4 can be created in the similar manner.

The sequence diagram must be kept as simple and understandable as possible. It should be easy to visualize the objects and interaction between these objects for a given scenario.

6.3.4 Creating Sequence Diagram of Use Cases with Extensions

A use case may contain exceptional cases which are modelled through the extend relationship between two use cases. In the interaction diagram, the extends relationship may be modelled by the procedure call, i.e. inserting the functionality of the extended use case. For example, the fine calculation use case extends the return book use case. Thus, the use case description of the fine calculation use case extends the use case description of return book. During execution of the sequence diagram, the fine calculation procedure is inserted. The position of the procedure call can be determined by the use case description of return book. Figure 6.20 shows the sequence diagram of the basic flow of return book. The fine is only checked if the member is a student. This condition is checked by the guard condition represented by square brackets. The ReturnBookController object passes the message to the Transaction object to check the fine in case of student. The Transaction object sends the fine status as false to the ReturnBookController object. Thus, in the basic flow represented in Figure 6.20, the fine is not calculated.

Figure 6.21 shows the use case fine calculation inserted into the use case return book use case. The ReturnBookController object calls the FineCalculator object to calculate the fine. The procedure call is depicted in the sequence diagram. The FineCalculator object calculates and saves the fine. After this, the FineCalculator object is destroyed by the ReturnBookController

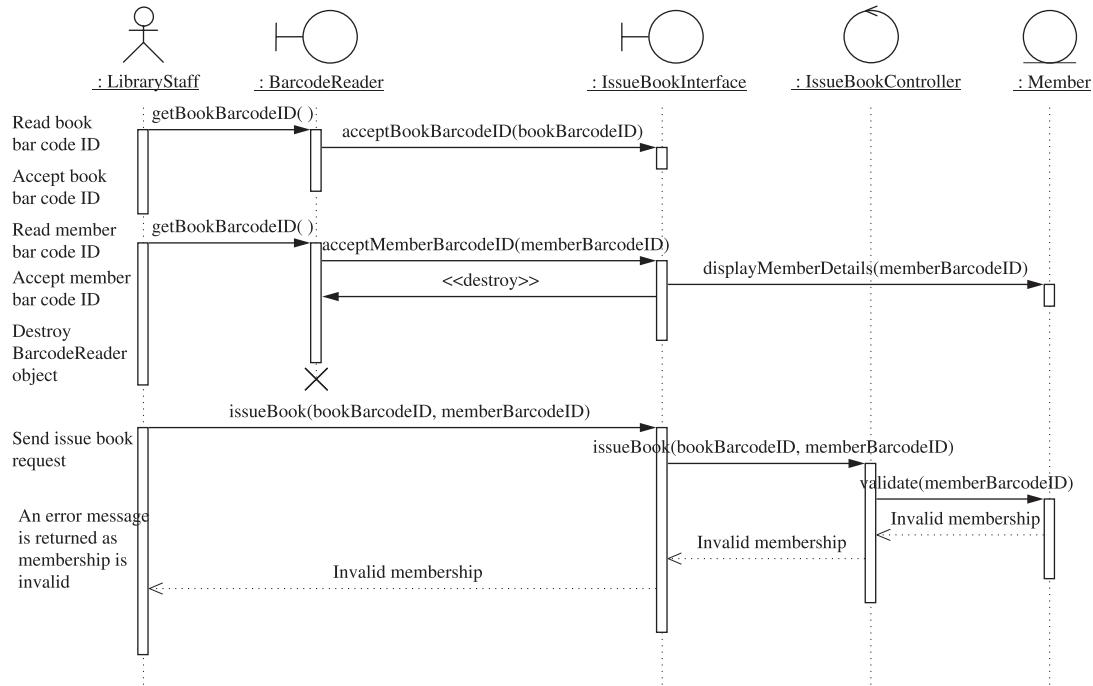


Figure 6.18 Sequence diagram for alternative flow 1 of issue book.

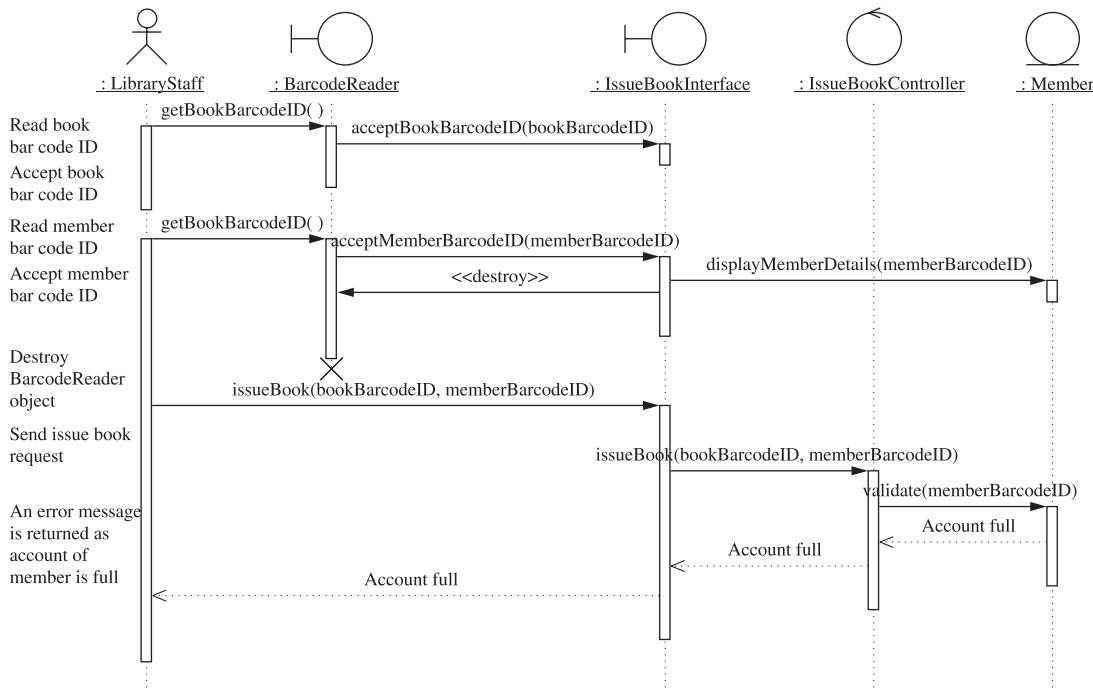


Figure 6.19 Sequence diagram for alternative flow 2 of issue book.

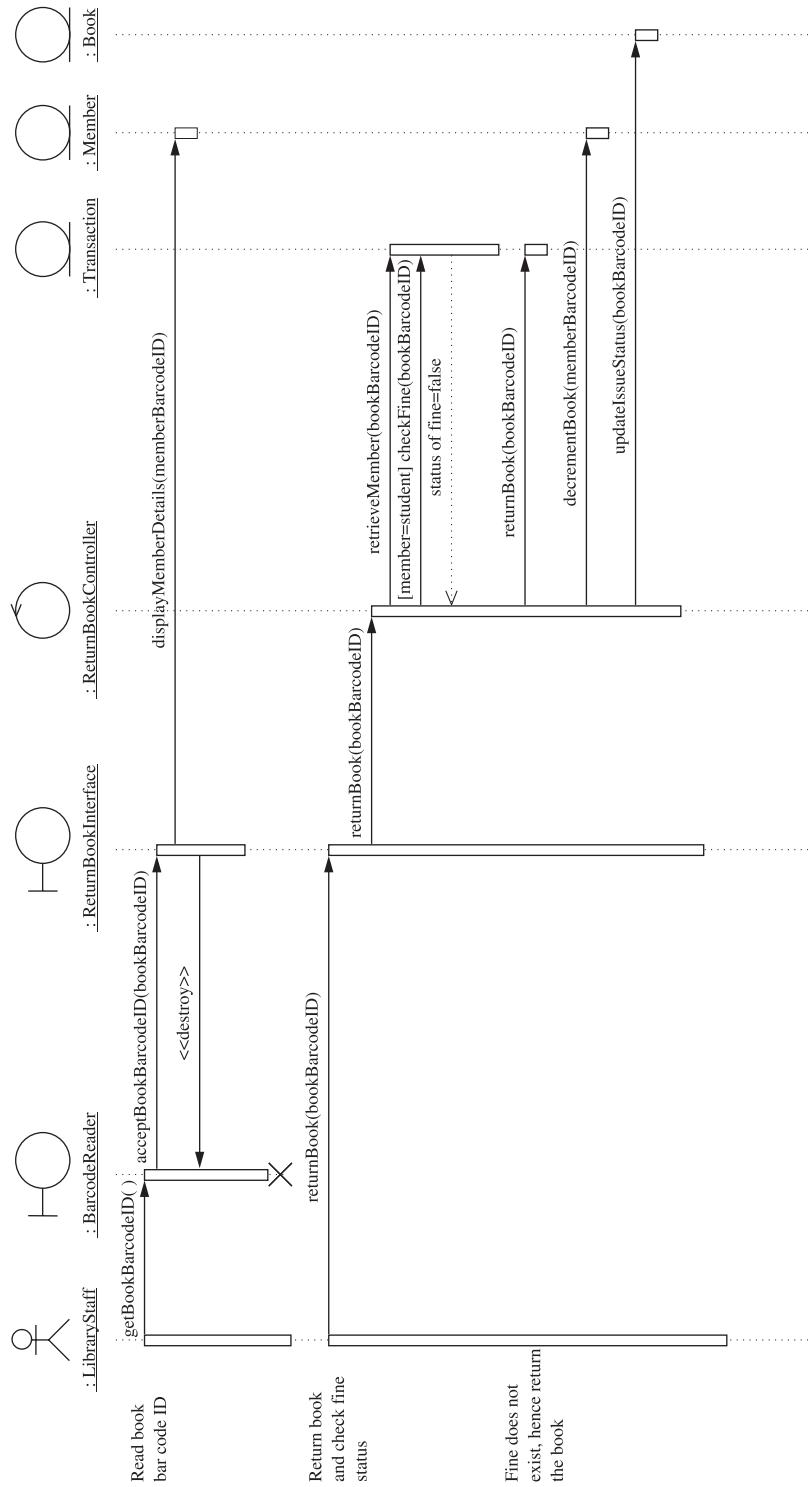


Figure 6.20 Sequence diagram for basic flow of return book.

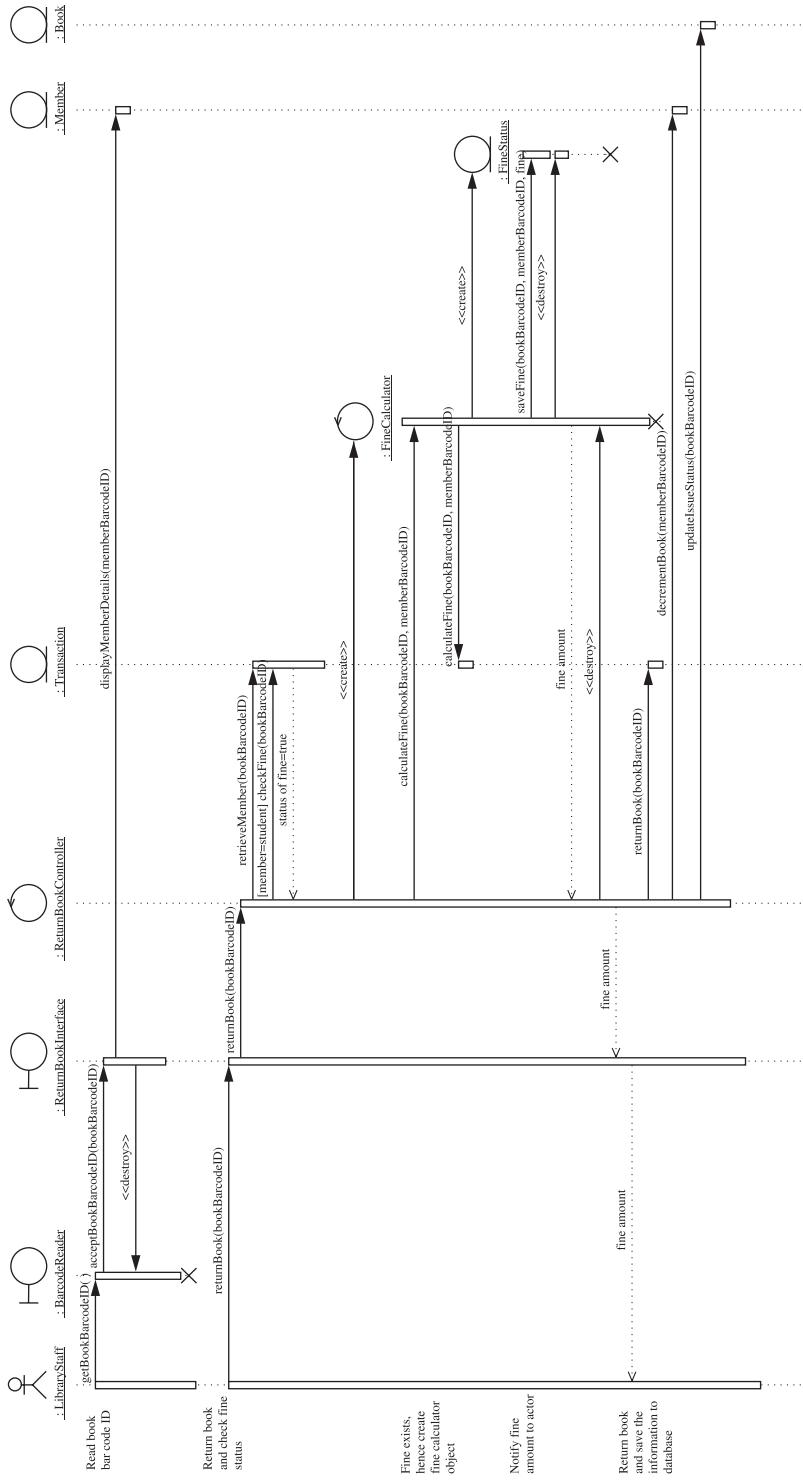


Figure 6.21 Sequence diagram with fine calculation use case inserted into return book.

object. The ReturnBookController object sends the fine to the interface and the interface finally sends it to the external entity (LibraryStaff in this case). Then the book is returned by the ReturnBookController object as done in the basic flow.

EXAMPLE 6.1 Consider the login use case of the LMS. Draw the sequence diagram for it.

Solution The portion of the use case description of the basic flow of ‘login’ use case is given below:

Basic Flow 1: Login

- (i) The system requests that the actor enter his/her login ID and password information.
- (ii) The actor enters his/her login ID and password.
- (iii) The actor enters into the system.

The sequence diagram for the basic flow of ‘login’ use case is shown in Figure 6.22. The actor LibraryStaff enters login ID and password and hence the message is shown as a simple message in Figure 6.22.

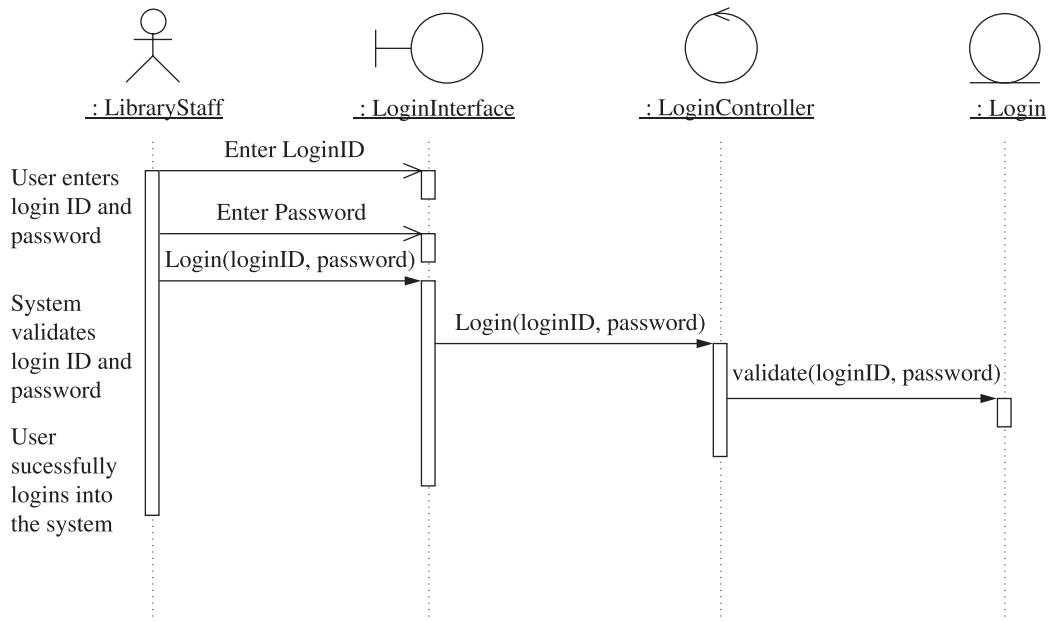


Figure 6.22 Sequence diagram for basic flow of login.

There are three alternative flows in the ‘login’ use case. The portion of alternative flow 1 is shown below and the sequence diagram for alternative flow 1 is shown in Figure 6.23. Similarly, the sequence diagrams for the rest of the alternative flows can be created.

Alternative Flow 1: Invalid login ID/password

If in the login flow, the actor enters an invalid login ID and/or password or leaves the login ID and/or password empty, the system displays an error message. The actor returns to the beginning of the basic flow.

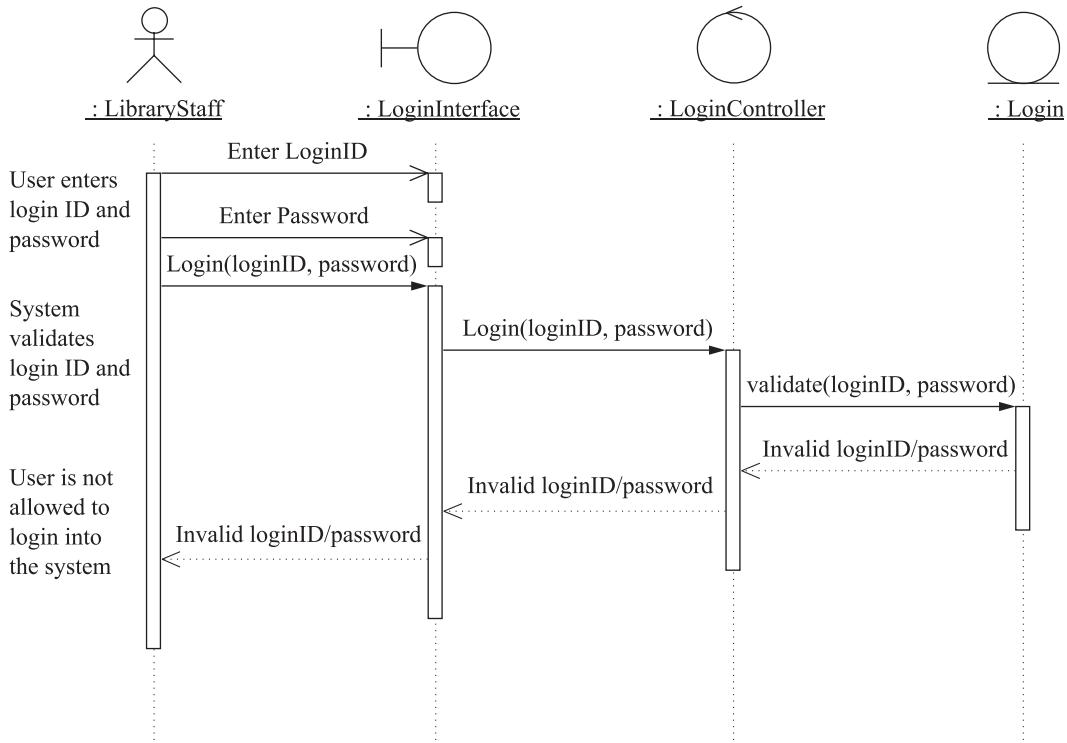


Figure 6.23 Sequence diagram for alternative flow 1 of 'login' use case.

EXAMPLE 6.2 Consider the 'maintain book details' use case of the LMS. Draw the sequence diagram for this use case.

Solution There are four basic flows in the 'maintain book details' use case: add a book, delete a book, update a book and view a book. Hence, four sequence diagrams corresponding to each basic flow will be created. The portion of the basic flow (basic flow 1: add a book) is given below and its sequence diagram is shown in Figure 6.24.

Basic Flow 1: Add a book

The system requests that the administrator/DEO enter the book information. This includes:

- Accession number (book bar code ID)
- Subject descriptor
- ISBN
- Title
- Language
- Author
- Publisher

Once the administrator/DEO provides the requested information, the book is added to the system.

The operations addDetail() and saveDetails() consist of all attributes of the book specified in the basic flow given above as arguments. However, due to scarcity of space, these arguments have not been shown in the sequence diagram.

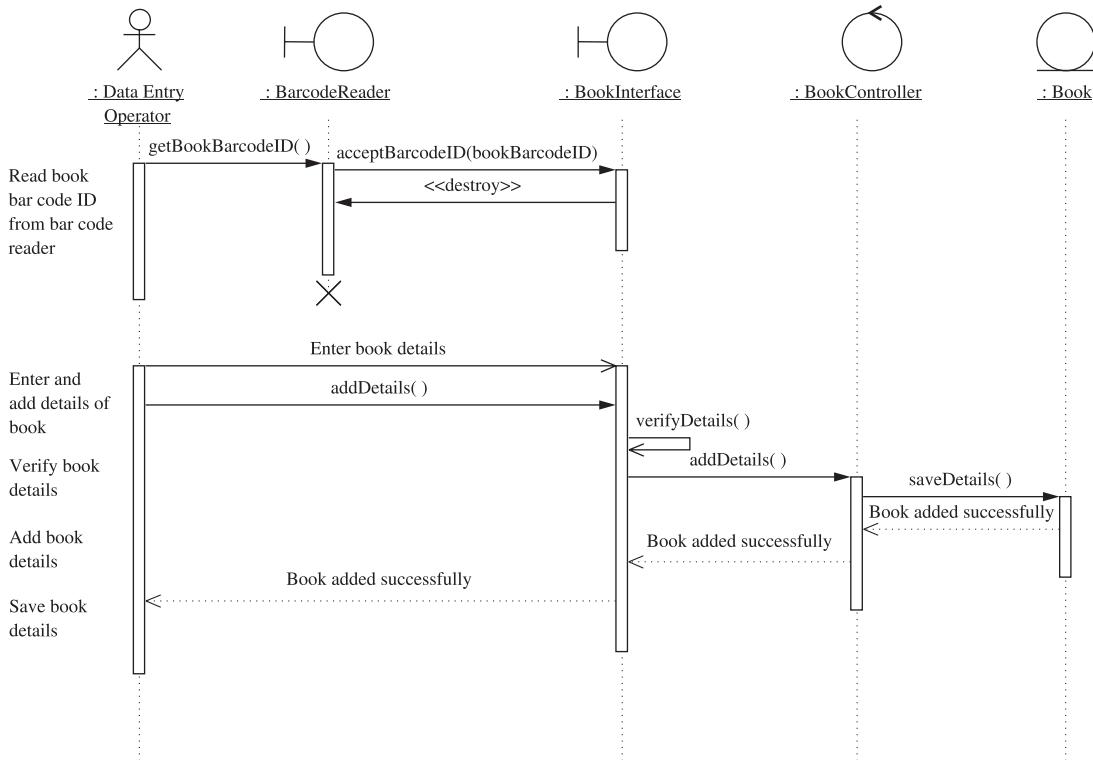


Figure 6.24 Sequence diagram for add a book.

The portion of the basic flow (basic flow 2: update a book) is given below and its sequence diagram is shown in Figure 6.25.

Basic Flow 2: Update a book

1. The system requests that the administrator/DEO enter the accession number.
2. The administrator/DEO enters the accession number.
3. The system retrieves and displays the book information.
4. The administrator/DEO makes the desired changes to the book information. This includes any of the information specified in the **Add a Book** subflow.
5. The system prompts the administrator/DEO to confirm the updation of the book.
6. After confirming the changes, the system updates the book information with the updated information.

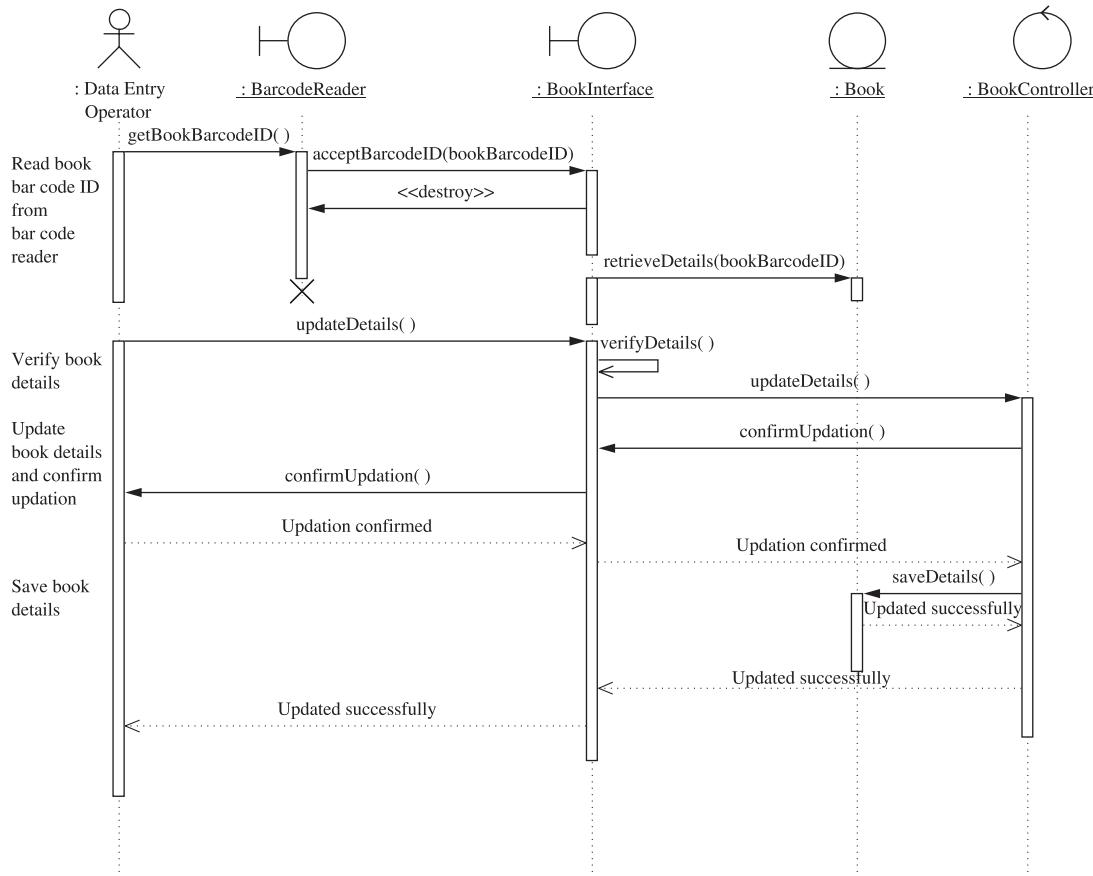


Figure 6.25 Sequence diagram of basic flow ‘update a book’.

The portion of the basic flow (basic flow 3: delete a book) is given below and its sequence diagram is shown in Figure 6.26.

Basic Flow 3: Delete a book

1. The system requests that administrator/DEO specify the accession number.
2. The administrator/DEO enters the accession number. The system retrieves and displays the required information.
3. The system prompts the administrator/DEO to confirm the deletion of the book record.
4. The administrator/DEO verifies the deletion.
5. The system deletes the record.

The portion of basic flow (basic flow 4: view a book) is given below and its sequence diagram is shown in Figure 6.27.

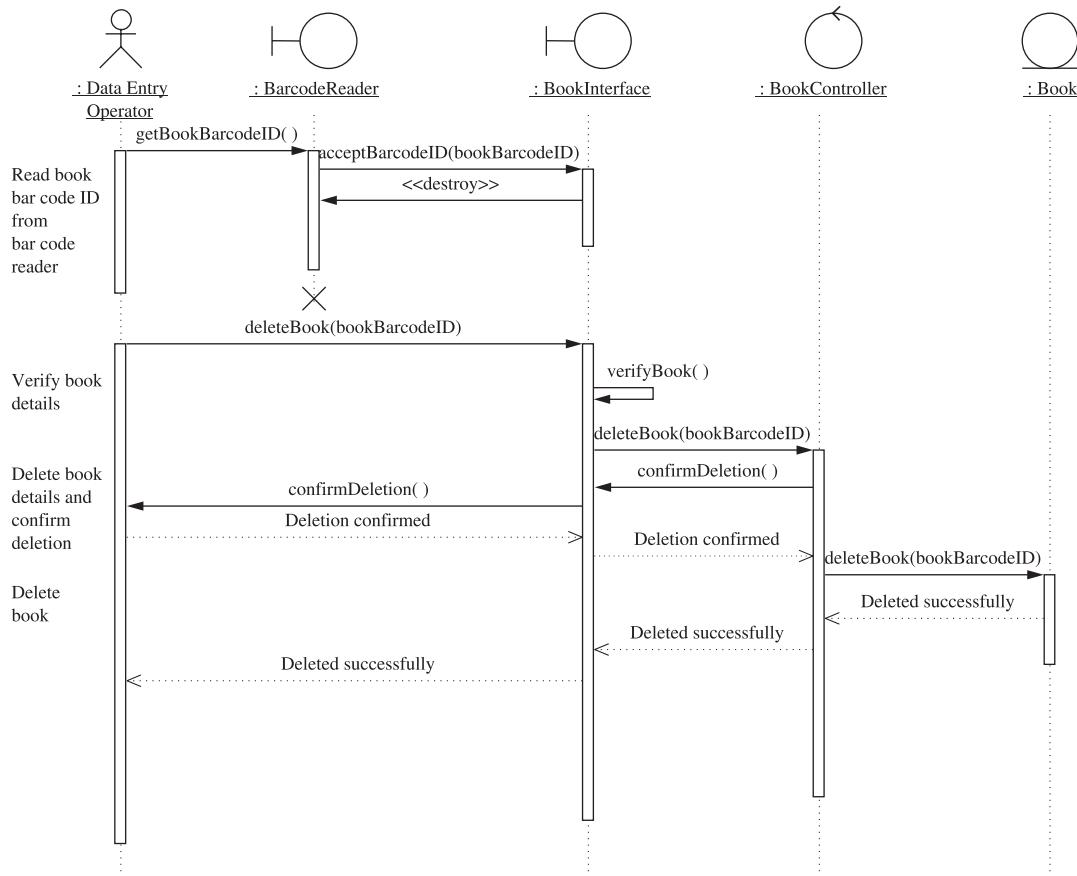


Figure 6.26 Sequence diagram of basic flow ‘delete a book’.

Basic Flow 4: View a book

1. The system requests that the administrator/DEO specify the accession number.
2. The system retrieves and displays the book information.

6.4 Collaboration Diagrams

Unlike in sequence diagrams, in collaboration diagrams, the **sequence of events is not time ordered**. Collaboration diagrams depict the flow of events of a scenario in a use case. It shows the interaction between objects by organizing them.

6.4.1 Objects, Links and Messages

Similar to the sequence diagrams, objects are depicted by rectangles (*objectname:classname*). The links between objects are depicted by arrows and the direction of the link is depicted

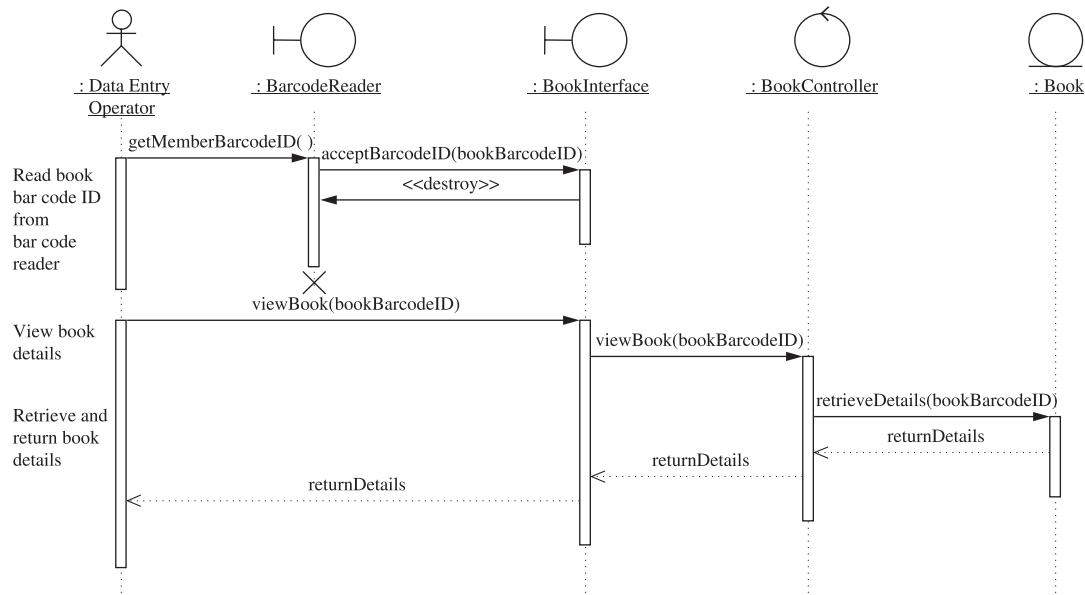


Figure 6.27 Sequence diagram of basic flow ‘view a book’.

by a directional arrow. The directional arrow shows the numbered message sent from the sending object to the receiving object as shown in Figure 6.28. The collaboration diagram gives a different view than the sequence diagram. Although the messages are shown in the collaboration diagram, the time for which the object is alive and the time till which an object participates in an operation are missing in this diagram. The objects have the same representation as in sequence diagrams. The parameters can be specified inside the parentheses of the message. The sequence number of the message is separated by the message name by a colon (:).

A link in a collaboration diagram can represent multiple messages, whereas a link in the sequence diagram can depict only one single message. Figure 6.29 shows that Object1 sends two messages to Object2 by using the same link: getData() and sendValue(a).

6.4.2 Creating Collaboration Diagrams

The objects participating in the use case must be identified in order to create a collaboration diagram. The objects for the ‘issue book’ use case have already been identified for the sequence diagram. The sequence of messages is determined and each link between the objects consists of the message that it sends during the interaction along with the sequence number of the message. The collaboration diagram for the ‘issue book’ use case is shown in Figure 6.30.

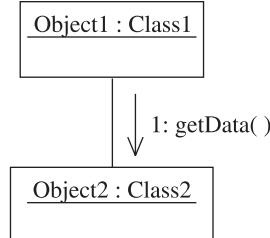


Figure 6.28 Collaboration diagram.

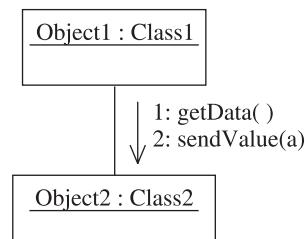


Figure 6.29 Sending multiple messages by a single link in a collaboration diagram.

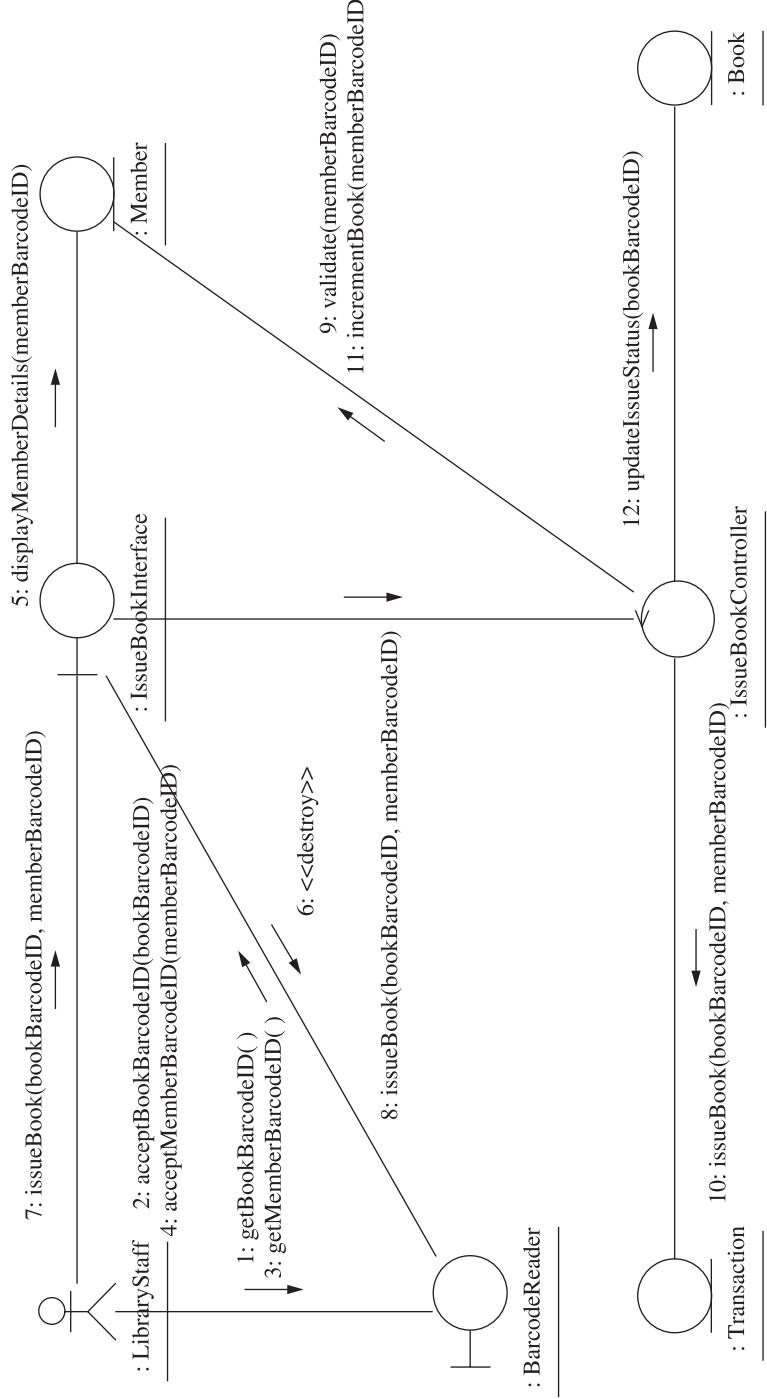


Figure 6.30 Collaboration diagram for basic flow of 'issue book' use case.

This diagram shows that the actor LibraryStaff initiates the interaction and sends message numbers 1 and 3 to the BarcodeReader object during the interaction. Similarly, other messages passed between the objects are depicted.

EXAMPLE 6.3 Consider the ‘login’ use case of the LMS. Draw the collaboration diagram for this use case.

Solution The collaboration diagram for the basic flow of ‘login’ use case is shown in Figure 6.31 and the collaboration diagram for alternative flow 1 of ‘login’ use case is shown in Figure 6.32.

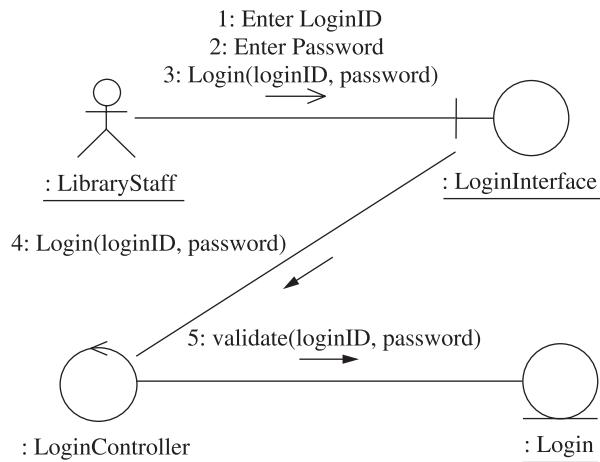


Figure 6.31 Collaboration diagram for basic flow of ‘login’ use case.

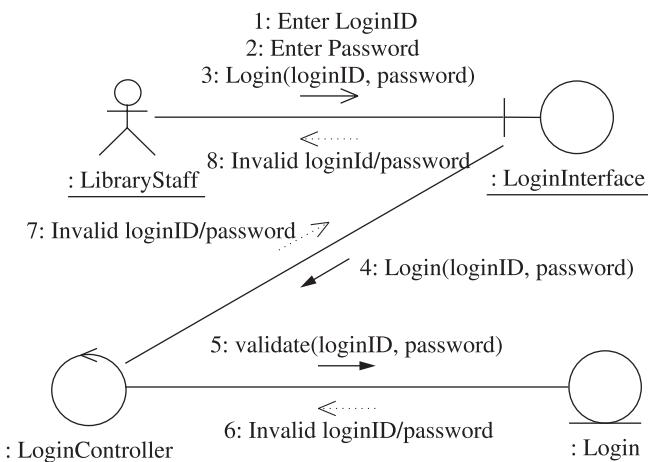


Figure 6.32 Collaboration diagram for alternative flow 1 of ‘login’ use case.

EXAMPLE 6.4 Consider the ‘return book’ use case of the LMS. Draw the collaboration diagram for this use case.

Solution The collaboration diagram for the basic flow of ‘return book’ use case is depicted in Figure 6.33.

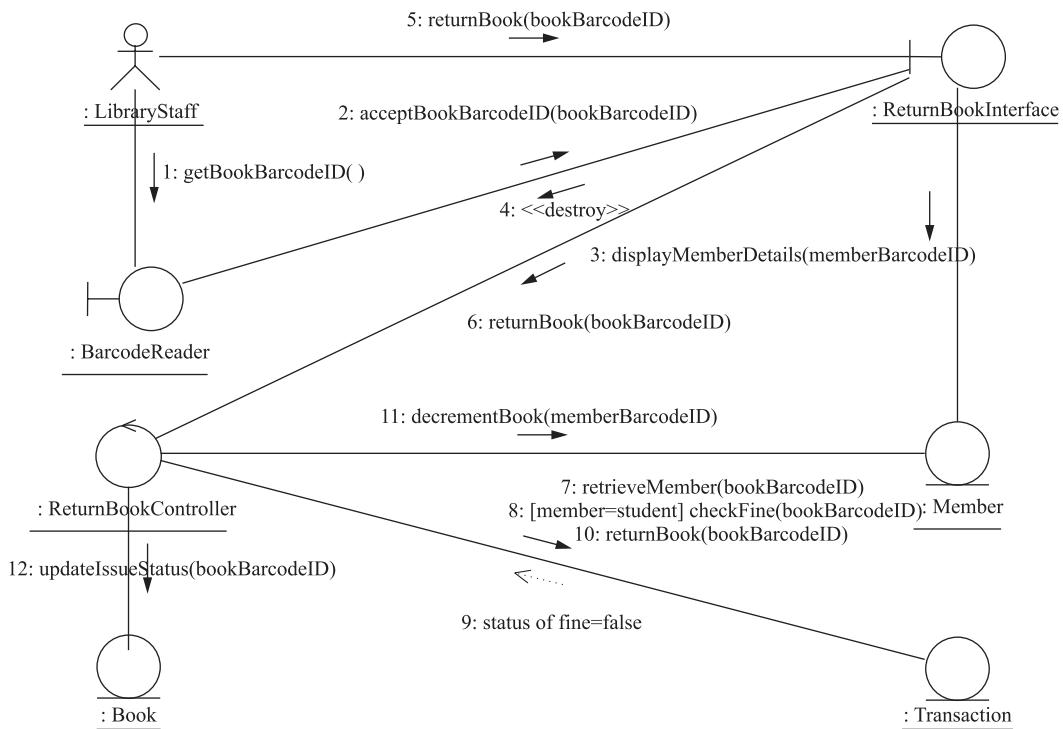


Figure 6.33 Collaboration diagram for basic flow of ‘return book’ use case.

6.5 Refinement of Use Case Description

The use cases are refined during the design phase when implementation environment and other details are considered. We may also consider the steps which an actor follows while actually interacting with the system and how the sequence of messages is sent amongst objects.

The flow of events is the very essential part of a use case. It usually describes the sequence of actions by the actors and the responses of the system to these actions. As the actions between the system and the actors are identified, the designer may have to modify the flow of events in the initial use case descriptions. In the UML, the interaction diagrams may prove to be very helpful in refining the flow of events in the use case descriptions. The understanding of the system increases as we create these diagrams.

The issue book use case is modified in the design phase. The modified basic flow with changes highlighted in bold letters is given as follows:

Basic Flow

1. The student/faculty/employee membership number is read through the bar code reader.
2. The system displays information about the student/faculty/employee.
3. Book information is read through the bar code reader.
- 4. The system checks the reservation status of the book.**
5. The system validates the membership and number of books issued in the account.
- 6. The system checks whether the member fine exceeds the maximum limit of the fine.**
7. The book is issued for the specified number of days and the return date of the book is calculated.
8. The book and student/faculty/employee information is saved into the database.
- 9. If the book is reserved by the same member, then it is marked as unreserved.**

Many times, exceptions and error conditions are missed in the initial use case description and identifying them is an important activity in refining the use case description. There is no limit to alternative flows and all of them must be discovered.

In the issue book use case, three alternative flows are added during the in-depth study of exceptional cases. The “Book is already reserved” alternative flow was not considered earlier which is quite possible in practice. If a student asks for a “reserved book”, permission for the same should be denied by the LMS. Alternative flow “fine exceeds the specified limit” is added in which a student may want to get a book issued while his/her fine limit has been exceeded. Another alternative flow “Unable to read entry” is also added in order to represent the exceptional case when the bar code of the book or member could not be read by the system. The permission of the requested operation in such a case should be denied by the system. We need to add these alternative flows as given in Figure 6.34.

Alternative Flow 1: Unauthorized member

If the system doesn't validate the member's (student/faculty/employee) membership number (due to membership expiry or any other reason), then an error message is flagged and the use case ends.

Alternative Flow 2: Account is full

If the student/faculty/employee has requested a book in Account and it is full, i.e. he/she has already the maximum number of books issued on his/her name, then the request for issue is denied and the use case ends.

Alternative Flow 3: Book is already reserved

If the book is already reserved by some other member of the library, then the request for issue is denied and the use case ends.

Alternative Flow 4: Fine exceeds the specified limit

If the fine of the student exceeds the specified limit, then the request for issue is denied and the use case ends.

Alternative Flow 5: Unable to read entry

If the barcodeID or memberID is unreadable, then an error message is flagged and the use case returns to the beginning of the basic flow.

Alternative Flow 6: Invalid entry

If in the issue book flow, the actor enters an invalid barcodeID and/or memberID or leaves the barcodeID and/or memberID empty, the system displays an error message. The actor returns to the beginning of the basic flow.

Alternative Flow 7: User exits

This allows the user to exit at any time during the use case. The use case ends.

Figure 6.34 Revised alternative flow of issue book.

Consider the scenario diagram given in Figure 3.9, the revised version of it is shown in Figure 6.35.

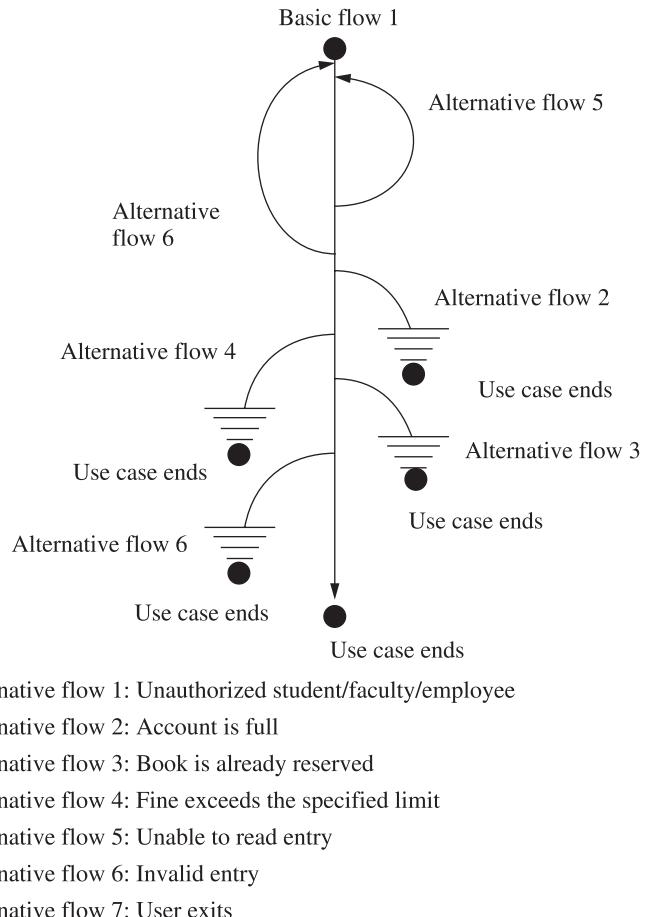


Figure 6.35 Revised basic and alternative flows for issue book use case.

The design phase updates the use cases and may modify other related diagrams. We may also find need of new use cases to simplify complex steps and increase the reusability. Thus, continuous refinement is a key to development of software through iterative process. The understanding of the system continuously increases and this brings changes in the various models and documents that are produced and finally results in the development of the source code. The use case descriptions are the medium through which our continuous increase in knowledge of the system is reflected. The revised sequence diagram and collaboration diagram according to the revised basic flow of the issue book use case are shown in Figures 6.36, 6.37 and 6.38, respectively. In Figure 6.36, the revised sequence diagram with textual messages is shown, whereas in Figure 6.37 the revised sequence diagram with operations is shown.

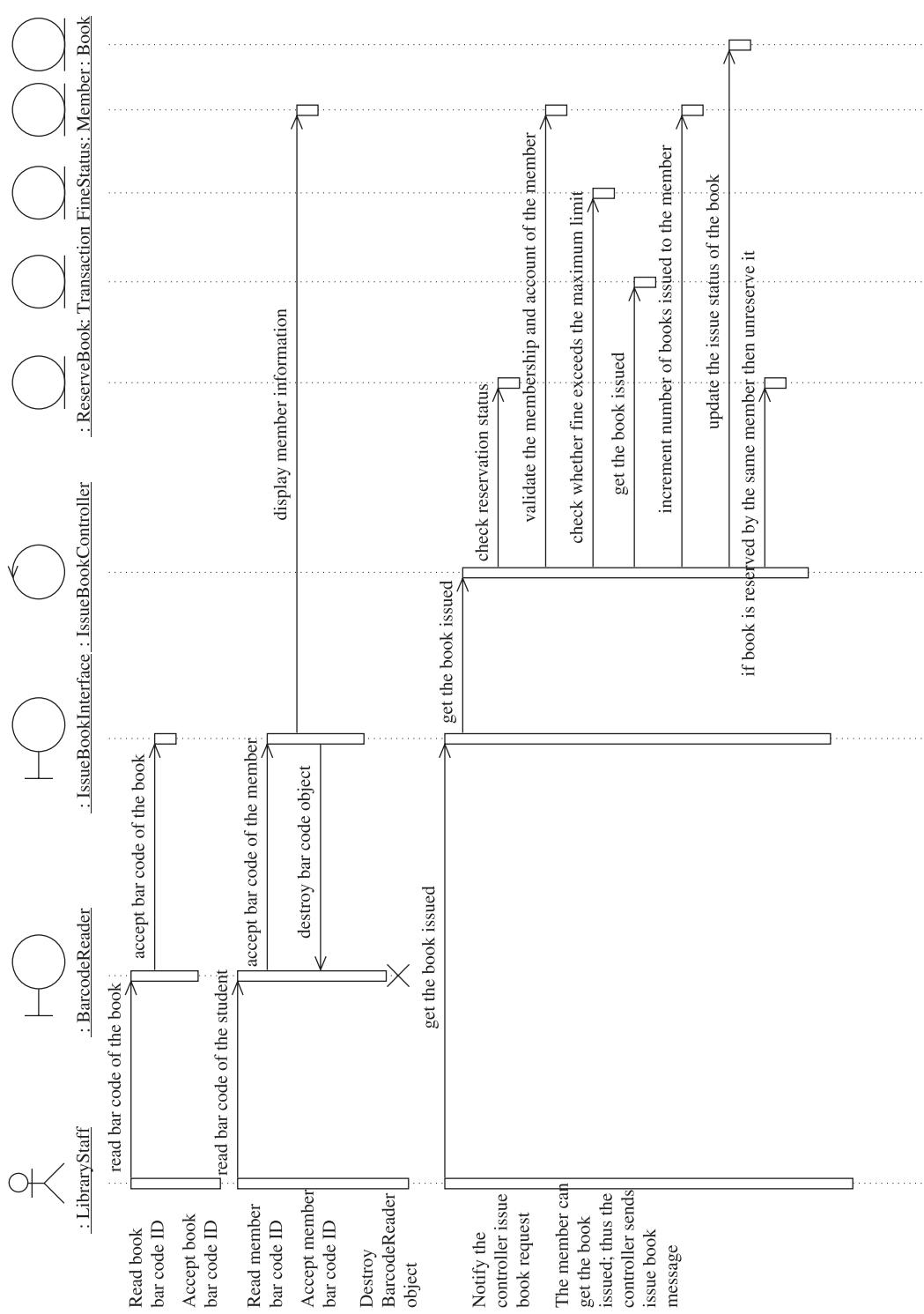


Figure 6.36 Revised sequence diagram of issue book basic flow with textual messages.

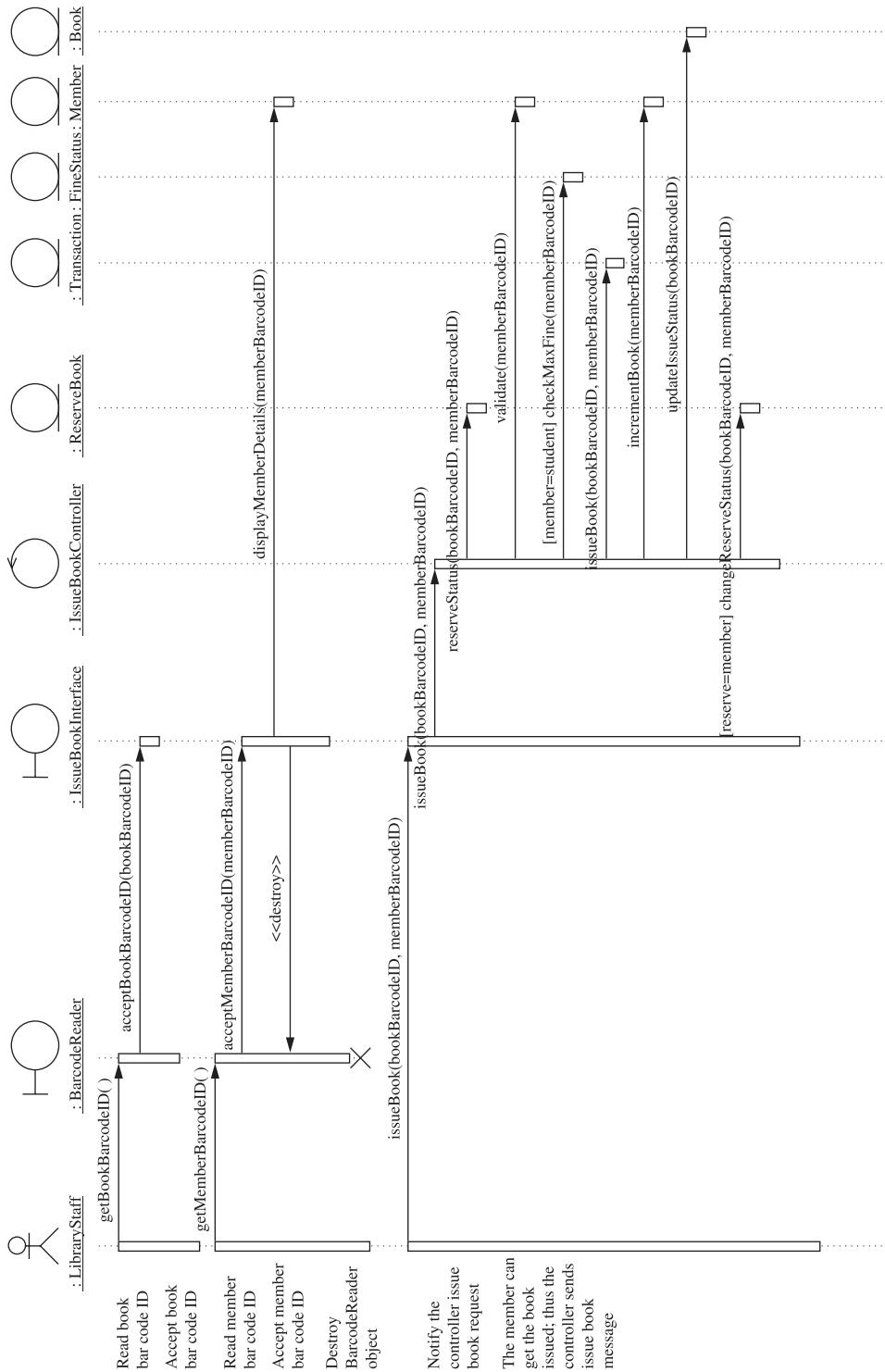


Figure 6.37 Revised sequence diagram of issue book basic flow with operations.

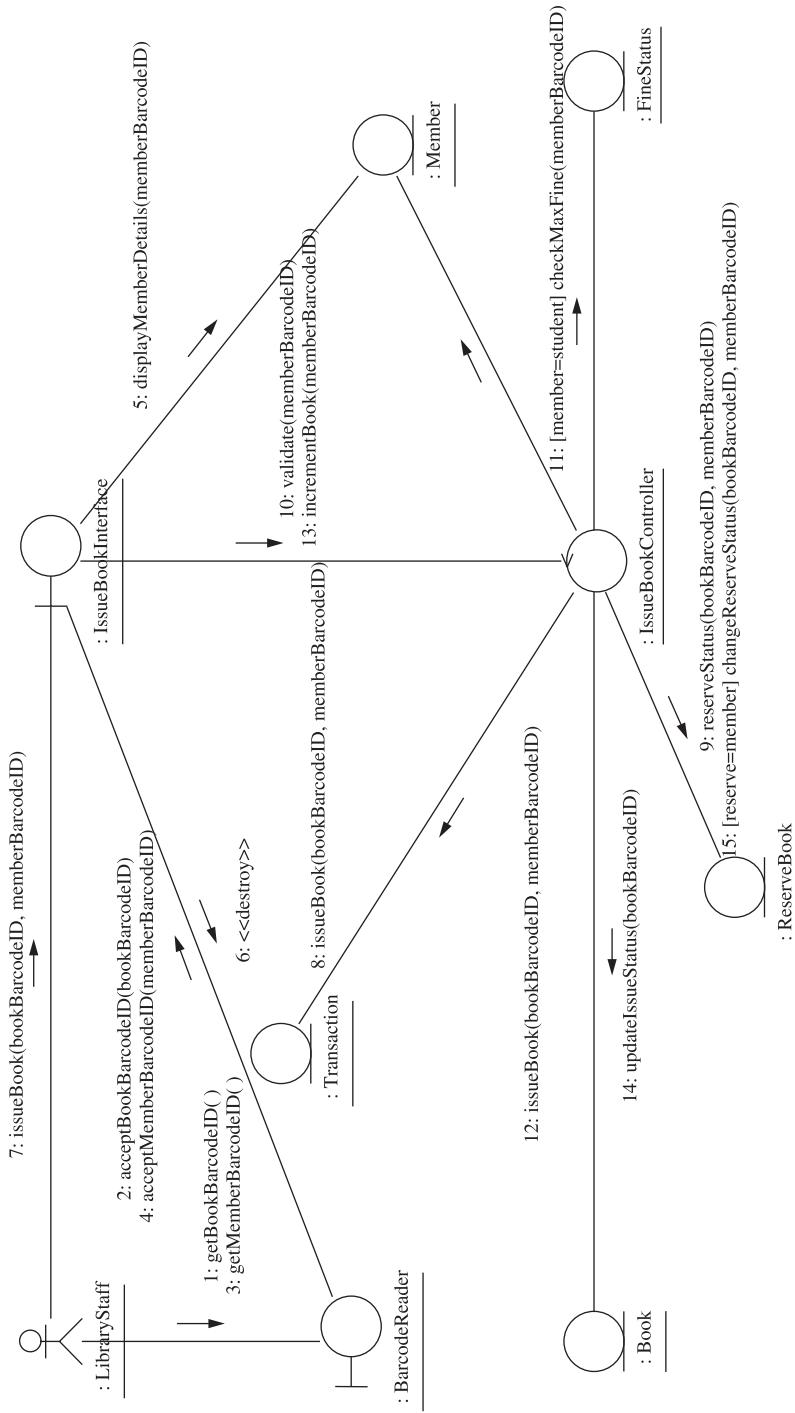


Figure 6.38 Revised collaboration diagram of issue book use case.

6.6 Refinement of Classes and Relationships

Classes are further refined after the modification of use cases, interaction diagrams including sequence diagrams and collaboration diagrams. The modification may make the system closer to reality. The revised portion of the class diagram of issue book use case is shown in Figure 6.39.

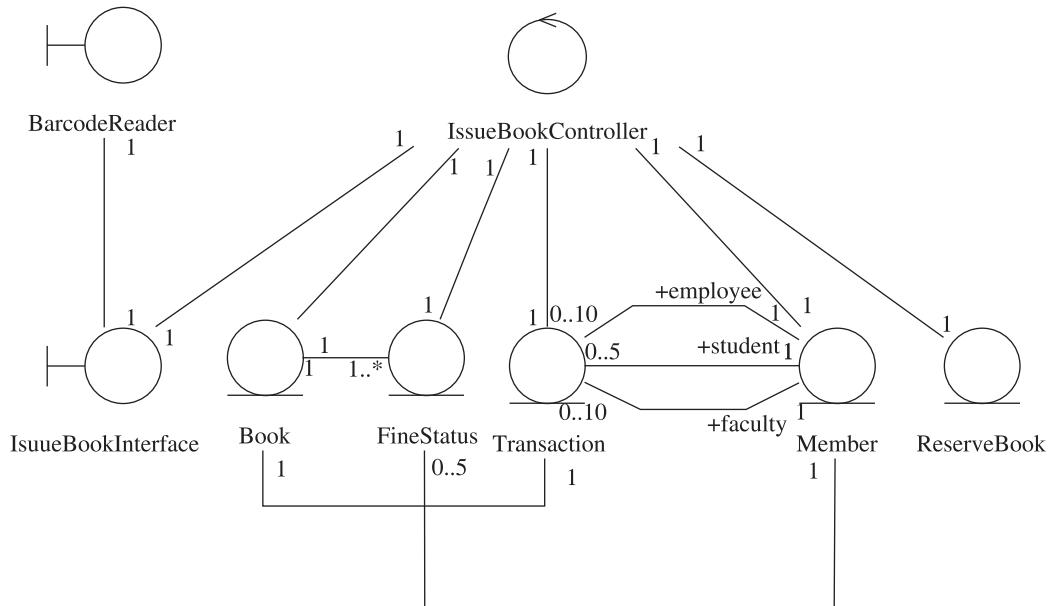


Figure 6.39 Revised class diagram of issue book use case.

6.7 Identification of Operations to Reflect the Implementation Environment

The messages (procedure call) sent from the **sender object to the receiver object in a sequence diagram become operations of the receiver object**. The operations of classes in the issue book use case are identified from the sequence diagram shown in Figure 6.37 and are given in Table 6.1.

The detailed list of operations along with their arguments, argument types and return values is shown in Table 6.1. The identification of all these details in the OOD phase will be very helpful for the programmer.

EXAMPLE 6.5 Consider the login use case of the LMS. Identify detailed operations for each class along with their parameters and return values. Describe the purpose of each operation.

Solution The detailed operations for each class involved in the login use case including parameters (with types) and return values are shown in Table 6.2.

Table 6.1 Operations in issue book use case

Class	Operation name	Parameter type	Return value	Return type	Description
BarcodeReader	getBookBarcodeID	—	—	—	This operation reads the bar code ID of the book.
	getMemberBarcodeID	—	—	—	This operation reads the member ID of the member.
	destroy	—	—	—	This operation destroys the object.
IssueBookInterface	acceptBookBarcodeID	bookBarcodeID	Long	—	This operation accepts bar code ID of book from the bar code reader.
IssueBookController	acceptMemberBarcodeID	memberBarcodeID	Long	—	This operation accepts the bar code ID of the member from the bar code reader.
	issueBook	bookBarcodeID memberBarcodeID	Long Long	—	This operation issues book to a member of the library.
	issueBook	bookBarcodeID memberBarcodeID	Long Long	—	This operation issues book to a member of the library.
Transaction	issueBook	bookBarcodeID memberBarcodeID	Long Long	—	This operation issues book to a member of the library.
ReserveBook	reserveStatus	bookBarcodeID memberBarcodeID	Long Long	Boolean	This operation checks whether the book has been reserved or not.
FineStatus	changeReserveStatus	bookBarcodeID memberBarcodeID	Long Long	Boolean	This operation checks whether the book has been reserved or not.
	checkMaxFine	memberBarcodeID	Long	Boolean (true/false)	This operation checks whether the fine of the member has reached its maximum limit.
Member	validate	memberBarcodeID	Long	Boolean (true/false)	This operation validates the membership and account of the member.
	incrementBook	memberBarcodeID	Long	—	This operation increments the number of books in the member account.
	displayMemberDetails	memberBarcodeID	Long	Member details object	This operation returns the details of the member.
Book	updateIssueStatus	bookBarcodeID	Long	—	This operation changes the issue status of the book.

Table 6.2 Operations in classes identified in login use case

Class	Operation name	Parameter	Parameter type	Return value	Return type	Description
LoginInterface	login	loginID	string	—	—	This operation allows the actor to login into the system.
		password	string	—	—	
LoginController	login	loginID	string	—	—	This operation allows the actor to login into the system.
		password	string	—	—	
Login	validate	loginID	string	—	—	This operation validates the login ID and password of the actor.
		password	string	Boolean	Boolean (true/false)	

EXAMPLE 6.6 Consider the return book use case of the LMS. Identify detailed operations, parameters and return values for each class involved in this use case. Describe the purpose of each operation.

Solution The detailed operations for each class involved in the return book use case including parameters (with types) and return values are shown in Table 6.3.

6.8 Construction of Detailed Class Diagram

The portions of the class diagram were created in the OOA phase. Now the classes are refined, i.e. new classes have been added, relationships and attributes are refined and a detailed set of operations are identified. The detailed class diagram is created in the OOD phase.

The detailed operations from interaction diagrams of each scenario were determined in Section 6.7. The visibility of the attributes and operations must also be identified and specified.

6.9 Development of Detailed Design and Creation of Software Design Document

A detailed design is constructed for all the classes identified in the system. The tabular representation of each class with detailed description can be used to construct the design. The detailed design is finally handed over to the programmer. The detailed design increases the understanding of the system and helps the programmer in developing the source code. The detailed design of each class will facilitate the developer's team to easily understand each individual class. The format of the detailed design for each class is shown in Table 6.4. The class type field specifies whether the class is an entity, interface or control. The data types of the attributes are specified. The operations with their full signature including parameters and return type are included.

The detailed designs of classes related to 'issue book' use, i.e. Transaction, Book, Member, BarcodeReader and IssueBookController classes are given in Tables 6.5–6.9.

Table 6.3 Operations in classes identified in return book use case

Class	Operation name	Parameter type	Parameter value	Return type	Description
BarcodeReader	getBookBarcodeID	—	—	—	This operation reads the bar code ID of the book.
	getMemberBarcodeID	—	—	—	This operation reads the member ID of the member.
	destroy	—	—	—	This operation destroys the object.
ReturnBookInterface	acceptBookBarcodeID	bookBarcodeID	Long	—	This operation accepts the bar code ID of the book from the bar code reader.
	acceptMemberBarcodeID	memberBarcodeID	Long	—	This operation accepts the bar code ID of the member from the bar code reader.
	returnBook	bookBarcodeID	Long	—	This operation issues book to a member of the library.
ReturnBookController	returnBook	memberBarcodeID	Long	—	This operation issues book to a member of the library.
	create	bookBarcodeID memberBarcodeID	Long Long	—	This operation creates the FineCalculator object.
	calculateFine	bookBarcodeID memberBarcodeID	Long Long	Fine amount	This operation calls calculate fine procedure of Transaction object.
Transaction	create	—	—	—	This operation creates the FineStatus object.
	destroy	—	—	—	This operation destroys the object.
	returnBook	bookBarcodeID memberBarcodeID	Long Long	—	This operation issues book to a member of the library.
retrieveMember	retrieveMember	bookBarcodeID bookBarcodeID	Long Long	—	This operation retrieves the member to whom book is to be returned.
	checkFine	bookBarcodeID memberBarcodeID	Long Long	—	This operation checks that whether a member has a fine or not.
	calculateFine	bookBarcodeID memberBarcodeID	Long Long	Fine amount	This operation calculates fine amount of the member and return it.
FineStatus	checkFine	memberBarcodeID	Long	Boolean	This operation checks whether the fine of the member has reached its maximum limit.
	saveFine	bookBarcodeID memberBarcodeID fine	Long Long integer	—	This operation saves the fine amount for the specific transaction.
	destroy	—	—	—	This operation destroys the object.
Member	decrementBook	memberBarcodeID	Long	—	This operation decrements the number of books in member account.
	updateIssueStatus	bookBarcodeID	Long	—	This operation changes the issue status of the book.

Table 6.4 Format of detailed design of a class

Class name	
Class type	
Description	
Attributes	Attribute 1 Attribute 2 ⋮ Attribute <i>n</i>
Operations	Operation 1 Operation 2 ⋮ Operation <i>n</i>

Table 6.5 Detailed design of Transaction class

Class name	Transaction
Class type	Entity
Description	This class is used to store the details of each transaction, i.e issue and return.
Attributes	Long bookBarcodeID Long memberBarcodeID Date dateOfIssue Date expectedDOR Date actualDOR
Operations	void issueBook(bookBarcodeID : Long, memberBarcodeID : Long) member retrieveMember(bookBarcodeID : Long) Boolean checkFine(bookBarcodeID) void returnBook(bookBarcodeID : Long)

Table 6.6 Detailed design of Book class

Class name	Book
Class type	Entity
Description	This class is used to store the details of each book in the library.
Attributes	Long bookBarcodeID String subjectDescriptor String ISBN String bookTitle String language String authorName String publisher Boolean issueStatus

(Contd.)

Table 6.6 Detailed design of Book class (*Contd.*)

Operations	void updateIssueStatus(bookBarcodeID : Long) void saveDetails() string deleteBook(bookBarcodeID : Long) string updateBook(bookBarcodeID : Long) string viewBook(bookBarcodeID : Long)
------------	---

Table 6.7 Detailed design of Member class

Class name	Member
Class type	Entity
Description	This class is used to store the details of each member in the library.
Attributes	Long memberBarcodeID String photograph String name String fname String DOB String phone String email Boolean memberdate validUpto noIssuedbooks
Operations	void displayMemberDetails(memberBarcodeID : Long) void validate(memberBarcodeID : Long) void incrementBook(memberBarcodeID : Long) void decrementBook(memberBarcodeID : Long) void addMember(m1:Member) void deleteMember(memberBarcodeID : Long) void updateMember(memberBarcodeID : Long)

Table 6.8 Detailed design of BarcodeReader class

Class name	BarcodeReader
Class type	Interface
Description	This class is used to read book and member bar code ID.
Attributes	Long ID
Operations	void getBookBarcodeID() void getMemberBarcodeID() void destroy()

Table 6.9 Detailed design of IssueBookController class

Class name	IssueBookController
Class type	Control
Description	This class is used to manage and control the sequence of flow of issue book use case.
Attributes	Long bookBarcodeID Long memberBarcodeID
Operations	void issueBook(bookBarcodeID : Long, memberBarcodeID : Long)

6.10 Generating Test Cases from Use Cases

Test cases are an integral part of any testing activity. The creation of test cases and their execution when the source code is available ensure that a robust, defect-free and high-quality system is constructed. A tester may ask the following questions:

- What is the exact functionality of the system?
- What are the exceptional or error conditions that may occur in the system?
- How to decide when the system has been tested completely?

The use cases consist of basic flow, alternative flow and special requirement and can be used to answer all of the above questions. If the test cases are derived from the use cases, it will help to improve the development process, testing efficiency and quality of the developed product. Use cases can be used to derive a large number of test cases in early phases of software development.

The test cases may be created from the use cases in the requirement analysis phase. However, we will start writing test cases now as the refined use cases are available with us.

6.10.1 Commonly Used Testing Terminology

Testing is a very important and essential activity that continues throughout the software development life cycle. Although the detailed process and activities of testing are covered in Chapter 9, for basic understanding, we will define some terms in this section.

- *Test case:* A test case may execute a particular path of the program or verify a given requirement of the system. A test case consists of input given to the program and its expected output from the program.
- *Test suite:* A test suite consists of a set of test cases. The test suite may consist of a set of successful test cases and a set of unsuccessful test cases.
- *Test design and procedure:* Test design and procedure consists of a detailed set of instructions for setting, designing and interpreting the results for a given test case.
- *Test coverage:* Test coverage defines the extent to which the test cases are covered by a given test for a given use case, class or system.
- *Test result:* Test results are a repository in which all the results and data produced during the execution of a program are kept.

After the construction of a detailed design, software design description (SDD) document may be created as per the IEEE standard (IEEE std. 1016-1998). The SDD document can serve as a primary medium for communicating software design information (Aggarwal et al., 2009). It simply translates the requirements into description of software attribute, operations and interfaces. The SDD document may serve as a blue print for implementing the software.

The relationships amongst the commonly used testing terms are shown in Figure 6.40.

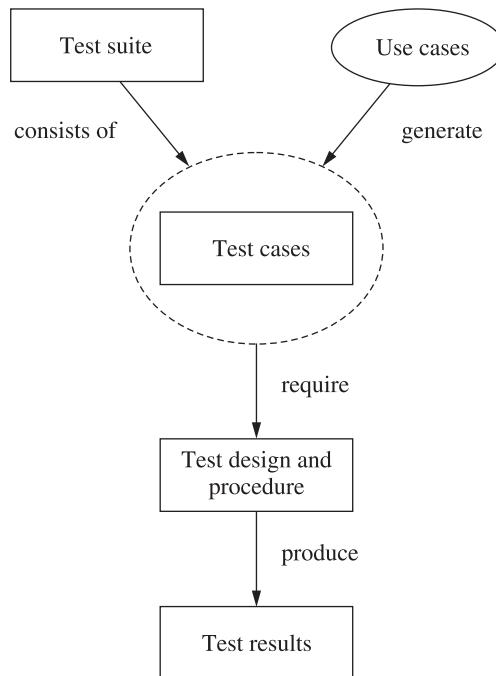


Figure 6.40 Relationship between testing terms.

The use cases generated and refined during requirement analysis, OOA and OOD phases can be directly used to generate the test cases and this reduces the effort to design and define the test cases in later phases of software development.

6.10.2 Deriving Test Cases from Use Cases: A Five-Step Process

A systematic approach may follow the following five steps in order to derive test cases from the use cases:

1. Creation of use case scenario matrix
2. Identification of variables in a use case
3. Identification of different input states of variables
4. Design of test case matrix
5. Assigning actual values to input variables

Step 1: Creation of Use Case Scenario Matrix

The use case scenario as described in Chapter 3 defines a path through a use case. The scenarios may consist of one or more basic flows and one or more alternative flows. A use case and scenarios have one to many relationships. A use case scenario matrix represents combinations of all possible basic and alternative flows. The format of a use case scenario matrix is given in Table 6.10.

Table 6.10 Format of use case scenario matrix

Scenario number and description	Originating flow	Alternative flow	Next alternative flow
Scenario 1			
Scenario 2			

The use case scenario matrix of the issue book use case scenario diagram shown in Figure 6.35 is given in Table 6.11.

In Table 6.11, we have identified eight possible scenarios for issue book use case from the use case description given in Figure 6.34.

Table 6.11 Scenario matrix for the issue book use case

Scenario number and description	Originating flow	Alternative flow
Scenario 1—Issue book basic flow	Basic flow 1	
Scenario 2—Issue book alternative flow: Unauthorized student/faculty/employee	Basic flow 1	Alternative flow 1
Scenario 3—Issue book alternative flow: Account is full	Basic flow 1	Alternative flow 2
Scenario 4—Issue book alternative flow: Book is already reserved	Basic flow 1	Alternative flow 3
Scenario 5—Issue book alternative flow: Fine exceeds the specified limit	Basic flow 1	Alternative flow 4
Scenario 6—Issue book alternative flow: Unable to read entry	Basic flow 1	Alternative flow 5
Scenario 7—Issue book alternative flow: Invalid entry	Basic flow 1	Alternative flow 6
Scenario 8—Issue book alternative flow: User exits	Basic flow 1	Alternative flow 7

Step 2: Identification of Variables in a Use Case

We have to identify all input variables which have been used in every use case. For an issue book use case, we use ‘bookBarcodeID’ and ‘memberBarcodeID’ as inputs for entering into the use case. These are two input variables for the issue book use case. There may be selection variables such as ‘updateConfirmed’ for update a book flow of maintain book details use case. These variables will have true/false values as inputs. These variables will be given as inputs to the

system and thus some response is expected from the system. Hence, it is important to identify the inputs corresponding to each use case.

Step 3: Identification of Different Input States of Variables

An input variable may have different states and the behaviour of the system may change if the state of a variable is changed. Any variable may have at least one of the two states—valid state and invalid state. In the test case matrix (see step 4), three types of values are used to specify the type of values expected by the test case: valid input, invalid input and N/A. These terms are defined below:

- Valid input indicates the input condition that must be true for the basic flow to execute.
- Invalid input indicates the input condition that must be true for the alternative flow to execute.
- N/A indicates that the corresponding input is not applicable to the particular test case for the given scenario.

For example, in the basic flow of issue book, we have two input variables: ‘bookBarcodeID’ and ‘memberBarcodeID’ and both must have a valid input condition in order to execute the flow.

Step 4: Design of Test Case Matrix

A test case consists of a set of input values, expected result and actual result. A common format for representing a test case corresponding to a given use case scenario is to use a test matrix. In the test matrix, the row represents a test case and the columns represent test case ID, scenario name and description, inputs, expected output, actual output and remarks. A typical format of a test case matrix is given in Table 6.12.

Table 6.12 Format of a test case matrix

Test case ID	Scenario name and description	Input 1	Input 2	Expected output	Actual output	Remarks (if any)
TC ₁	Scenario 1					
TC ₂	Scenario 2					
TC ₃	Scenario 3					
TC ₄	Scenario 3					

Table 6.12 shows that one scenario may represent multiple test cases. This situation arises when one scenario of a use case consists of various logical combinations. For example, in ‘issue book’ use case the alternative flow ‘Invalid entry’ states that:

If in the issue book flow, the actor enters an invalid barcodeID and/or memberID or leaves the barcodeID and/or memberID empty, the system displays an error message. The actor returns to the beginning of the basic flow.

This single use case scenario in the use case will produce four test cases. The test case matrix for issue book use case is given in Table 6.13.

Table 6.13 Test case matrix for the issue book use case

Test case ID	Scenario name and description	memberBarcodeID	bookBarcodeID	Expected output	Remarks (if any)
TC ₁	Scenario 1—Issue book basic flow	Valid input	Valid input	Book is issued successfully	—
TC ₂	Scenario 2— Issue book alternative flow: Unauthorized student/faculty/employee	Valid input	Valid/invalid input	Membership expired	The membership of a member in the library is not validated by the system.
TC ₃	Scenario 3— Issue book alternative flow: Account is full	Valid input	Valid input	Account full	The account of a member has reached the maximum limit of books that can be issued.
TC ₄	Scenario 4—Issue book alternative flow: Book is already reserved	Valid input	Valid input	Book is reserved	Book is already reserved by some other member in the library.
TC ₅	Scenario 5— Issue book alternative flow: Fine exceeds the specified limit	Valid input	Valid input	Fine exceeds the specified limit	The fine of the member exceeds the maximum permissible limit.
TC ₆	Scenario 6—Issue book alternative flow: Unable to read entry	Invalid input	Invalid input	Error	The specified bar code of the member/book is not in a readable format.
TC ₇	Scenario 7— Issue book alternative flow: Invalid entry	Valid/invalid input	Invalid input	Invalid book bar code	Book bar code is invalid.
TC ₈	Scenario 7— Issue book alternative flow: Invalid entry	Invalid input	Valid/invalid input	Invalid book bar code	Member bar code is invalid.
TC ₉	Scenario 7— Issue book alternative flow: Invalid entry	Valid input		Book bar code is blank	Book bar code cannot be blank.
TC ₁₀	Scenario 7— Issue book alternative flow: Invalid entry		Valid input	Member bar code is blank	Member barcode cannot be blank
TC ₁₁	Scenario 8— Issue book alternative flow: User exits	Valid input	Valid input	User comes out of the system	—

Step 5: Assigning Actual Values to Input Variables

Now we have created the scenario matrix, identified the variables with their input conditions, and created the test matrix. It is time now to provide actual data values to the input variables for each test case. It is not possible to execute test cases and generate test results without providing real data to a test case. Table 6.14 shows the test case matrix for issue book use case with actual input values. The actual output column may be added after executing the test cases.

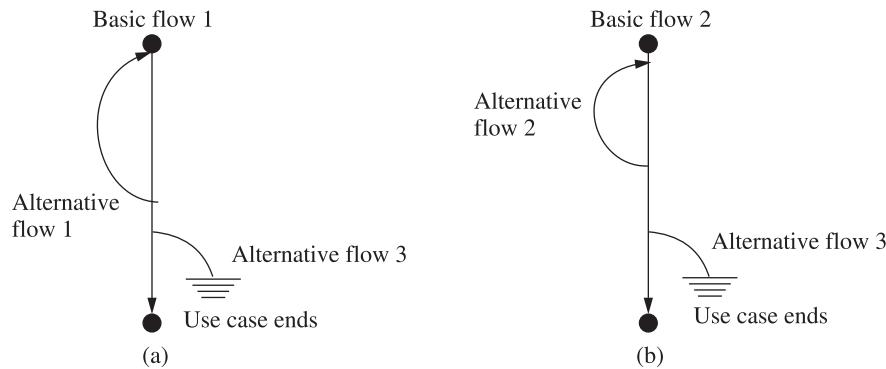
Table 6.14 Test case matrix for the issue book use case

Test case ID	Scenario name and description	memberBarcodeID	bookBarcodeID	Expected output	Remarks (if any)
TC ₁	Scenario 1—Issue book basic flow	1024	40925	Book is issued successfully	—
TC ₂	Scenario 2—Issue book alternative flow: Unauthorized student/faculty/employee	1035	*	Membership expired	The membership of a member in the library is not validated by the system.
TC ₃	Scenario 3—Issue book alternative flow: Account is full	1095	41256	Account full	The account of a member has reached the maximum limit of books that can be issued.
TC ₄	Scenario 4—Issue book alternative flow: Book is already reserved	1095	41257	Book is reserved	Book is already reserved by some other member in the library.
TC ₅	Scenario 5—Issue book alternative flow: Fine exceeds the specified limit	1089	455602	Fine exceeds the specified limit	The fine of the member exceeds the maximum permissible limit.
TC ₆	Scenario 6—Issue book alternative flow: Unable to read entry	—	—	Error	The specified bar code of the member/book is not in a readable format.
TC ₇	Scenario 7—Issue book alternative flow: Invalid entry	3456	456	Invalid book bar code	Book bar code is invalid.
TC ₈	Scenario 7—Issue book alternative flow: Invalid entry	3	45667	Invalid book bar code	Member bar code is invalid.
TC ₉	Scenario 7—Issue book alternative flow: Invalid entry	3456		Book bar code is blank	Book bar code cannot be blank.
TC ₁₀	Scenario 7—Issue book alternative flow: Invalid entry		56788	Member bar code is blank	Member bar code cannot be blank.
TC ₁₁	Scenario 8—Issue book alternative flow: User exits	3456	56789	User comes out of the system	—

* implies do not care, — represents unreadable input

EXAMPLE 6.7 Consider the problem statement of the LMS as given in Chapter 3. Construct the scenario diagram and generate test cases from the login use case.

Solution The scenario diagram of the login use case is shown in Figure 6.41. The scenario matrix is given in Table 6.15.



Alternative flow 1: Invalid login ID/password

Alternative flow 2: Invalid entry

Alternative flow 3: User exits

Figure 6.41 Basic and alternative flows for login use case:
 (a) Login and (b) change password.

Table 6.15 Scenario matrix for the login use case

Scenario 1—Login	Basic flow 1	
Scenario 2—Login alternative flow: Invalid login ID/password	Basic flow 1	Alternative flow 1
Scenario 3—Login alternative flow: User exit	Basic flow 1	Alternative flow 3
Scenario 4—Change Password	Basic flow 2	
Scenario 5—Change password alternative flow: Invalid entry	Basic flow 2	Alternative flow 2
Scenario 6—Change password alternative flow: User exits	Basic flow 2	Alternative flow 3

The test case matrix for the login use case is given in Table 6.16. The test cases with actual values for the login use case are given in Table 6.17.

6.11 Object-Oriented Design Principles for Improving Software Quality

Coad and Yourdon (1991) defined a set of design principles for improving the quality of a software product. These principles will result in the development of a software product with better object-oriented design. They provided guidelines related to cohesion, coupling, inheritance and clarity of design. These guidelines include:

- (i) **Cohesion:** The attributes and operations in a class should be highly cohesive. Each operation should be designed to fulfil one single purpose.

Table 6.16 Test case matrix for the login use case

Test case ID	Scenario name and description	Input 1 Login ID	Input 2 Password	Input 3 Old password	Input 4 New password	Input 5 Confirm password	Expected output	Remarks (if any)
TC ₁	Scenario 1—Login	Valid input	Valid input	n/a	n/a	n/a	User is allowed to login	—
TC ₂	Scenario 2—Login alternative flow: Invalid entry	Invalid input	Valid input	n/a	n/a	n/a	Login ID invalid	Login ID is not in the specified format.
TC ₃	Scenario 2—Login alternative flow: Invalid entry	Valid input	Valid input	n/a	n/a	n/a	Login ID invalid	Login ID does not exist in database.
TC ₄	Scenario 2—Login alternative flow: Invalid entry	Valid input	Invalid input	n/a	n/a	n/a	Password invalid	Password is not in the specified format.
TC ₅	Scenario 2—Login alternative flow: Invalid entry	Valid input	Valid input	n/a	n/a	n/a	Password invalid	Password does not exist in database.
TC ₆	Scenario 2—Login alternative flow: Invalid entry	Invalid input	Invalid input	n/a	n/a	n/a	Login ID and password invalid	Login ID and password are not in the specified format.
TC ₇	Scenario 3—Login alternative flow: User exits	Valid/invalid input	invalid input	n/a	n/a	n/a	User comes out of the system	—
TC ₈	Scenario 4—Change password	Valid input	n/a	Valid input	Valid input	Valid input	User is allowed to change password	Password is changed in the database.
TC ₉	Scenario 5—Change password alternative flow: Invalid entry	Invalid input	n/a	Valid/invalid input	Valid/invalid input	Valid/invalid input	Old password invalid	Login ID is not in the specified format.
TC ₁₀	Scenario 5—Change password alternative flow: Invalid entry	Valid input	n/a	Invalid input	Valid/invalid input	Valid/invalid input	Old password invalid	If the old password is not valid, other entries become ‘do not care’ entries.
TC ₁₁	Scenario 5—Change password alternative flow: Invalid entry	Valid input	n/a	Valid input	Invalid input	Valid/invalid input	New password invalid	Password is not in the specified format.
TC ₁₂	Scenario 5—Change password alternative flow: Invalid entry	Valid input	n/a	Valid input	Valid input	Valid input	Confirm password does not match new password	New and confirm password entries are different.
TC ₁₃	Scenario 6—Change password alternative flow: User exits	Valid/invalid input	n/a	Valid/invalid input	Valid/invalid input	Valid/invalid input	User is allowed to exit and returns to login screen	—

Table 6.17 Test case matrix with actual data values for the login use case

Test case ID	Scenario name and description	Login ID	Password	Old password	New password	Confirm password	Expected output	Remarks (if any)
TC ₁	Scenario 1—Login	01164521657	Abc123	n/a	n/a	n/a	User is allowed to login	—
TC ₂	Scenario 2—Login alternative flow: Invalid entry	1234	Abc123	n/a	n/a	n/a	Login ID invalid	Login ID is not in specified format which is less than 11 characters.
TC ₃	Scenario 2—Login alternative flow: Invalid entry	01164521658	Abc123	n/a	n/a	n/a	Login ID invalid	Login ID does not exist in database.
TC ₄	Scenario 2—Login alternative flow: Invalid entry	01164521657	R34	n/a	n/a	n/a	Password invalid	Password is not in specified format which is less than 4 characters.
TC ₅	Scenario 2—Login alternative flow: Invalid entry	01164521657	Abc124	n/a	n/a	n/a	Login ID invalid	Password does not exist in database.
TC ₆	Scenario 2—Login alternative flow: Invalid entry	1234	R34	n/a	n/a	n/a	Login ID/password invalid	Login ID and password are not in the specified format. Login ID is less than 11 characters and password is less than 4 characters.
TC ₇	Scenario 3—Login alternative flow: User exits	*	*	n/a	n/a	n/a	User comes out of the system	—
TC ₈	Scenario 4—Change password	01164521657	n/a	Abc123	Abc124	Abc124	User is allowed to change password	—
TC ₉	Scenario 5—Change password alternative flow: Invalid entry	01165	n/a	*	*	*	Login ID is invalid	Login ID is not in the specified format.
TC ₁₀	Scenario 5—Change password alternative flow: Invalid entry	01164521657	n/a	Abc1	*	*	Old password invalid	Old password does not match the corresponding password in the database. Other entries (new password and confirm password) become ‘do not care’.
TC ₁₁	Scenario 5—Change password alternative flow: Invalid entry	01164521657	n/a	Abc123	R12	*	New password invalid	New password is not in the specified format which is less than 4 characters. Other entries (confirm password) become ‘do not care’.
TC ₁₂	Scenario 5—Change password alternative flow: Invalid entry	01164521657	n/a	Abc123	Abc124	Abc125	Confirm password does not match new password	—
TC ₁₃	Scenario 6—Change password alternative flow: User exits	*	n/a	*	*	*	User is allowed to exit and returns to login screen	—

*: do not care conditions (valid/invalid inputs); n/a: option(s) not available for respective scenario

- (ii) *Coupling:* Interaction coupling between classes should be kept minimum. The number of messages (sent and received) between classes should be reduced. Inheritance-based coupling between classes should be increased.
- (iii) *Design clarity:* The vocabulary used should be correct, concise, unambiguous and consistent. The names of the classes along with their attributes and operations should convey their intended meaning. The meaning and purpose of the class along with its responsibilities should be clear.
- (iv) *Class hierarchy depth:* The concept of generalization-specialization should be used wherever it is necessary. In other words, unnecessary and excessive use of inheritance (only for the sake of increasing reusability) should be avoided. Inheritance hierarchy should represent solution to a problem.
- (v) *Simple classes and objects:* Excessive attributes should be avoided in a class. Class definition should be simple, clear, concise and understandable.

According to Coad and Yourdon, adherence to the above design guidelines for the object-oriented software will lead to a better design and hence improves software quality and produces a maintainable system.

EXAMPLE 6.8 Consider the use case description of “Maintain Book Details” use case as given in Chapter 3. Construct the scenario diagram and generate test cases from this use case.

Solution The scenario diagram of the “Maintain Book Details” use case is shown in Figure 6.42. The scenario matrix is given in Table 6.18.

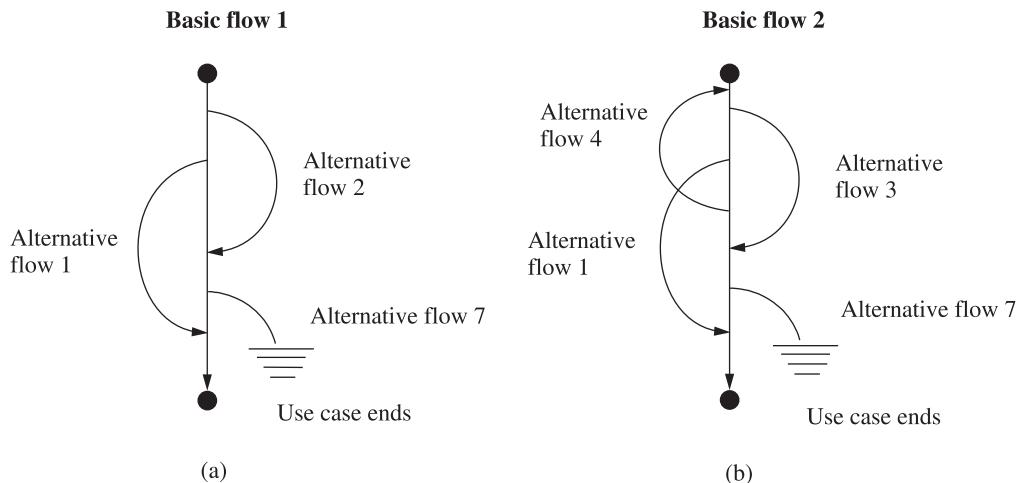
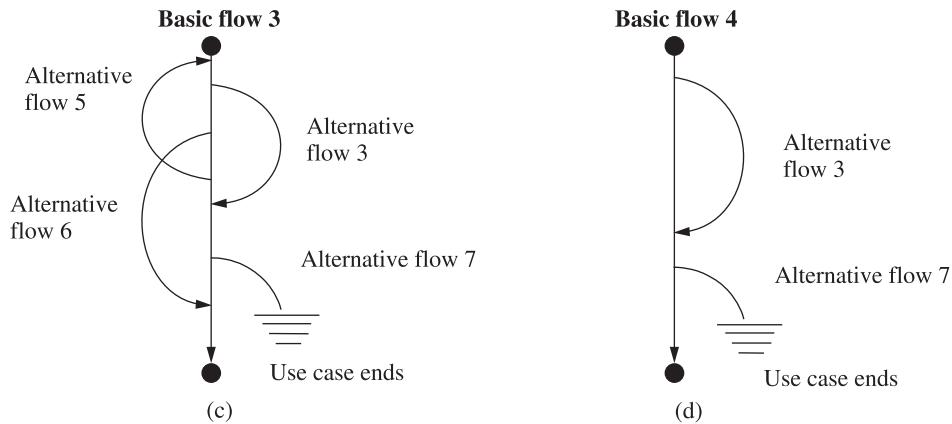


Figure 6.42 (Contd.)

**Maintain Book Details**

- Alternative flow 1: Invalid entry
- Alternative flow 2: Book already exists
- Alternative flow 3: Book not found
- Alternative flow 4: Update cancelled
- Alternative flow 5: Delete cancelled
- Alternative flow 6: Deletion not allowed
- Alternative flow 7: User exits

Figure 6.42 Alternative flows for Maintain Book Details use case: (a) Add details, (b) Update details, (c) Delete details and (d) View details.

Table 6.18 Scenario matrix for the Maintain Book Details use case

Scenario 1—Add a book	Basic flow 1	
Scenario 2—Add a book alternative flow: Invalid entry	Basic flow 1	Alternative flow 1
Scenario 3—Add a book alternative flow: Book code already exists	Basic flow 1	Alternative flow 2
Scenario 4—Add a book alternative flow: User exits	Basic flow 1	Alternative flow 7
Scenario 5—Update a book	Basic flow 2	
Scenario 6—Update a book alternative flow: Invalid entry	Basic flow 2	Alternative flow 1
Scenario 7—Update a book alternative flow: Book not found	Basic flow 2	Alternative flow 3
Scenario 8—Update a book alternative flow: Update cancelled	Basic flow 2	Alternative flow 4
Scenario 9—Update a book alternative flow: User exits	Basic flow 2	Alternative flow 7
Scenario 10—Delete a book	Basic flow 3	
Scenario 11—Delete a book alternative flow: Book not found	Basic flow 3	Alternative flow 3
Scenario 12—Delete a book alternative flow: Delete cancelled	Basic flow 3	Alternative flow 5
Scenario 13—Delete a book alternative flow: Delete not allowed	Basic flow 3	Alternative flow 6
Scenario 14—Delete a book alternative flow: User exits	Basic flow 3	Alternative flow 7
Scenario 15—View a book	Basic flow 4	
Scenario 16—View a book alternative flow: Book not found	Basic flow 4	Alternative flow 3
Scenario 17—View a book alternative flow: User exits	Basic flow 4	Alternative flow 7

As shown in Table 6.18, there are 17 scenarios for four basic flows of Maintain Book Details use case. For Maintain Book Details use case, we identify nine input variables for various flows in the use case. There are seven input variables and two selection variables (update confirmed, delete confirmed) in this use case. These inputs will be available in the required flows as specified in the use case.

There are 28 test cases created for the given 17 scenarios as shown in Table 6.19. Seven test cases are designed for scenario 2. After constructing these test cases, actual input values are given to all the variables in order to generate actual output and verify whether test case passes or fails (refer to Table 6.20).

Table 6.19 Test case matrix for the Maintain Book Details use case

Test case ID	Scenario and description	Input 1 Accession number	Input 2 Subject descriptor	Input 3 ISBN	Input 4 title	Input 5 language	Input 6 author	Input 7 publisher	Update confirmed	Deletion confirmed	Expected result	Remarks (if any)
TC ₁	Scenario 1—Add a book	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	n/a	n/a	Book is added – successfully	
TC ₂	Scenario 2—Add a book alternative flow: Invalid entry	Invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid book code	Book code is not in the specified format. Book name becomes 'do not care'.
TC ₃		Valid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid subject descriptor	Subject descriptor is not in the specified format.
TC ₄		Valid input	Valid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid ISBN	ISBN is not in the specified format.
TC ₅		Valid input	Valid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid title	Book title is not in the specified format.
TC ₆		Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	n/a	n/a	Invalid language	Language is not in the specified format.
TC ₇		Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	n/a	n/a	Invalid author name	Author name is not in the specified format.
TC ₈		Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	n/a	n/a	Invalid publisher name	Publisher is not in the specified format.
TC ₉	Scenario 3—Add a book alternative flow: Book code already exists	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	n/a	n/a	Book	The book with the same code is already present in the database.
TC ₁₀	Scenario 4—Add a book alternative flow: User exits	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	User is allowed to exit and returns to the Main menu	–

(Contd.)

Table 6.19 Test case matrix for the Maintain Book Details use case (*Contd.*)

Test case ID	Scenario and description	Input 1 Accession number	Input 2 Subject descriptor	Input 3 ISBN	Input 4 title	Input 5 language	Input 6 author	Input 7 publisher	Update confirmed	Deletion confirmed	Expected result	Remarks (if any)
TC ₁₁	Scenario 5—Update a book	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Yes	n/a	Book is updated	—
TC ₁₂	Scenario 6—Update a book alternative flow: Invalid entry	Valid input	Invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Subject descriptor is not in the specified format.	Subject descriptor is not in the specified format.
TC ₁₃		Valid input	Valid input	Invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid ISBN	ISBN is not in the specified format.
TC ₁₄		Valid input	Valid input	Valid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid title	Book title is not in the specified format.
TC ₁₅		Valid input	Valid input	Valid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid language	Language is not in the specified format.
TC ₁₆		Valid input	Valid input	Valid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid author name	Author name is not in the specified format.
TC ₁₇		Valid input	Valid input	Valid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	Invalid publisher name	Publisher is not in the specified format.
TC ₁₈	Scenario 7—Update a book alternative flow: Book not found	Valid input	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Book not found	Book with the specified code does not exist in the database.
TC ₁₉	Scenario 8—Update cancelled	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	Valid input	No	n/a	Main screen of book appears	—
TC ₂₀	Scenario 9—Update a book alternative flow: User exits	Valid/ invalid input	Valid/ invalid input	Valid/invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	Valid/ invalid input	n/a	n/a	User is allowed to exit and returns to the Main menu	—
TC ₂₁	Scenario 10—Delete a book	Valid input	n/a	n/a	n/a	n/a	n/a	n/a	Yes	n/a	Book is deleted successfully	—

(Contd.)

Table 6.19 Test case matrix for the Maintain Book Details use case (*Contd.*)

Test case ID	Scenario and description	Input 1 Accession number	Input 2 Subject descriptor	Input 3 ISBN	Input 4 title	Input 5 language	Input 6 author	Input 7 publisher	Update confirmed	Deletion confirmed	Expected result	Remarks (if any)
TC ₂₂	Scenario 11—Delete a book alternative flow: Book not found	Valid input	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Book not found	Book with the specified code does not exist in the database.
TC ₂₃	Scenario 12—Delete a book alternative flow: Delete cancelled	Valid input	n/a	n/a	n/a	n/a	n/a	n/a	No	Main screen of book appears	User does not confirm the delete operation.	
TC ₂₄	Scenario 13—Delete a book alternative flow: Deletion not allowed	Valid input	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Deletion not allowed	Book is already issued.	
TC ₂₅	Scenario 14—Delete a book alternative flow: User exits	Valid / invalid input	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	User is allowed to exit and returns to the Main menu	—
TC ₂₆	Scenario 15—View a book	Valid input	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Book is displayed successfully	The book details are displayed.
TC ₂₇	Scenario 16—View a book alternative flow: Book not found	Valid input	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Book not found	Book with the specified accession number does not exist in the database.
TC ₂₈	Scenario 17—View a book alternative flow: User exits	Valid / invalid input	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	User is allowed to exit and returns to the Main menu	—

n/a: option(s) not available for respective scenario.

Table 6.20 Test case matrix with actual data values for the Maintain Book Details use case

Test case ID	Scenario and description	Input 1 accession number	Input 2 subject descriptor	Input 3 ISBN	Input 4 title	Input 5 language	Input 6 author	Input 7 publisher	Update confirmed	Deletion confirmed	Expected result	Remarks (if any)	
TC ₁	Scenario 1—Add a book	456789	Computer science	81-224-2360-4	Software engineering	English	Pressman	TMH	n/a	n/a	Book is added successfully	—	
TC ₂	Scenario 2—Add a book alternative flow:	Rt5	*	*	*	*	*	*	n/a	n/a	Invalid book code	Book code is not in the specified format.	
TC ₃	Invalid entry	456789	567	*	*	*	*	*	n/a	n/a	Invalid subject descriptor	Subject descriptor is not in the specified format.	
TC ₄		456789	Computer science	Rty	*	*	*	*	n/a	n/a	Invalid ISBN	ISBN is not in the specified format.	
TC ₅		456789	Computer science	81-224-2360-4	Software engineering	English	Pressman	TMH	n/a	n/a	Invalid title	Book title is not in the specified format.	
TC ₆		456789	Computer science	81-224-2360-4	Software engineering	English	789	*	n/a	n/a	Invalid language	Language is not in the specified format.	
TC ₇		456789	Computer science	81-224-2360-4	Software engineering	English	890	*	n/a	n/a	Invalid author	Author name is not in the specified format.	
TC ₈		456789	Computer science	81-224-2360-4	Software engineering	English	Pressman	TMH	n/a	n/a	Invalid publisher	Publisher is not in the specified format.	
TC ₉	Scenario 3—Add a book alternative flow:	456789	Computer science	81-224-2360-4	Software engineering	English	Pressman	TMH	n/a	n/a	Book accession number already exists	The book with the same code is already present in the database.	
TC ₁₀	Book code already exists	*	*	*	*	*	*	*	n/a	n/a	User is allowed to exit and returns to the Main menu	—	
TC ₁₁	Scenario 4—Add a book alternative flow:	User exists	456789	Computer science	81-224-2360-4	Software engineering	English	Pressman	TMH	Yes	n/a	Book is updated successfully	—
TC ₁₂	Scenario 5—Update a book	456789	Invalid input	*	*	*	*	*	n/a	n/a	Invalid subject descriptor	Subject descriptor is not in the specified format.	
TC ₁₃	Scenario 6—Update a book alternative flow:	Invalid entry	456789	Computer science	yu	*	*	*	n/a	n/a	Invalid ISBN	ISBN is not in the specified format.	
TC ₁₄		456789	Computer science	81-224-2360-4	Software engineering	English	Pressman	TMH	n/a	n/a	Invalid title	Book title is not in the specified format.	
TC ₁₅		456789	Computer science	81-224-2360-4	Software engineering	English	78	*	n/a	n/a	Invalid language	Language is not in the specified format.	
TC ₁₆		456789	Computer science	81-224-2360-4	Software engineering	English	9op	*	n/a	n/a	Invalid author name	Author name is not in the specified format.	

(Contd.)

Test case ID	Scenario and description	Input 1 Accession number	Input 2 Subject descriptor	Input 3 ISBN	Input 4 title	Input 5 language	Input 6 author	Input 7 publisher	Update confirmed	Deletion confirmed	Expected result	Remarks (if any)
TC ₁₇		456789	Computer science	81-224 n/a	Software -2360-4 engineering	English n/a	Pressman	7uy0	n/a	n/a	Invalid publisher	Publisher is not in the specified format.
TC ₁₈	Scenario 7—Update a book alternative flow:	456789	Computer science	81-224 n/a	Software -2360-4 engineering	English n/a	Pressman	TMH	No	n/a	Main screen of book appears	Book with the specified code does not exist in the database.
TC ₁₉	Book not found						*	*	*	n/a	User is allowed to exit and returns to Main menu	—
TC ₂₀	Scenario 8—Update cancelled	456789	Computer science	81-224 n/a	Software -2360-4 engineering	English n/a	Pressman	TMH	No	n/a	User is allowed to exit and returns to Main menu	—
TC ₂₁	Scenario 9—Update a book alternative flow:	*	*	*	*	*	*	*	*	n/a	Book deleted successfully	—
TC ₂₂	User exits								n/a	Yes	Book deleted successfully	Book with the specified code does not exist in the database.
TC ₂₃	Scenario 10—Delete a book alternative flow:	456789	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Book not found	Book with the specified code does not exist in the database.	—
TC ₂₄	Scenario 11—Delete a book alternative flow:	456789	n/a	n/a	n/a	n/a	n/a	n/a	n/a	No	Main screen of book appears	User does not confirm the delete operation.
TC ₂₅	Book not found								n/a	n/a	Deletion not allowed	Book is already issued.
TC ₂₆	Scenario 12—Delete a book alternative flow:	456789	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	User is allowed to exit and returns to the Main menu	—
TC ₂₇	Scenario 13—Delete a book alternative flow:	*	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Book is displayed successfully	The book details are displayed.
TC ₂₈	Deletion not allowed								n/a	n/a	Book not found	Book with the specified accession number does not exist in the database.
	Scenario 14—Delete a book alternative flow:								n/a	n/a	User is allowed to exit and returns to the Main menu	—
	User exits								n/a	n/a	Book is displayed successfully	Book with the specified accession number does not exist in the database.
	Scenario 15—View a book	456789	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Book not found	Book with the specified accession number does not exist in the database.
	Scenario 16—View a book alternative flow:	456789	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	User is allowed to exit and returns to the Main menu	—
	Book not found								n/a	n/a	User is allowed to exit and returns to the Main menu	—
	Scenario 17—View a book alternative flow:	*	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	User exits	—

*: do not care conditions (valid/invalid inputs).
n/a: option(s) that are not available for respective scenario.

Review Questions

1. What is object-oriented design (OOD)? List the steps to be followed during the OOD process.
2. How is the OOD different from coding a software?
3. Write short notes on the following:
 - (a) Lifeline in a sequence diagram
 - (b) Focus of control in a sequence diagram
4. Describe the various types of messages in UML with their notations.
5. Explain the elements of sequence diagrams with the help of an example.
6. Differentiate between sequence and collaboration diagrams.
7. (a) What is the purpose of centralized structure in a sequence diagram? Differentiate between centralized and decentralized structures.
(b) List the advantages of centralized structures in sequence diagrams.
8. Consider the railway reservation system. Draw the sequence diagram for reservation and cancellation of trains.
9. Define the following:
 - (a) Test case
 - (b) Test suite
 - (c) Test design and procedure
 - (d) Test results
10. List and explain the steps to derive the test cases from the use cases.
11. The ABC Bank wants you to write a software application that runs its ATMs. The bank has customers, and a customer has one or more accounts in the bank. Consider the example of withdrawing cash from an ATM machine. The process consists of the following steps:
 - (i) The customer is asked to insert card and enter PIN.
 - (ii) If the bank validates the PIN, then the withdrawal transaction will be performed:
 - (a) Customer selects amount.
 - (b) The system verifies that if it has sufficient money to satisfy the request, then the appropriate amount of cash is dispensed by the machine and a receipt is issued.
 - (c) If sufficient amount is not available in the account, a message “Balance not sufficient” is issued.
 - (iii) If the bank reports that the customer’s PIN is invalid, then the customer is requested to reenter the PIN.
- Draw the sequence diagram for cash withdrawal from the ATM machine.
12. Consider Question 11. Draw the collaboration diagram for cash withdrawal from the ATM machine.

13. Consider Question 11. Construct a detailed design for cash withdrawal from the ATM machine with operations along with their full signatures.
14. Consider Question 11 and perform the following:
 - (a) Construct the scenario diagram.
 - (b) Construct the scenario matrix, test case matrix and test matrix case with actual data.
15. Describe the guidelines given by Coad and Yourdon for creating a good design for object-oriented software.
16. The aim is to develop a Delhi Metro Automation System (DMAS). The DMAS has the following functions:
 - Token and smart card: To use the metro train, every passenger is required to have a valid token or a smart card.
 - ◆ The duration of a token or a smart card is 10 minutes from the time of entry; if an exit is performed after 120 minutes a fine of ₹ 50 is imposed.
 - ◆ A token can be used to exit from stations where the fare exactly matches the amount in the token; a smart card does not have this restriction.
 - ◆ Smart card carries a discount of 10% on all transactions.
 - ◆ To refill a smart card: maximum limit is ₹ 1000, minimum limit is ₹ 100 and denomination has to be in multiples of ₹ 50. A security deposit of ₹ 50 is to be paid during purchase of the smart card (refundable on return of the card).
 - ◆ To enter a station, the minimum balance on the smart card should be ₹ 10.
 - ◆ If the fare exceeds ₹ 8, the balance in the smart card can go negative upon exit.
 - Metro train: Every metro train runs between source and destination.
 - ◆ For each station there is a button. Station alert is shown by illumination of buttons. All buttons illuminate when the metro train begins from the source station. The button illuminates with blinking light on the next coming station. The blinking illumination is cancelled when the metro visits the desired station and then the button for next station blinks.
 - ◆ Closing of doors is mandatory for any movement of train.
 - ◆ Door opening time is 30 seconds in ideal conditions. In case of any obstruction, the system will attempt to clear the door three times. If obstruction persists, all the doors will be open on all coaches and again they are tried to close.
 - Reporting: The following reports are maintained during metro operation:
 - ◆ Entry and exit times along with stations for all tokens.
 - ◆ Fine collected along with reasons.
 - ◆ Daily collection of money.
 - ◆ Number of customers who travel per day.
 - Schedule of all trains are maintained.

Consider the DMAS, identify the use cases and write the use case descriptions.

17. Consider Question 16 and perform the following:
 - (a) Construct the scenario diagram
 - (b) Construct the scenario matrix, test case matrix and test matrix case with actual data.
18. Consider Question 16 and draw the sequence diagram of any two use cases.
19. Consider Question 16 and draw the collaboration diagram of any two use cases.
20. Consider the railway reservation system and draw sequence diagram of reservation and cancellation of tickets for this system. Make the necessary assumptions.
21. Consider a course scheduling system (CSS) for scheduling courses in Delhi Technological University. The purpose of CSS is to:
 - (i) enable entering data of courses, faculty members, the available facilities.
 - (ii) calculate and propose schedule for courses.
 - (iii) enable to manually update the proposed schedule, but keep track of the consistent schedule.
 - Entering programs and courses: An administrator should be able to enter new program and its running period. A data entry operator should be able to enter details of courses with their examiners. The examiner should be able to upload lecture, tutorials, projects, etc.
 - Entering resources: An administrator should be able to enter information about lecture rooms and laboratories in which classes will be taken place.
 - Scheduling: It provides schedule proposal, the days and time, and places where they can be scheduled. The scheduler allows some manual predefinition of the schedule. It shows if there are any conflicts.

Consider the CSS and construct the following:

- (a) Use case diagram
 - (b) Use case description of all the use cases
 - (c) Identification of classes and their relationships
22. Consider Question 21 and draw sequence diagram of any three use cases.
 23. Consider Question 21 and draw collaboration diagram of any two use cases.
 24. Consider the maintain course detail use cases of CSS case study given in Question 21 and create scenario matrix and test case matrix, and assign actual values to the test cases.
 25. Draw and explain the format of detailed design of classes.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. Interaction diagrams model:

(a) Static aspects of the system	(b) Dynamic aspects of the system
(c) Constant aspects of the system	(d) None of the above

13. In a collaboration diagram, a link can depict:

 - (a) Multiple messages
 - (b) Single message
 - (c) Abstract messages
 - (d) None of the above

14. In collaboration diagrams, the sequence number of a message is separated by the message name by:

 - (a) Dash
 - (b) Colon
 - (c) Arrow
 - (d) Dot

15. The aim of Coad and Yourdon guidelines is to:

 - (a) Improve software quality
 - (b) Produce clear and concise design
 - (c) Create simple and understandable classes
 - (d) All of the above

16. If you want to plan test activities such as developing test case, which analysis or design artifact is most useful?

 - (a) Use case
 - (b) Sequence diagram
 - (c) Class
 - (d) Object

17. How would you describe that some animals keep animals as pet?

 - (a) Use cases
 - (b) Sequence diagrams
 - (c) Classes
 - (d) Objects

18. Which of the following diagrams do **not** commonly illustrate a use case?

 - (a) Use case diagram
 - (b) Sequence diagram
 - (c) Collaboration diagram
 - (d) Statechart diagram

Further Reading

Robert and Kemerer compare structured design and object-oriented design methodologies in their research paper:

Robert, F. and Kemerer, C., Object oriented and conventional analysis and design methodologies: Comparison and critique. *IEEE Computer*, **25**(10): 22–39, 1992.

The following book offers design notations and methodologies given by Booch:

Booch, G., *Object-Oriented Design with Applications*. Redwood City, CA: Benjamin-Cummings, 1991.

A classic book on OOD was given by Coad and Yourdon:

Coad, P. and Yourdon, E., *Object-Oriented Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

The following book introduces UML modelling techniques for OOD along with case studies:

James, R., et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

An excellent OOD of an ATM case study is available on the following wed page:

<http://www.math-CS.gordon.edu/courses/cs211/ATMExample/>

In the following book chapter, the application of OOD methodology for web applications development is proposed:

Shah, A., OODM: An object-oriented design methodology for development of web applications.
In: *Information Modeling for Internet Applications*. USA: IGI Publishing, 2003.

The following research paper authored by Eden provides formal specification of OOD:

Eden, A., Theory of object design. *Information Systems Frontiers*, 4(4): 379–391, 2002.

7

Moving Towards Implementation

Modelling dynamic behaviour of a system allows to represent, visualize, depict and document the changing parts of the system. In the previous chapter, we have studied interaction diagrams (sequence and collaboration). In the interaction diagrams, the interaction among a set of objects to accomplish a given use case scenario is depicted. The behaviour of the operations and state of the objects may be modelled using activity diagrams and statechart diagrams before starting the implementation. This intermediate step will help us to bridge the gap between design and implementation. The behaviour of the operation may become transparent for the purpose of implementation. The states of the objects are depicted along with inputs and outputs produced based on these inputs. This will enhance the understanding of the system and may provide a strong foundation for implementation particularly in the new areas of implementation and technology.

In this chapter, we will describe two UML diagrams that model dynamic parts of the system, namely, activity diagrams and statechart diagrams. The activity diagram models the sequence of steps followed by a process or operation. In the statechart diagram, the various states through which an object goes through during its life cycle are depicted. Implementation transforms the detailed design into a source code. This task is achieved by a group or team of developers. In this chapter, the issues related to implementation are also addressed.

7.1 Activity Diagrams

The dynamic aspects of a system are modelled through activity diagrams. Activity diagrams are used to model the working of a process or an operation. The activities carried out in a process/workflow or an operation are depicted in an activity diagram. Activity diagrams enable to visualize, understand and document the flow of activities in an operation or a process. The notations used for various symbols used in an activity diagram are given in Table 7.1.

Table 7.1 Symbols used in activity diagrams

S. No.	Construct	Notation	Description
1	Activity		An activity is an executing set of steps that take place in an activity diagram.
2	Transition		Transition represents a path from one activity to another activity.
3	Start		Start state represents beginning of an activity diagram.
4	Stop		Stop state represents end of an activity diagram.
5	Branching		Branching is used to represent an if-else condition with a guard condition in square brackets.
6	Fork		Fork breaks up an activity into a set of concurrent subactivities.
7	Join		Join is used to combine the flow of concurrent activities into the next single activity.

Activity diagrams have many applications and they are used to:

1. increase the understanding of a business model, a process or a use case.
2. determine the objective for business-related e-commerce applications.
3. simplify a given operation.

In sequence diagrams, the interaction amongst objects is represented, whereas activity diagrams depict the steps involved in a process or operation. An activity diagram represents executable computational steps in the system. They explore the order in which the activities must be carried out to achieve a goal.

7.1.1 Activities and Transitions

An activity represents execution of a step or a set of steps in a process or an operation. The activity may be atomic or may consist of multiple steps. An activity either manual or automated

is used to complete a given task. Transition represents the path from one activity to another activity. An activity is represented by an oval-shaped circle with rounded ends and a transition is represented by a directed line. The activity diagram of a simple operation to subtract two numbers is shown in Figure 7.1.

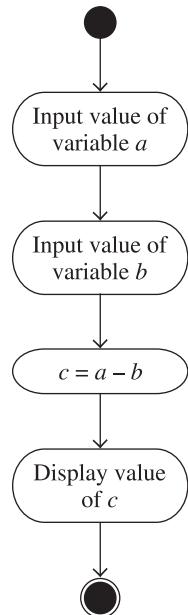


Figure 7.1 Activities and transitions for subtracting two numbers.

7.1.2 Branching

Branching is used for making decisions in an activity diagram. A branch consists of a guard condition that is used to control which activity should be followed from a set of alternative activities once the current activity is completed. Similar to a flow chart, a boolean expression (guard condition) depicted in brackets is used to take decision about which alternative path to follow.

The decision branch is represented by a diamond with one incoming transition and more than one outgoing transition. On each outgoing transition, a guard condition is specified. The activity corresponding to the outgoing transition that meets the guard condition will be executed. For example, if we want to obtain only positive difference of two numbers, this can be achieved by using decision branch with guard conditions as shown in Figure 7.2.

7.1.3 Modelling Concurrency

When modelling a process, we may encounter concurrent flows. In UML, concurrent flows are depicted by synchronization bars to represent fork and join of parallel activities. A synchronization bar is represented by a thick horizontal line. A fork divides the activities into two or more concurrent subactivities. The fork may consist of one incoming transition and more than one outgoing transition. A join combines the two or more concurrent flows when they are

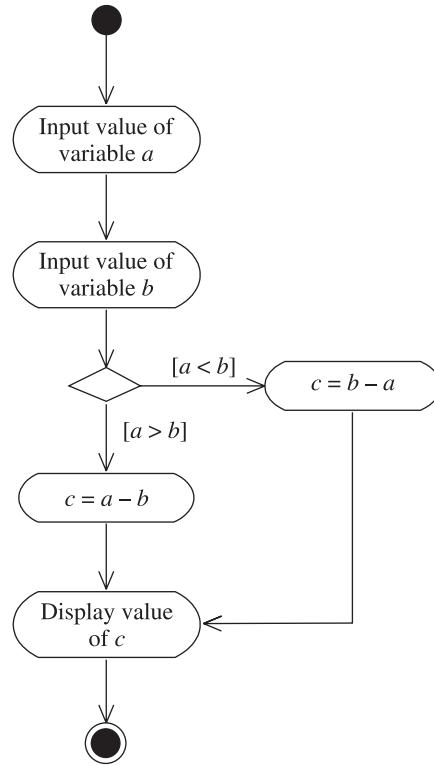


Figure 7.2 Branching.

completed. The join is used for synchronization of concurrent activities. Each concurrent activity waits for another activity to complete and finally all are joined at a single point and all activities after this point continue to execute. A join may have more than one incoming transition and one outgoing transition.

In Figure 7.3, a fork represents split of flow into three concurrent flows in which addition, subtraction and multiplication of two numbers are carried out in parallel. After completion of these flows, each flow joins back into a single flow and the output is displayed. Each fork should have a join.

7.1.4 Using Swimlanes

A swimlane groups all the activities that are carried out by the owner of the swimlane. Swimlanes are used to determine which business entity is responsible for carrying out a specific activity. Each swimlane is represented by a unique name. The transitions may cross between entities in the swimlanes. Swimlanes are represented by vertical lines that divide each group from its neighbours. For example, a system may have the following three swimlanes as shown in Figure 7.4:

1. *Manager:* The activities residing in this swimlane show the activities for which the manager is responsible. As shown in Figure 7.4, the manager enters data.

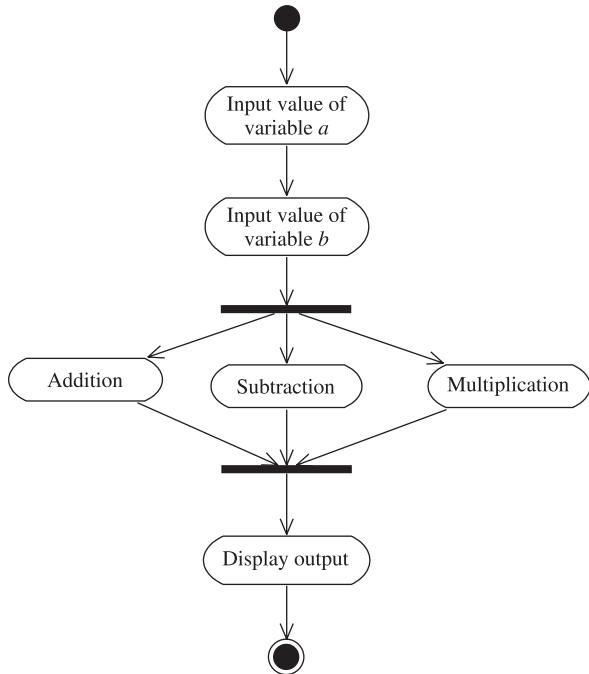


Figure 7.3 Modelling concurrent activities.

2. *System*: The activities residing in this swimlane show the activities for which the system is responsible. The system generates the requested report.
3. *Printer*: The activities residing in this swimlane show the activities for which the printer is responsible. The printer prints the report generated by the system.

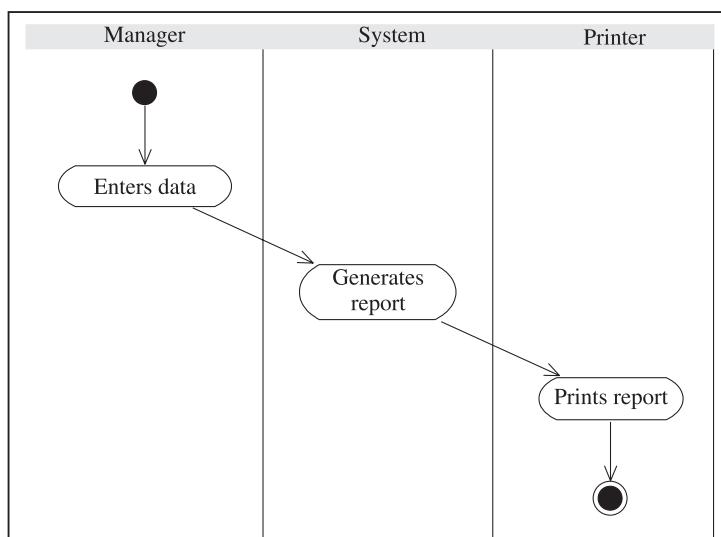


Figure 7.4 Swimlanes.

In an activity diagram, objects may be involved to represent the changes in the object because of an activity. For example, in Figure 7.5, the issue book activity changes the state of Book object to ‘closed for issue’. As shown in Figure 7.5, the objects can be specified in the activity diagram. These objects can be connected to an activity through dotted directed lines.

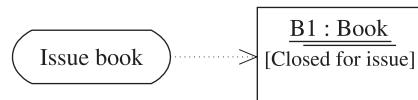


Figure 7.5 Object flow.

7.1.5 Uses of Activity Diagrams

Activity diagrams can be used to model a system as a whole, a process, a use case, or an operation. It can be used in the following ways:

1. **For modelling a workflow or process:** There are many business processes that consist of activities that need to be followed throughout the process. The following steps may be carried out to model a workflow or a process:
 - (i) Determine the business entities that will participate in the activity diagram.
 - (ii) Identify the initial state with its preconditions and final state with its postconditions.
 - (iii) Specify the activities that will take place.
 - (iv) Provide a separate activity diagram for each complete activity.
 - (v) Identify decisions and concurrent flows.
 - (vi) Show the changing values of objects in the activity diagram.

For example, consider an e-commerce application for purchasing an item from the Internet. The following steps are followed:

- (i) The customer selects the items to be purchased.
- (ii) The customer adds the items in the shopping cart.
- (iii) The customer places an order.
- (iv) The vendor receives the order.
- (v) A secure payment gateway asks for the customer’s credit card details.
- (vi) The customer provides his/her details to the payment gateway.
- (vii) The credit card company verifies the customer’s details and approves the transaction.
- (viii) The vendor confirms the order.
- (ix) The vendor communicates with warehouse that ships the order to the customer at the specified shipping address.
- (x) The credit card company sends the bill to the customer for payment.

Figure 7.6 shows the activity diagram of the above business process. In this diagram, there are five swimlanes. The get item and ship item activities reside in the warehouse swimlane indicating that it is the responsibility of the warehouse to get the correct item and then ship it to the specified shipping address of the customer. The workflow ends when the customer receives the bill from the credit card company.

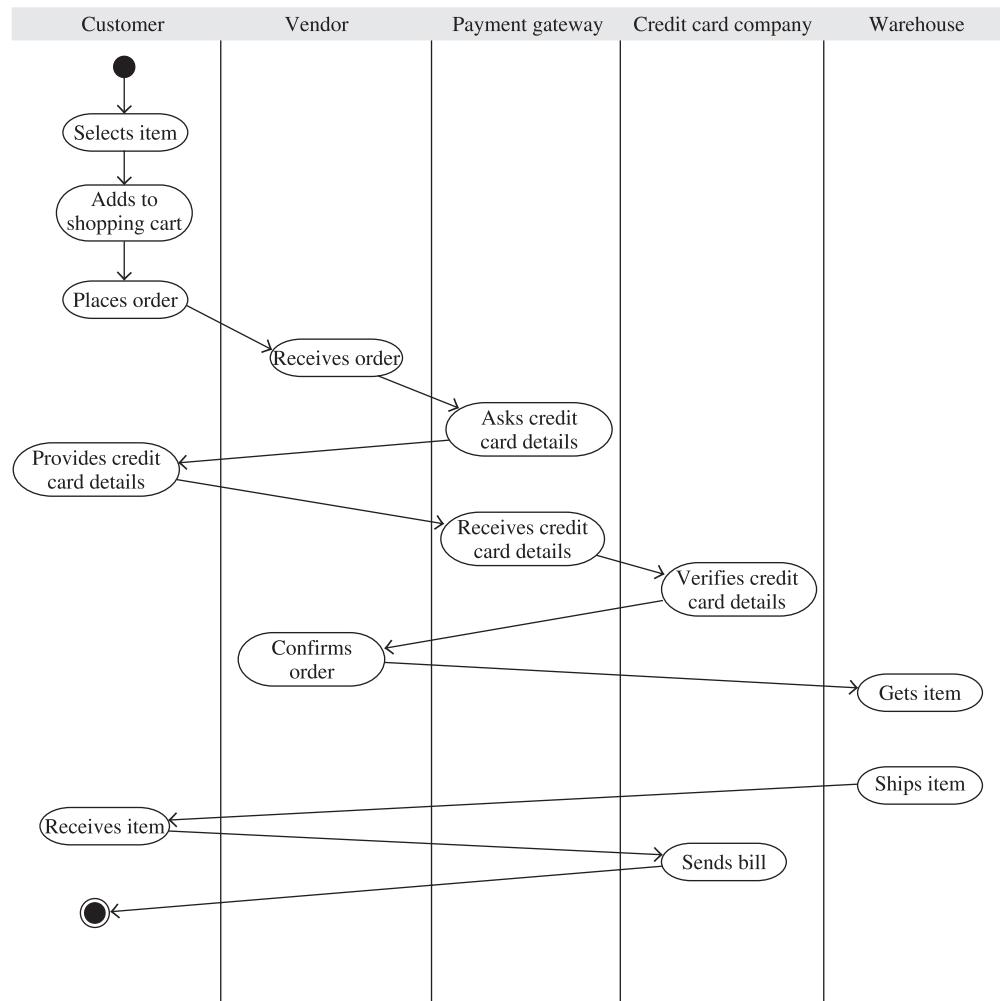


Figure 7.6 Activity diagram of e-commerce application.

2. **To model a use case:** Each use case represents functionality of the system. We can depict the flow of the use cases by using the activity diagram. In order to create an activity diagram of a use case, the following steps should be followed:

- Identify the preconditions and postconditions of the use case as given in the use case diagram.
- Identify the beginning of the use case.
- Identify the activity that will take place in the use case.
- Keep the activities simple and understandable.

For example, consider the ‘maintain book details’ use case of the LMS. The activity diagram for this use case is shown in Figure 7.7. The activity diagrams depict the steps followed in the use case. The appropriate activity is selected based on the desired flow (add book, delete book, update book, view book).

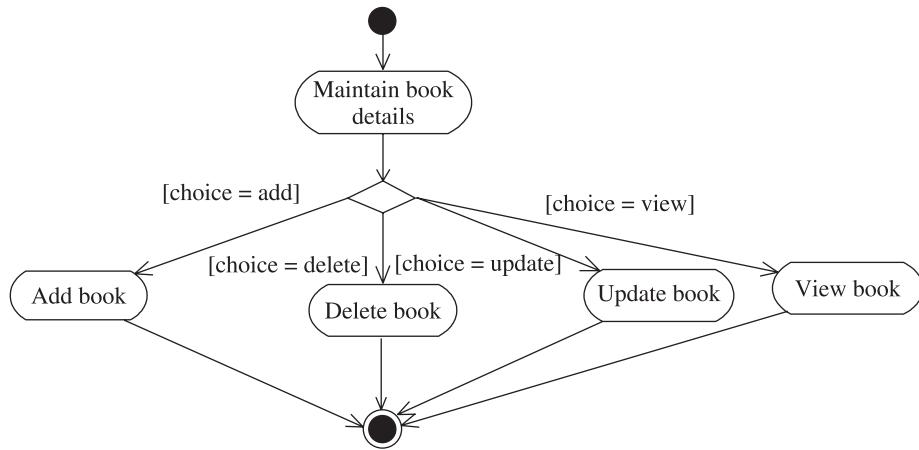


Figure 7.7 Activity diagram of 'maintain book details' use case.

Further, the detailed activity diagrams of add book, delete book, update book and view book activities may be created.

3. **To model an operation:** An activity diagram can be created for an operation. When the activity diagram is used to model an operation, it simply depicts the flow of activities in an operation as in the case of a flow chart. The following steps should be followed to model an operation:
 - (i) Identify parameters and return the value of an operation.
 - (ii) Identify preconditions and postconditions of an operation.
 - (iii) Determine the detailed set of activities that may take place in an operation.
 - (iv) If necessary, use branching to represent conditions.

The algorithm of 'issueBook' operation of Transaction class is shown in the activity diagram given in Figure 7.8. First, the current date is retrieved. Then, there is a guard condition that

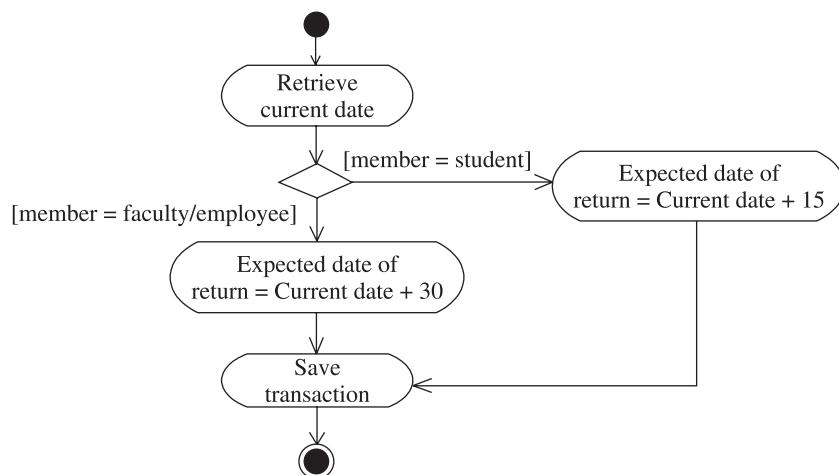


Figure 7.8 Activity diagram of 'issueBook' operation.

checks whether the member is a student or faculty/employee. If the member is a student, then the expected date of return is incremented by 15 days with respect to the current date. Otherwise, the operation adds 30 days in the current date. Finally, the transaction is saved to the database.

Activity diagrams can be used to model the behaviour of a complex operation that is difficult to understand. The activity diagram will increase the understanding of the algorithm of the operation.

EXAMPLE 7.1 Consider an operation to calculate the area of a triangle. Construct the activity diagram for this operation.

Solution The activity diagram for computing the area of a triangle is shown in Figure 7.9.

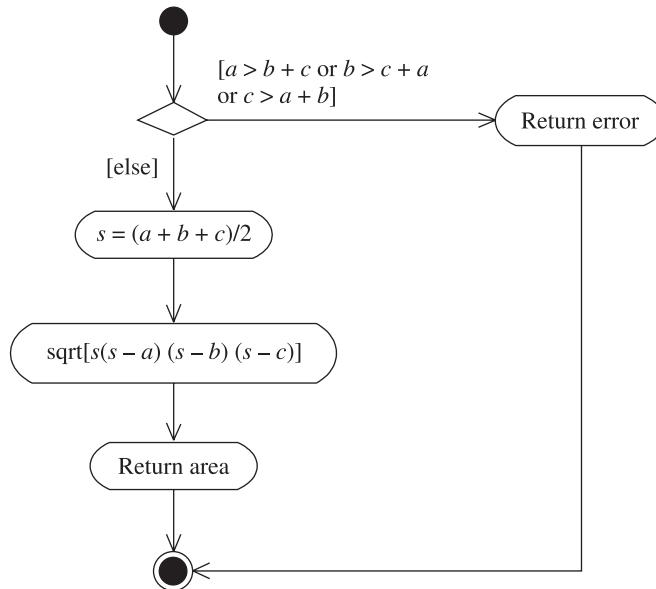


Figure 7.9 Activity diagram of 'computeArea' operation.

EXAMPLE 7.2 Consider the calculateFine operation of Transaction class in the LMS. Construct the activity diagram for this operation.

Solution The activity diagram for calculating fine is shown in Figure 7.10.

7.2 Statechart Diagrams

Like activity diagrams, statechart diagrams also model the dynamic aspects of the system. These diagrams are used to model the life cycle of the object, from the time it is created until the object is destroyed. The difference between activity diagrams and statechart diagrams is that statechart diagrams are based on states and activity diagrams are based on activities. A statechart diagram models the states of an object's lifetime, whereas an activity diagram is used to model the

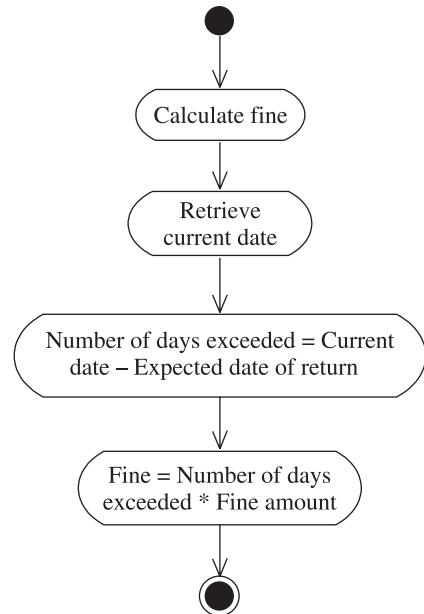


Figure 7.10 Activity diagram of ‘calculateFine’ operation.

sequence of activities in a process or an operation.

Statechart diagrams are created to model the classes which have significant dynamic behaviour. A class is said to have a significant dynamic behaviour if it goes through multiple states. The transition causes an object to move from one state to another state. For example, consider a faculty member in the university. If the person is in the relationship with the university, then the status of the person is employed. Otherwise the person may be expelled or retired from the university. These models will help the developers to obtain a clear understanding of the dynamic behaviour of the object. These diagrams are very useful for modelling reactive systems.

7.2.1 States and State Transition

A state is one of the conditions that an object may satisfy during its lifetime. In the given state, the object may perform some activity or wait for some event to happen. The state of the object may be determined by one of the possible values of the attributes of the class. For example, in the LMS, an object of Book class may be open for issue (able to issue the book to a member) or close for issue (the book cannot be issued). The state depends upon the actual return date of the book. Most of the notations used in statechart diagrams are common with those used in activity diagrams. The UML notation for a state is shown in Figure 7.11. A state is represented by a rectangle with rounded ends.



Figure 7.11 State notation.

A state can also be identified by looking at the links between objects. A faculty member may be teaching or on leave. This depends upon the link between faculty and course classes and multiplicity specified between the objects of these classes. The sequence diagram for each use case may be examined to reveal the object's state. For example, in the LMS, the book is initialized (added), open for issue (members can get the book issued), close for issue (book is already issued) and expired (not in physical condition to be issued).

State transitions represent the change of an object from one state to another state. Like in activity diagrams, in state chart diagram also the state transition is represented by a directed line. In the UML, the state may also consist of entry action, do activity and exit action. Entry action takes place when the object enters a given state. The do activity specifies the tasks/activities that must be performed while in the current state and continues until the state is exited. The entry and exit actions are preceded by a word entry/exit followed by a '/' (slash) as shown in Figure 7.12.

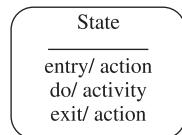


Figure 7.12 State with entry and exit actions.

Every statechart diagram consists of two special states—start state and stop state. The start state specifies the creation of the object and the stop state signifies the destruction of the object. The notations for these states are the same as those of start and stop states in an activity diagram as shown in Table 7.1.

7.2.2 Event, Action and Guard Condition

A state transition may be associated with an event, action or guard condition. The notations used to depict these three associations with state transition are shown in Figure 7.13.

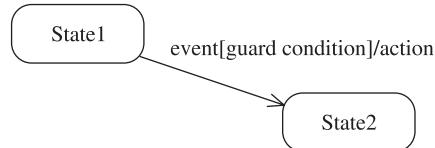
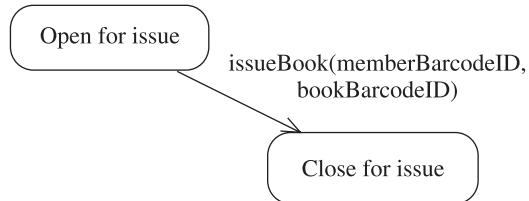
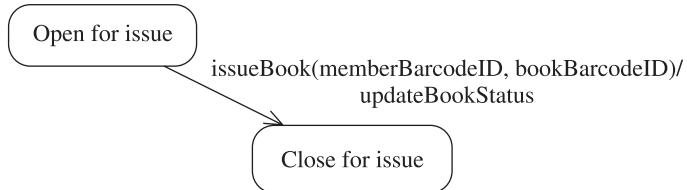


Figure 7.13 State transition details.

An event is a message that is sent from one object to another object. In the statechart diagram, an event can be shown using the operation name on the state transition. The operation may consist of arguments. The operation may be a message from some other object. For example, Library Staff requests issue of book; the event moves the book from open for issue state to close for issue state as shown in Figure 7.14. In the LMS example, the 'issueBook' operation is associated with the state transition.

**Figure 7.14 Event for issue of book.**

An action is a behaviour that occurs in response to a state transition. The actions become the operation of the object. All actions are noninterruptible and are shown along with state transition after the event name is separated by ‘/’ (slash). For example, when transitioning from the open for issue state to the close for issue state, the action updateIssueStatus occurs as shown in Figure 7.15.

**Figure 7.15 Action for issue of book.**

A guard condition is a Boolean expression that tests whether the condition is true or not. If the condition is true, then the state transition is done, otherwise not. For example, the book will only be issued to a member if the number of already issued books does not cross the maximum permissible limit of the books that can be issued.

7.2.3 Modelling Life Cycle of an Object

The primary use of the statechart diagram is to model the life cycle of an object. The interaction diagrams model the interaction between objects working together, whereas the statechart diagram model states through which a single object goes through during its life cycle. The activity diagrams are used to model the sequence of activities of a process, whereas statechart diagrams are used to model states of an object. The dynamic aspects of a class use case or a system can be modelled by identifying their dynamic or reactive objects. Hence, the most common use of the statechart diagrams is to model the dynamic objects.

When a dynamic object is modelled, essentially three things are specified about it: the states through which the object goes, the events that are triggered for transition from one state to another state, and the actions that are performed during the transition of the state. This process involves the life cycle of the object from the time it is created to the time it is destroyed (Figure 7.16).

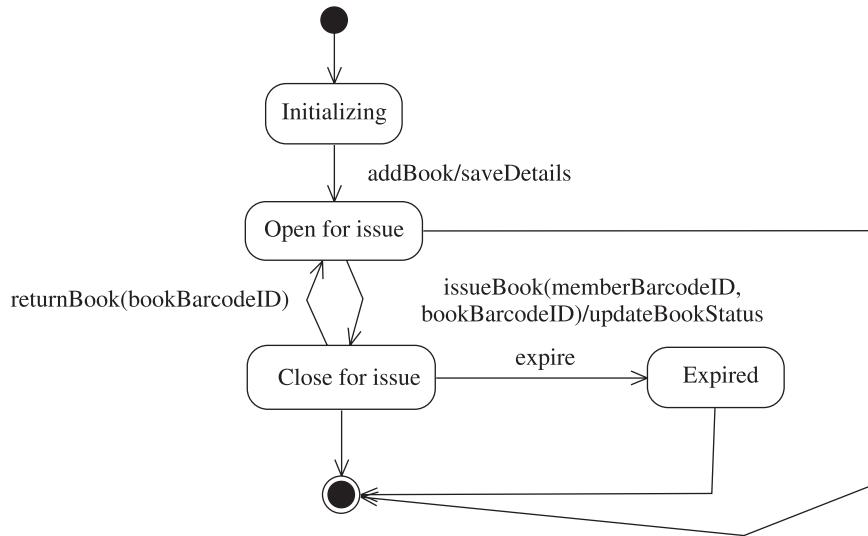


Figure 7.16 Life cycle of object of a Book class.

7.2.4 Creating Substates

Substates are nested inside other states. The substates are also referred to as nested states and the outer state is referred to as a superstate. A simple state has no substate, whereas a superstate is composed of two or more nested states. For example, a printer in its startup state might be both heating and initializing. Thus, starting up is a superstate and heating and initializing may be two substates. The superstate reduces the complexity of the state transition diagram. The substates are simply shown inside the superstates and may be nested up to any level. For example, consider an ATM system. It consists of three basic states: idle (when no customer transactions are being carried out), processing (when a customer transaction is being executed) and switched off (when the ATM is either out of service or switched off due to maintenance purposes). While the state ATM system is processing, the following steps will be followed:

1. Reading information from the card
2. Validation of customer card and pin
3. Customer selects the transaction
4. Processing of transaction selected by the customer
5. Printing of receipt of the desired transaction
6. If the customer wants to continue with another transaction, then the flow returns to step 3
7. Ejecting the customer card

These steps can be represented as: Reading card, validating, selecting transaction, performing transaction, printing receipt and ejecting card. The customer may abort or cancel the transaction at any point in time during processing state and move to the idle state. Thus, rather than putting transitions from all the states involved in processing superstate to the idle state, it is better to nest

all the states in processing superstate as shown in Figure 7.17. The processing superstate consists of six substates: Reading card, validating, selecting transaction, performing transaction, printing receipt and ejecting card. As soon as the card is inserted by the customer, the ATM system changes from the idle state to processing state and the control passes to reading card state of the substructure. After reading the card, the customer details are validated, and the customer selects the desired transaction which is processed by the bank followed by printing of receipt. Finally the card is ejected and the ATM system moves back to the idle state.

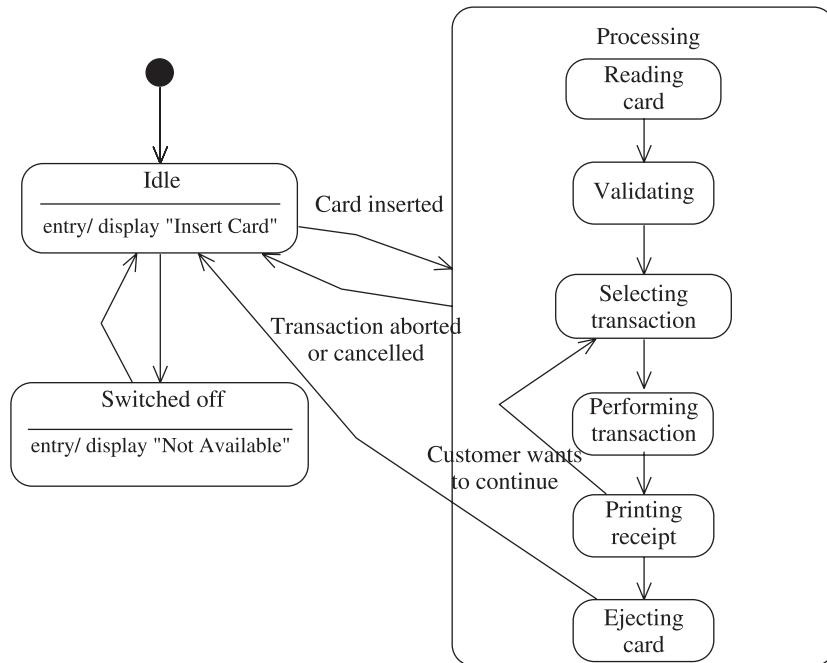


Figure 7.17 State transition diagram of ATM system with substates.

If the substate structure is not followed, transition from every state to idle state (invoked by abort or cancel operation) will be required.

The history state can be specified in a superstate. This state enables the control to remember the point in last active state, if due to any reason the superstate is to be left. There may be times when before leaving the superstate we want to remember the last active substate of an object. For example, consider a process. It consists of four basic states—ready, running, waiting and terminated. The running superstate has three substates—initializing, executing and exiting. Now, if a process is interrupted in between the execution, the control must remember the last active state of the process. When the action of waiting state is performed, the control returns to the history state of the running superstate. This time the control is passed directly to the executing state instead of the initializing state as the executing state was the last active substate prior to the transition from running to waiting. The state transition diagram of the process is shown in Figure 7.18. The history state shows the last state to remember before leaving the superstate. As shown in Figure 7.18, the history state is depicted by an H inside the small circle.

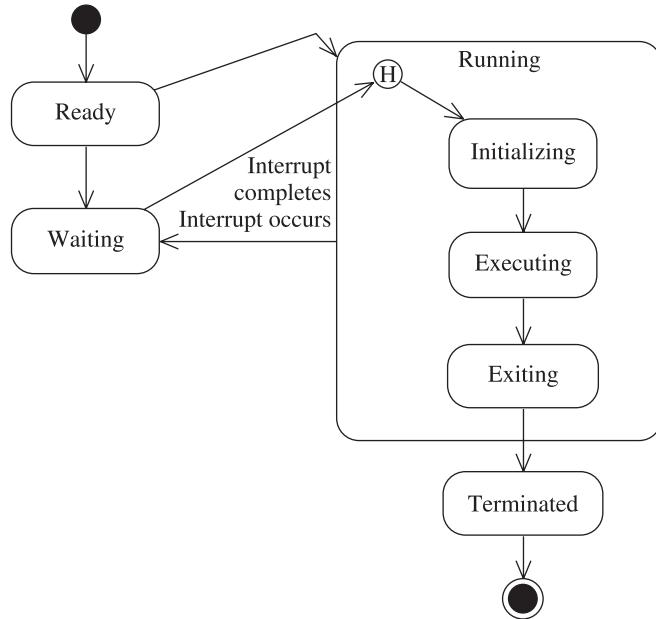


Figure 7.18 State transition diagram of a process with history state.

The advantages of the state transition diagram can be seen in modelling the objects where states are important. These diagrams are also beneficial in modelling complicated sequence of events.

EXAMPLE 7.3 Consider the University Registration System. The students are registered in the beginning of each semester. The registration must be completed by the students within 15 days of the start of the registration process. Draw the statechart diagram showing the states of registration.

Solution An object of registration class may be open for registration (able to register a student) or close for registration (maximum number of days for registration has finished). The state depends upon the number of days registration is open. The statechart diagram is shown in Figure 7.19.

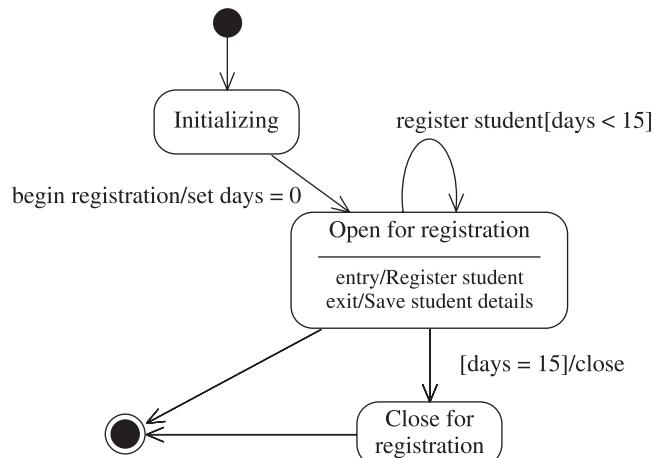


Figure 7.19 Statechart diagram for registration class.

EXAMPLE 7.4 Consider an e-commerce application for purchasing an item from the Internet. The state of order class needs to be checked in this application. The orders are processed after being initialized. The order may get cancelled by the customer. Draw the statechart diagram showing the states of order class.

Solution The statechart diagram for order class is shown in Figure 7.20.

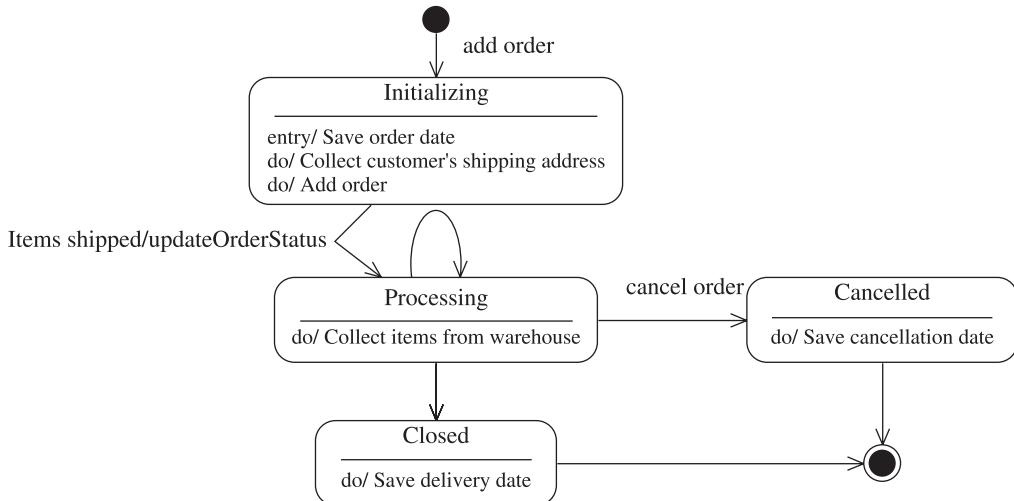


Figure 7.20 Statechart diagram for order class.

7.3 Storing Persistent Data in Database

The persistent objects survive even after the program is no longer in execution. Thus, there is a need to store these objects permanently in the secondary memory. The database management system (DBMS) is often required for handling persistent objects. A DBMS is a software system that can manage large amount of data in an efficient and robust manner. The DBMS offers the following features:

1. *Efficient storage of data:* This feature is very helpful especially when the data is stored in an external storage device.
2. *Data independence:* Application programs are kept as much independent as possible from the internal details of management and storage of data.
3. *Integrity and security:* In the DBMS, verification checks can be enforced before the insertion of data. The portion of the database that must be visible to the given users can also be controlled.
4. *Concurrency control and recovery:* The DBMS allows multiple users to work simultaneously and also provides the facility of recovering the database into a consistent state after the failure.
5. *Easy query:* The DBMS allows the user to access the data in the required manner easily.

The DBMS is used in most of the applications and usually it is specified in the documents produced during the requirement analysis phase of software development. For example, as given in Chapter 3, in the SRS document it is specified that the LMS will use SQL Server 2000 for storing and managing data.

The persistent objects can be retrieved from the OOA and OOD phases. The entity classes are used to store the information that survives for longer time in the system. Thus, the attributes included in these classes will be required to be stored in the database. The relational model of DBMS is being widely used in the industries these days.

7.3.1 Mapping Entity Classes to Database Tables

In the relational model, the data is stored in the form of a table. Such an entity class must be represented in the form of a table in the database. The following steps must be followed when converting the entity classes into tables:

1. Each entity class marked as persistent should be converted into a table.
2. The attributes in each entity class should be represented as columns in the table. If an attribute can be decomposed into further attributes, then it should be either converted into a separate table or represented using multiple columns. For example, address can be decomposed into street address, state, city, PIN, country, etc.
3. The data type of each attribute in the entity class must be mapped into the corresponding data type of the column in structured query language (SQL).
4. The unique identifier in the entity class will be represented as primary key in the table. For example, in member persistent object, the memberBarcodeID is a unique identifier which must be specified as primary key while converting into table.
5. Other integrity constraints including referential integrity constraints (also known as foreign key), if any, must be specified for each attribute in the table. For example, the member name can never be null.
6. The values of attributes for each object will be represented as rows in the table.

As shown in Table 7.2, the login class consists of three columns with their column names. The attributes are very simple, hence will not be converted into tables.

Table 7.2 Login table

LoginID	Password	Role

The table definition of the login table in SQL is given in Figure 7.21.

```
CREATE TABLE T_Login (
    loginID VARCHAR ( 255 ) NOT NULL,
    password VARCHAR ( 255 ) NOT NULL,
    role VARCHAR ( 255 ) NOT NULL,
    CONSTRAINT PK_T_Login13 PRIMARY KEY (loginID)
);
```

Figure 7.21 Table definition of login table.

This definition states that loginID is a primary key. The definition of the login table also illustrates that all the columns should be ‘NOT NULL’, which means that some value must be specified for each loginID, password and role when inserting a new row. If any of the specified constraint is violated, then the error will be returned by the DBMS.

7.3.2 Representing Inheritance in Tables

If the entity classes (those which are persistent) are having generalization relationships amongst themselves, then the following ways can be used while converting them into tables:

1. The base class and derived classes are converted into tables. The derived class consists of reference to the primary key of the base table. The primary key of the tables of the derived class(es) will be the same as in the case of a table representing the base class.
2. The inherited attributes (the ones in the base class) are copied to all the tables that are created to represent the derived class(es).

For example, consider the LMS. The member entity class consists of three derived classes, namely, student, faculty and employee.

The first alternative states that a separate table is created for each base class and derived classes with common primary key as shown in Tables 7.3 to 7.6.

Table 7.3 Table representing Member entity class

memberID	photograph	name	fname	DOB	phone	email	memberdate	validUpto	noIssuedbooks

Table 7.4 Table representing Student entity class

memberID	rollNo	programme	school

Table 7.5 Table representing Faculty entity class

memberID	empID	designation	school

Table 7.6 Table representing Employee entity class

memberID	empID	designation	branch

The second alternative is depicted in Tables 7.7 to 7.9.

Table 7.7 Alternative way of representing Student entity class

member ID	photo-graph	name	fname	DOB	phone	email	noIssued books	roll no.	programme	school

Table 7.8 Alternative way of representing Faculty entity class

member ID	photo-graph	name	fname	DOB	phone	email	member date	valid Upto	noIssued books	empID	designation	school

Table 7.9 Alternative way of representing Employee entity class

member ID	photo-graph	name	fname	DOB	phone	email	member date	valid Upto	noIssued books	empID	designation	branch

The first alternative does not involve duplications of columns inherited. However, as the second alternative requires less joins, it is faster than the first one. If the attributes of the base class change, then these changes will affect all the derived classes. The first alternative is more preferable as compared to the second alternative. The table definition of the tables constructed following the first alternative is shown in Figure 7.22.

```

CREATE TABLE T_Member (
    memberID INTEGER NOT NULL,
    photograph SMALLINT NOT NULL,
    name VARCHAR ( 255 ) NOT NULL,
    fname VARCHAR ( 255 ) NOT NULL,
    DOB DATE NOT NULL,
    phone INTEGER NOT NULL,
    email VARCHAR ( 255 ) NOT NULL,
    memberdate DATE NOT NULL,
    validUpto DATE NOT NULL,
    noIssuedbooks INTEGER NOT NULL,
    CONSTRAINT PK_T_Member14 PRIMARY KEY (memberID)
);

```

Figure 7.22 (Contd.)

```

CREATE TABLE T_Student (
    rollNo INTEGER NOT NULL,
    programme VARCHAR ( 255 ) NOT NULL,
    school VARCHAR ( 255 ) NOT NULL,
    memberID INTEGER NOT NULL,
    CONSTRAINT PK_T_Student15 PRIMARY KEY (memberID)
);

CREATE TABLE T_Employee (
    empID INTEGER NOT NULL,
    designation VARCHAR ( 255 ) NOT NULL,
    branch VARCHAR ( 255 ) NOT NULL
);

CREATE TABLE T_Faculty (
    empID INTEGER NOT NULL,
    designation VARCHAR ( 255 ) NOT NULL,
    school VARCHAR ( 255 ) NOT NULL
);

```

Figure 7.22 Entity classes converted into tables.

EXAMPLE 7.5 Consider transaction, reserve, fine status and book entity classes and create tables corresponding to these classes.

Solution The tables for transaction, reserve, fine status and book entity classes are given in Figure 7.23.

```

CREATE TABLE T_Transaction (
    bookBarcodeID INTEGER NOT NULL,
    memberBarcodeID INTEGER NOT NULL,
    dateOfIssue DATE NOT NULL,
    expectedDOR DATE NOT NULL,
    actualDOR DATE NOT NULL,
    T_Transaction_ID INTEGER NOT NULL,
    memberID INTEGER NOT NULL,
    CONSTRAINT PK_T_Transaction1 PRIMARY KEY (T_Transaction_ID)
);

CREATE TABLE T_Reserve (
    date DATE NOT NULL,
    time VARCHAR ( 255 ) NOT NULL,
    bookBarcodeID INTEGER NOT NULL,
    memberBarcodeID INTEGER NOT NULL,
    T_Reserve_ID INTEGER NOT NULL,
    CONSTRAINT PK_T_Reserve9 PRIMARY KEY (T_Reserve_ID)
);

CREATE TABLE T_FineStatus (
    bookBarcodeID INTEGER NOT NULL,
    memberBarcodeID INTEGER NOT NULL,

```

Figure 7.23 (Contd.)

```
    fine DOUBLE PRECISION NOT NULL,
    paid SMALLINT NOT NULL,
    T_FineStatus_ID INTEGER NOT NULL,
    T_Transaction_ID INTEGER,
    CONSTRAINT PK_T_FineStatus5 PRIMARY KEY (T_FineStatus_ID)
);

CREATE TABLE T_Book (
    bookBarcodeID INTEGER NOT NULL,
    subjectDescriptor VARCHAR ( 255 ) NOT NULL,
    ISBN INTEGER NOT NULL,
    bookTitle VARCHAR ( 255 ) NOT NULL,
    language VARCHAR ( 255 ) NOT NULL,
    authorName VARCHAR ( 255 ) NOT NULL,
    publisher VARCHAR ( 255 ) NOT NULL,
    issueStatus SMALLINT NOT NULL,
    CONSTRAINT PK_T_Book2 PRIMARY KEY (bookBarcodeID)
);
```

Figure 7.23 Tables for transaction, reserve, fine status and book entity classes.

7.4 Implementing the Classes

We have described the activity diagrams and statechart diagrams, and mapped the entity classes into tables. The implementation part can begin using the specified programming language. We should be able to trace the source code back to the OOD and OOA. The traceability helps in managing changes during the software development life cycle. The developed source code should follow the good programming practices and the coding standards specific to the programming languages being used. The classes are directly mapped into classes in object-oriented languages. The attributes identified in the OOA phase are implemented in classes with their variable type specified in the selected object-oriented programming language. Similarly the operations defined in the OOD phase are added to their desired classes with appropriate parameter types and return type. The inheritance is directly mapped onto inheritance between classes which is specific to the programming language. For example, the class IssueBookInterface consists of the following attributes and operations in visual basic:

```
Private bookBarcodeID As Long
Private memberBarcodeID As Long

Public Sub acceptBookBarcodeID(bookBarcodeID As Long)
End Sub

Public Sub acceptMemberBarcodeID(memberBarcodeID As Long)
End Sub
```

In the case of non-object-oriented languages such as C, the object-oriented concepts such as inheritance and polymorphism need to be translated.

7.4.1 Good Programming Practices

There are many general specifications for coding a program which are followed irrespective of any programming language used. Some of the specifications which are language independent are listed below:

1. ***Use of meaningful variable names:*** The variables should convey their meaning and purpose. The programmer developing the source code is not the only person who works on the class but there are many other programmers who will work on the classes and source code in the later phases of software development. Hence, the use of meaningful variables will help the programmers to understand the source code in the maintenance phases or when the person who actually developed the source code is unavailable. In addition, the variables should be consistent. For example, in the LMS, the identification name of the member should not be written differently at different places such as memberID, memberBarcodeID, MID, etc. The names of the variable that fulfil the same purpose should be the same.
2. ***Use of documentation:*** Documentation increases the readability and understandability of the source code. Documentation will help the programmers to easily understand the source code, which will benefit in the debugging process. The description and purpose of the class and each of its operations must be specified in the beginning of each class. Comment lines are used in the programming languages for adding documentation in the source code. The information that is not easily available in the source code must also be provided in the comments.
3. ***Use global data rarely:*** The programmers should avoid the use of global data. Global data is prone to accidental modifications. The change in global data at one place will affect all the places where the same data is being used.
4. ***Use consistent formatting:*** The programming should include tabs, spaces and indentation, wherever required. This will increase the readability of the program. Indentation shows that a particular line belongs to the same block represented by braces {....}.
5. ***Use of error messages:*** The appropriate error messages should be displayed when something unexpected is expected to happen. These messages will provide the reasons for the unexpected event before the program is terminated.
6. ***Use of if statement:*** The complicated if statements must be avoided. The nesting of if statement must be kept to a minimum level.

7.4.2 Coding Standards

The coding standards are specific to the programming language. The coding standards specify the restrictions on the use of features of the programming language. Today the coding standards are embedded in the programming tools itself. Thus these standards can be verified by the tool and if there is any deviation from the specified standard, then it is reported. The following coding standards are provided and followed in Java language (Aggarwal et al., 2009):

1. Developers write code to catch all the possible Java exceptions (generated at run-time to describe a problem encountered during the execution of a program) explicitly that could occur.
2. They made use of reusable components including the Java standard library classes provided to them.
3. The files are divided into different packages (group of classes and interfaces). Hence, a well-organized directory structure is maintained. It makes accessibility of classes easier to the testing teams.
4. Documentation (including implementation comments and documentation comments) of each block of code is added to the code that makes it more readable and understandable.

7.4.3 Refactoring

The source code **cannot be written perfectly for the first time**. The aim of the developer should be to construct the correct and **high-quality** source code. The source code should be **refined until it satisfies the non-functional attributes** of the software. Refactoring improves the internal structure of the program without changing the functionality of the software. Refactoring increases the readability and decreases the complexity of the source code. Thus, this improves the maintainability of the source code. The “code smells” are removed by the use of refactoring. The functionality of the operation remains the same, but the non-functional problems with the source code are reduced. The advantages of refactoring include:

1. Improved **understandability and readability**
2. Enhanced **maintainability**
3. Ease of **enhancement**
4. Ease of **adding new features**

For example, in order to improve the quality of the source code, the programmer may move the common functionality to the parent class, break the source code into more logical units (this includes operation breaking or class breaking), or improve names of attributes, etc.

7.4.4 Reusability

The components of a software should be easily reusable in some other software. Thus, the aim of implementation should be to build high-quality and maintainable components that can be easily reused. These components should be well documented and tested.

Review Questions

1. Differentiate between interaction diagrams and statechart diagrams.
2. What is an activity diagram? What are the basic symbols used in the construction of such a diagram? What are the applications of an activity diagram?
3. Differentiate between statechart diagrams and activity diagrams.

4. What is a statechart diagram? Discuss the notations used in a statechart diagram. Explain with the help of an example.
5. Consider an operation to calculate Fibonacci series of n numbers. Draw its activity diagram.
6. Consider a business process of meeting a new client. Draw its activity diagram. State the necessary assumptions.
7. Consider the process of creating a document in Microsoft Word. Draw its activity diagram. State the necessary assumptions.
8. Explain substates in a statechart diagram. How is a sequential substate different from a history substate?
9. Consider a traffic light system. Draw its statechart diagram. Specify the rules according to which the system is controlled.
10. Consider an automatic water controller system. This system is used for controlling water control. Draw its statechart diagram.
11. Differentiate between action states and activity states.
12. Construct an activity diagram for various phases of software development life cycle.
13. Consider an example of stack where two operations (push and pop) are allowed. There are four events, namely, new, push, pop and destroy with the following purposes:
 - *New*: To create an empty stack.
 - *Push*: To push an element in the stack, if space is available.
 - *Pop*: To pop out an element from the stack, if it is available.
 - *Destroy*: To destroy the stack after the completion of its requirement, i.e. instance of the stack class is destroyed.Identify the states and draw their statechart diagram.
14. How do you represent inheritance in tables? Explain with a suitable example.
15. Discuss the steps which are essential for converting entity classes into tables.
16. What are good programming practices for writing a program without considering the programming language? How are these practices different from coding standards?
17. What is the purpose of refactoring? Why should we improve the internal structure of the source code?
18. What is a persistent object? Why do we store these objects permanently in the secondary memory?
19. Consider the member entity class and create its table using SQL.
20. How do we implement the classes? What are good programming practices? Can we use a tool for implementation? If yes, name a few such tools.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

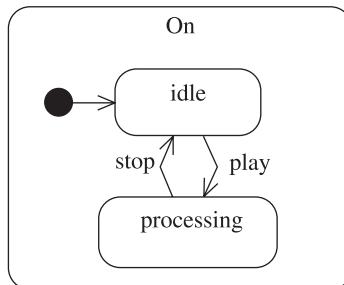
1. Activity diagrams are used to model:
 - (a) Static behaviour of the system
 - (b) Dynamic behaviour of the system
 - (c) Interaction between objects
 - (d) None of the above
2. Branching condition in an activity diagram is also known as:

(a) Branch condition	(b) Decision condition
(c) Guard condition	(d) Statement condition
3. Every fork should have:

(a) Multiple joins	(b) Single join
(c) No join	(d) None of the above
4. Fork and join are used to represent:

(a) Concurrent subactivities	(b) Serial subactivities
(c) Decision conditions	(d) Branch conditions
5. Match the following pairs:

1. Activity diagram	(a) requirements
2. Use case diagram	(b) states
3. Sequence diagram	(c) process
4. Statechart diagram	(d) messages
(a) 1-c, 2-a, 3-d, 4-b	(b) 1-c, 2-a, 3-b, 4-d
(c) 1-c, 2-b, 3-d, 4-a	(d) 1-b, 2-a, 3-d, 4-c
6. Which diagrams are used to model behaviour of an operation?
 - (a) Sequence diagrams
 - (b) Statechart diagrams
 - (c) Activity diagrams
 - (d) Use case diagrams
7. Which of the following is true about statechart diagram?



- (a) Play and stop are actions
- (b) Statechart diagram is invalid
- (c) On is a history state
- (d) On is a superstate

8. A statechart diagram describes:
 - (a) Interaction in a process
 - (b) Activities involved in an operation
 - (c) Attributes of an object
 - (d) Events invoked by an object
9. The state containing substates is known as:
 - (a) Superstate
 - (b) Concurrent state
 - (c) History state
 - (d) Base state
10. Swimlanes are represented by:
 - (a) Vertical lines
 - (b) Horizontal lines
 - (c) Dotted lines
 - (d) Directed arrows
11. The difference between activity diagrams and statechart diagrams is:
 - (a) Activity diagram models the states of an object's lifetime, whereas statechart diagram models the sequence of activities in a process or operation
 - (b) Statechart diagram models the states of an object's lifetime, whereas activity diagram models the sequence of activities in a process or operation
 - (c) Statechart diagram models the states of an object's lifetime, whereas activity diagram models the interaction between objects
 - (d) None of the above
12. Stop state in an activity diagram is represented by:
 - (a) Filled circle
 - (b) Empty circle
 - (c) Filled circle inside a circle
 - (d) Filled rectangle
13. Which one of the following model persistent data?
 - (a) Control classes
 - (b) Entity classes
 - (c) Interface classes
 - (d) Abstract classes
14. Parallel flow in an activity diagram is represented by:
 - (a) Swimlanes
 - (b) Guard conditions
 - (c) Concurrent states
 - (d) Fork and join
15. In a statechart diagram, history state is represented by:
 - (a) H inside a big circle
 - (b) H inside a small circle
 - (c) HS inside a small circle
 - (d) S inside a small circle
16. Which symbol is used to depict parallel activities in an activity diagram?
 - (a) Swimlanes
 - (b) Synchronization bar
 - (c) Activity
 - (d) Directed arrow
17. Which of the following are **not** included in an activity diagram?
 - (a) Events
 - (b) Transitions
 - (c) Actions
 - (d) Methods
18. The purpose of refactoring is to:
 - (a) Add new functionality
 - (b) Remove defects in the source code
 - (c) Handle change in environment
 - (d) Improve the internal structure of the source code

19. The advantages of refactoring include:
 - (a) Ease of handling new functionality
 - (b) Ease of enhancement
 - (c) Improvement of the internal structure of the source code
 - (d) All of the above
20. Good programming practices do **not** include:

(a) High use of if statement	(b) Use of error messages
(c) Use of documentation	(d) Use of global data rarely

Further Reading

The concepts of state machine are well explained in the following book:

Booch, G., Jacobson, I. and Rumbaugh, J., *The Unified Modeling Language (Version 2.2)*. The Object Management Group, 2008.

The coding rules for Java applications can be found in:

<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

Ambler, S., *Writing Robust Java Code—Java Coding Standards*. AmbySoft, 2000.

Bloch, J., *Effective Java—Programming Language Guide*. Boston, MA: Addison-Wesley, 2001.

Daconta, M.C., Monk, E., Keller, J.P. and Bohnenberger, K., *Java Pitfalls*. New York: John Wiley & Sons, 2000.

The following book describes the process of refactoring and explains the techniques for refactoring:

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley, 1999.

The detailed sources and information on refactoring can be obtained from:

www.refactoring.com

8

Software Quality and Metrics

The aim of object-oriented software engineering is to build a high-quality maintainable software system. The software quality process and assessment **should begin at the start** of the software development **life cycle** phases. In this chapter, we describe the basic concepts, elements and models of software quality.

The metrics are required to capture the object-oriented concepts such as **coupling**, **cohesion**, **inheritance** and **polymorphism**. The metrics are used to assess the quality of the product or process used to build it. We describe object-oriented metrics that capture the structural quality of object-oriented design and source code.

8.1 What is Software Quality?

What is the meaning of the term ‘quality’? It is very important but difficult to define quality. The Institute of Electrical and Electronics Engineers (IEEE) defines quality as:

1. *The degree to which a system, component, or process meets specific requirements.*
2. *The degree to which a system, component, or process meets customer or user needs or expectations.*

In terms of software quality, the software must conform to both functional and non-functional requirements specified by the customer or user. For example, if two cars meet their specified speed, standard, style and performance, then they are said to meet the specified requirements. If the product meets the customer’s requirements, the customer feels satisfied and the product is expected to be of high quality. The goal of software quality is to determine:

1. How well is the design of the software?
2. How well the software conforms to the developed design?

8.1.1 Software Quality Attributes

Software quality can be measured in terms of attributes. The attribute domains that are required to define for a given software are as follows:

1. Functionality
2. Usability
3. Testability
4. Reliability
5. Maintainability
6. Adaptability

These attribute domains can be further divided into attributes that are related to software quality and are given in Figure 8.1. The details of software quality attributes are given in Table 8.1.

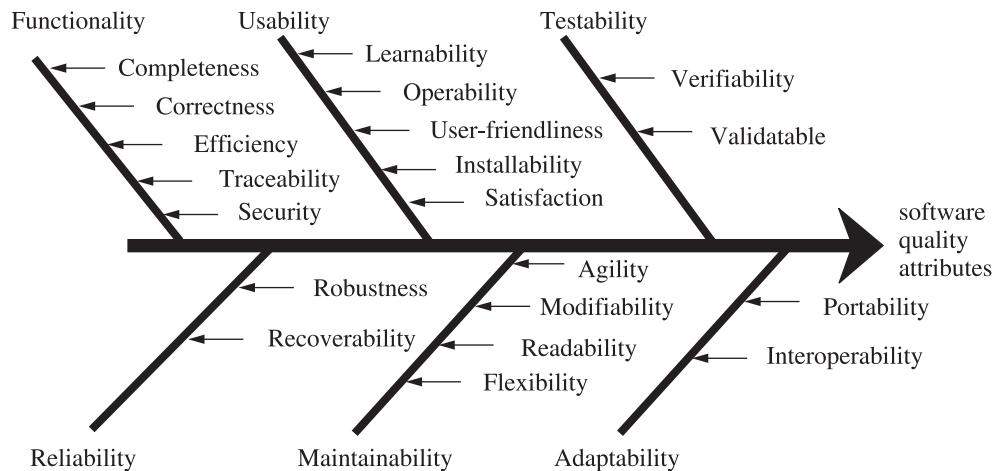


Figure 8.1 Software quality attributes.

Table 8.1 Software quality attributes

Functionality: The degree to which the purpose of the software is satisfied.

1	Completeness	The degree to which the software is complete.
2	Correctness	The degree to which the software is correct.
3	Efficiency	The degree to which the software requires resources to perform a software function.
4	Traceability	The degree to which requirement is traceable to the software design and source code.
5	Security	The degree to which the software is able to prevent unauthorized access to the program data.

Usability: The degree to which the software is easy to use.

1	Learnability	The degree to which the software is easy to learn.
2	Operability	The degree to which the software is easy to operate.
3	User-friendliness	The degree to which the interfaces of the software are easy to use and understand.

(Contd.)

Table 8.1 Software quality attributes (*Contd.*)

4	Installability	The degree to which the software is easy to install.
5	Satisfaction	The degree to which the user feels satisfied with the software.
Testability: The ease with which the software can be tested to demonstrate the faults.		
1	Verifiability	The degree to which the software deliverable meets the specified standards, procedures and process.
2	Validatable	The ease with which the software can be executed to demonstrate whether the established testing criterion is met.
Reliability: The degree to which the software performs failure-free functions.		
1	Robustness	The degree to which the software performs reasonably under unexpected circumstances.
2	Recoverability	The speed with which the software recovers after the occurrence of a failure.
Maintainability: The ease with which the faults can be located and fixed, quality of the software can be improved or software can be modified in the maintenance phase.		
1	Agility	The degree to which the software is quick to change or modify.
2	Modifiability	The degree to which the software is easy to implement, modify and test in the maintenance phase.
3	Readability	The degree to which the software documents and programs are easy to understand so that the faults can be easily located and fixed in the maintenance phase.
4	Flexibility	The ease with which changes can be made in the software in the maintenance phase.
Adaptability: The degree to which the software is adaptable to different technologies and platforms.		
1	Portability	The ease with which the software can be transferred from one platform to another platform.
2	Interoperability	The degree to which the system is compatible with other systems.

8.1.2 Elements of a Quality System

The software quality process should start at the beginning of the software development life cycle and should carry out throughout the software development life cycle. In order to produce a quality product, **the voice of the customer must be incorporated and then translated into software requirements**. These requirements must be properly defined and documented. Hence, the customer's satisfaction should be the prime concern of the developer. The quality system consists of 10 elements as given below:

1. Standards, processes and metrics
2. Reviews and audits
3. Software testing
4. Defect management and trend analysis
5. Configuration management
6. Risk management activities

7. Supplier control
8. Training
9. Documentation
10. Safety and security

The software development life cycle consists of various phases that divide the software development work into various parts. The software development life cycle phases are given below:

1. **Requirement analysis phase**—gathering and documentation of requirements.
2. **Design phase**—preliminary and detailed design of the software.
3. **Implementation and unit testing phase**—development of source code and initial testing of independent units.
4. **Integration and system testing phase**—testing the integrated parts of various units and the system as a whole.
5. **Operational phase**—delivering and installing the software at the customer's site.
6. **Maintenance phase**—removing defects, accommodating changes and improving the quality of the software after it goes into the operational phase.

These phases are non-overlapping (e.g. waterfall model) or overlapping and incremental (e.g. unified process model), depending on the type of the software development life cycle model being used.

The elements of the quality system are related with the phases of software development life cycle and these associations are shown in Figure 8.2. The associations depict the period for which the element of the quality system is more closely related to the phases of software development life cycle.

	Requirement analysis	Design	Implementation and unit testing	Integration and system testing	Operational	Maintenance
Standards, processes and metrics	√	√	√	√	√	√
Reviews and audits	√	√	√	√		√
Software testing	√	√	√	√	√	√
Defect management and trend analysis		√	√	√	√	√
Configuration management		√	√	√	√	√
Risk analysis and assessment	√	√	√	√		√
Supplier control		√	√	√		√
Training	√	√	√	√	√	√
Documentation	√	√	√	√		√
Safety and security			√	√		√

Figure 8.2 Associations between elements of quality system and life cycle phases.

Each of the 10 elements of the quality system is discussed as follows:

Standards, Processes and Metrics

The standards, processes and metrics that need to be followed and can be applied in the software development should be identified. The standards provide procedures that must be enforced during the software development life cycle. The standards may be defined by the Institute of Electrical and Electronics Engineers (IEEE), American National Standards Institute (ANSI), or International Organization for Standardization (ISO).

All aspects of the software development life cycle may be covered by the standards. They may specify the format of documentation of software requirement or may guide a developer in developing source code by following specified coding practices and standards. The standards that are important and feasible to apply and whose progress can be monitored must be chosen by an organization. These standards, thus, can be enforced on the software development life cycle.

The standards should be supported by concrete, well-defined and effective processes so that they are effectively implemented. A process is a collection of activities that are required to produce a good quality product. An effective process is well practised, enforced, documented and measured.

The metrics must be used to measure the effectiveness of software processes and practices followed during the software development. The metrics must be selected based on the aim to increase the customer's satisfaction level.

Reviews and Audits

Reviews are conducted as a part of verification activities. A detailed discussion on verification activities is provided in Chapter 9. Reviews are very effective as they are conducted in early phases of software development. They minimize the probability of occurrence of faults in the earlier phases of software development, i.e. before the validation activities begin. Reviews are cost effective and consume less cost. They increase the confidence about the correctness of the software under development. Reviews are carried out throughout the software development life cycle in order to verify the correctness of the documents and programs. Reviews are of two types: formal reviews and informal reviews. The informal reviews are carried out throughout the software development life cycle and are known as in-process reviews. Formal reviews are carried out at the end of the software development life cycle phase.

Reviews find faults well before the validation process begins. The cost of removal of such faults will be very reasonable as compared to those faults which we may find in the later phases of software development. Hence, the sooner the faults are detected in the software development life cycle, lower is the cost of the software as shown in Figure 8.3.

If an error is found and fixed in the specification and analysis phase, it hardly costs anything. We may say this as "1 unit of cost" for fixing an error during the specification and analysis phase. The same error, if propagated to design, may cost 10 units and, if further propagated to coding, may cost 100 units. If it is detected and fixed during the testing phase, it may lead to 1000 units of cost. If it could not be detected even during testing, and found by the customer after release, the cost may increase to 10000 units.

Reviews include software requirement specification review, software design review, software verification and validation plan or report review, user documentation review, etc. Physical and functional audits are part of formal audits. Physical audits verify whether the final product complies with the final documentation. Functional audits verify whether the developed software complies with software requirements in order to check that all the requirements have been met.

Software Testing

Software testing increases the confidence in the software by assuring that the requirements have been met. It includes both the verification and validation activities. The basic testing (refer to Chapter 6) consists of test planning, test cases, test design and procedures, and test reports.

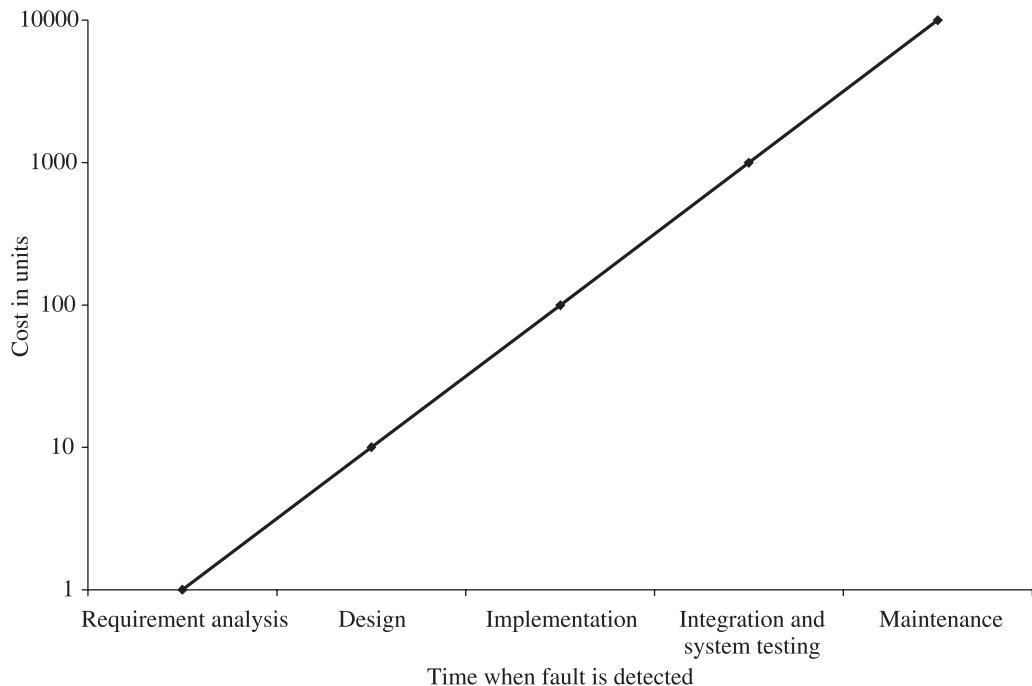


Figure 8.3 Phase-wise cost of fixing a fault.

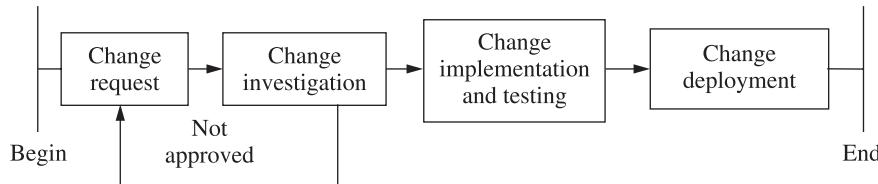
Test planning starts in parallel to requirement phases of software development. The test cases to be used in acceptance testing are also generated in this phase. By generating test cases in the requirements phase, the requirements are verified whether they are testable or not. Each test case includes inputs, expected results and test case pass/fail details. Test design and procedure guides the execution of testing.

The validation testing begins as soon as the source code is available for execution. The results generated are properly documented and recorded. Any unexpected result needs further corrective action. Finally, the acceptance testing is carried out by the user.

Defect Management and Trend Analysis

Defect management includes fault detection and correction activities. It consists of all the record of discrepancies observed in the software. Defect management is a part of change management activity. The typical cycle of change request and management is shown in Figure 8.4.

The change procedure allows to manage the faults reported, fixed and closed. These records will help in future projects in order to improve the processes so that the occurrences of some pattern of defects can be removed or at least reduced. The defects are reported by software defect report. The format of this report is given in Table 8.2 and the meanings of some of the fields are given in Table 8.3.

**Figure 8.4** Typical change procedure.**Table 8.2** Format of software defect report

Project name:	
Defect reported by:	
Defect description:	
Defect found in:	
Severity:	
Priority:	
Phase in which defect occurred:	
Phase in which defect is found:	
Process through which defect is detected:	
Estimate cost of correcting the defect:	
Actual cost of correcting the defect:	
Defect closed by:	

Table 8.3 Meaning of some fields in software defect report

Field	Meaning
Severity	Severity is defined as the impact of failure on the operation of the software. It can be categorized as: 1. Critical: It could be harmful to the life of the individual or a great threat to the existence of the organization. 2. Serious: It brings loss to the organization. 3. Moderate: It is a temporary loss or threat. 4. Trivial: It has very less impact.
Priority	Priority is defined as the speed with which the defect must be addressed. The priority can be categorized as: 1. Very high: The defect must be addressed immediately. 2. High: The defect has high impact on the existing work. 3. Medium: The defect has medium impact on the existing work. 4. Low: The defect does not interfere with the existing working.
Phase in which defect occurred	It specifies the phase of the software development life cycle in which the defect was introduced. 1. Requirement 2. Design 3. Implementation 4. Testing 5. Maintenance

(Contd.)

Table 8.3 Meaning of some fields in software defect report (*Contd.*)

Field	Meaning
Phase in which defect is found	<p>It specifies the phase of the software development life cycle in which the defect was detected.</p> <ol style="list-style-type: none"> 1. Requirement 2. Design 3. Implementation 4. Testing 5. Maintenance
Process through which defect is detected	<p>It specifies the process through which the defect is detected.</p> <ol style="list-style-type: none"> 1. Peer Review 2. Inspection 3. Walkthrough 4. Audit 5. Testing

Configuration Management

The management of changes is very important to ensure that the state in which the software is known at any given stage in the software development life cycle. It consists of identification and record of each item of the software. This helps in keeping track of each software item at any point of time in the project. Thus, the correct version of the software item is always known. Any change in the document or source code is assigned a new identifier.

The configuration management also involves approving the changes suggested by anyone during the software development. This prevents unauthorized changes being made in the software. Thus, each change that is necessary is implemented as we move ahead in the software development life cycle. The configuration management activities are addressed in detail in Chapter 10.

Risk Management Activities

A project may consist of several types of risks. Risk management is an essential activity in any project. The risks can vary from availability of skilled professionals, poor communication among developers to more severe such as identification of incorrect requirements, overrun of budget, unable to meet delivery deadlines, improper construction of design, etc.

The risk management activities include risk identification, determining of impact of the risk, prioritization of risk, and taking corrective actions to reduce the risks.

Supplier Control

The software that is purchased by a supplier must be assured in terms of quality. The purchased software should meet the software requirements.

Training

Training involves educating the developer about a new technique or tool or training the users about the software operation and use.

The quality of the software can only be assured when the developers are trained on the various technologies or tools to be used in the project. For example, if the project requires the use of Rational Rose software for analysis and design purpose, then the designers must be taught

about its use before the beginning of the analysis and design of the software. Similarly, the use of testing tools, debugging tools, documentation standards, platforms, etc. must be taught before they can be applied in the software development.

Once the software has been developed, the user must also be trained about the objectives and use of the software. The purpose of user interfaces, navigation items in the software, installation procedure of the software, data entry provisions, report generation mechanisms, system administration part, etc. must be taught to the user. In addition, all the things required to promote the effective use of the software must be explained to the user.

The system administrator must be taught of the environment in which the system must run, hardware requirements of the software, handling of unexpected or exceptional event and recovering from any failure. Security and backup mechanisms must also be explained to the system administrator.

The software quality professional must always keep themselves updated about any need of training of developer or users of the software. The software quality professional must also keep informing the higher authorities about the requirement of any training session to meet the quality of the software.

Documentation

Documentation is a very important part of the quality system. During the software development phases, the SRS document, SDD, test plans and test cases, user manuals and system guides, etc. must be produced. The specified documentation standards must be followed in order to create these documents. The documentation helps strongly in the debugging process and hence the maintenance of the product.

The documentation done in the requirement phase guides the design process and the documentation done in the design phase serves as the basis of implementation of the software. The documentation in the source code enhances its readability and understandability and helps in debugging activities. The minimum documents that must be produced by any software under development are given in Table 8.4.

Table 8.4 Recommended documentation for a software

S. No.	Document name
1	Software requirement specification
2	Software design description document
3	Test plans
4	Test cases
5	Test reports
6	Risk management plan
7	User manual
8	Operation manual
9	Installation guide

Safety and Security

The safety and security of the software is essential for the project. The safety of implication of the software failure after it is installed and the security of the program data are two major concerns for every software project.

The safety critical software is that whose failure could have an impact on the lives of the humans or heavy loss in terms of cost or resources. The world has seen many failures due to small faults in the software. For example, Ariane 5 failed due to small conversion/overflow error in the software. Similarly small error in missile causes death of many soldiers in the war. Y2K failure and other financial software failures are also examples of the software failure.

The failures that can affect the safety of human or that can have heavy losses or hardware damages are very critical to be identified and removed. Hence, the safety critical software should be adequately planned, assessed, monitored and tested.

The security of the data against accidental modifications or unauthorized access is a very critical issue. The place where the software is deployed must also be secure and protected against any damage or destruction. The data of the software should be protected from hackers, attackers and viruses. If the software data becomes inaccurate, it affects the software behaviour and the results obtained will not be correct.

The software quality professional must keep the management people aware of the security concerns. There should also be provisions of recovery from a catastrophic failure.

8.2 Software Quality Models

Many quality models are available in literature. Some popular models are discussed in subsequent sections.

8.2.1 McCall's Software Quality Model

McCall proposed a software quality model in 1997 which included many quality factors (McCall 1977). The aim of this software quality model is to reduce the gap between users and the developers. The model is divided into two levels of quality attributes: quality factors and quality criteria. The model covers quality factors from three levels of perspective: product operation (operation characteristics of the software), product revision (the extent to which the software can be modified), and product transition (the quality of the software to adapt to new environment) as shown in Figure 8.5.

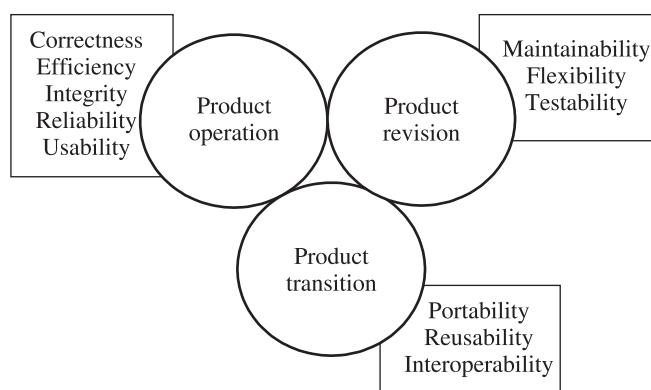


Figure 8.5 McCall's software quality factors.

The quality of product operation depends upon correctness, efficiency, integrity, reliability, and usability. Product revision includes maintainability, flexibility and testability. Product transition includes portability, reusability and interoperability. Most of the quality factors are explained in Table 8.1. The remaining quality factors are summarized in Table 8.5.

Table 8.5 Additional software quality factors

S. No.	Quality factors	Purpose
1	Integrity	The extent to which access to software or data by the unauthorized persons can be controlled.
2	Flexibility	The effort required to modify an operational program.
3	Reusability	The extent to which a program can be reused in other applications.
4	Interoperability	The effort required to couple one system with another.

The second level of quality attributes is known as quality criteria and is shown in Figure 8.6. McCall software quality management consists of 11 quality factors which are mapped into 23 quality criteria. However, users and developers are concerned with the quality factors only. The quality criteria provide insight into the quality attributes of the system.

The quality metrics can be used to capture the aspects of the quality criteria. The subjective assessment of the quality criteria can be made by rating them on a scale from 0 (very bad) to 10 (very good). However, the use of such assessment is difficult as different people may rate the same criterion differently. This makes correct quality assessment almost impossible.

8.2.2 Boehm's Software Quality Model

The second basic software quality model is given by Barry W. Boehm in 1978 (Boehm, 1978). It is similar to McCall's software quality model in the sense that it also presents software quality attributes in the form of hierarchy: primitive characteristics, intermediate-level characteristics and high-level characteristics. The levels of characteristics are shown in Figure 8.7.

The high-level characteristics represent primary use of the system. The following primary level factors are addressed by the high-level characteristics:

- *As a utility:* The ease with which the software can be used in its present form.
- *Maintainability:* The ease with which the software can be understood, modified and tested.
- *Portability:* The ease with which the software can be used by changing it from one platform to another platform.

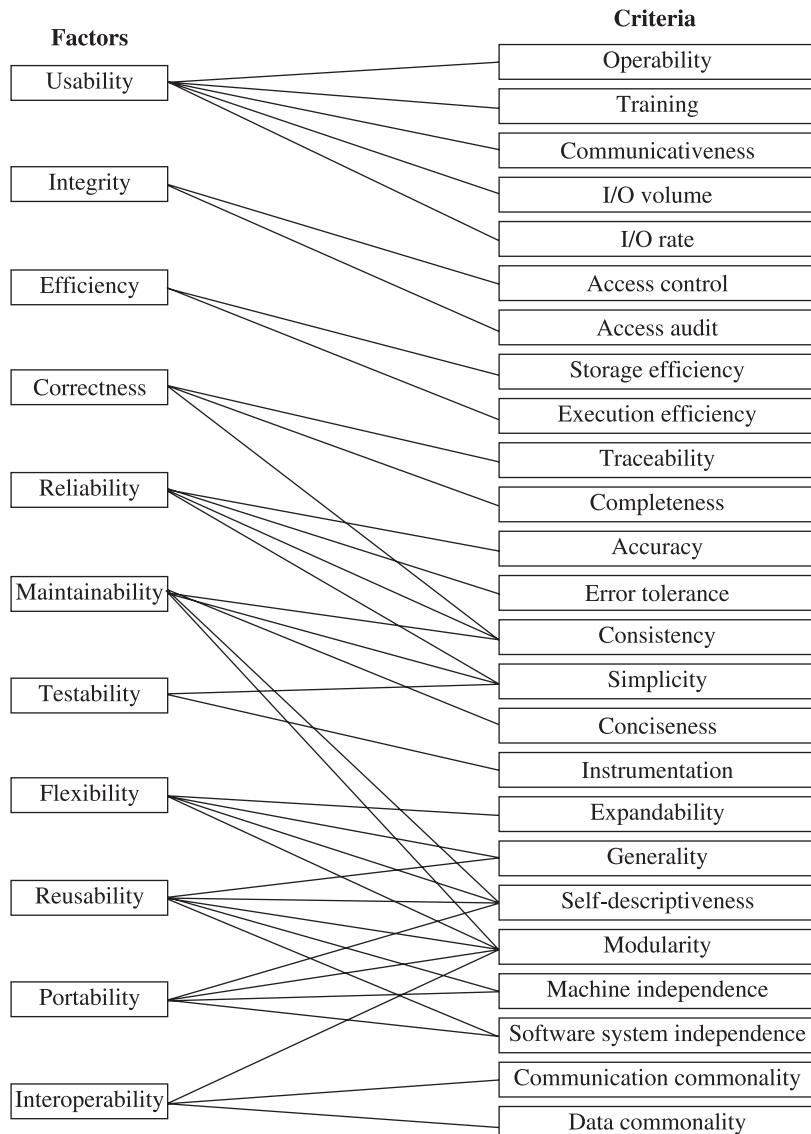


Figure 8.6 Software quality factors mapped into quality criteria.

The intermediate levels represent the quality that is expected from the software. The lowest level represents the primitive metrics that will define and measure the quality of the model. Boehm includes the characteristics of the hardware performance that are not present in McCall's model (Pfleeger, 2001). The human engineering aspect is sometimes very critical. The software must be easy to access, use and understand no matter how good it is in performance. Boehm also gives special focus on maintainability of the software.

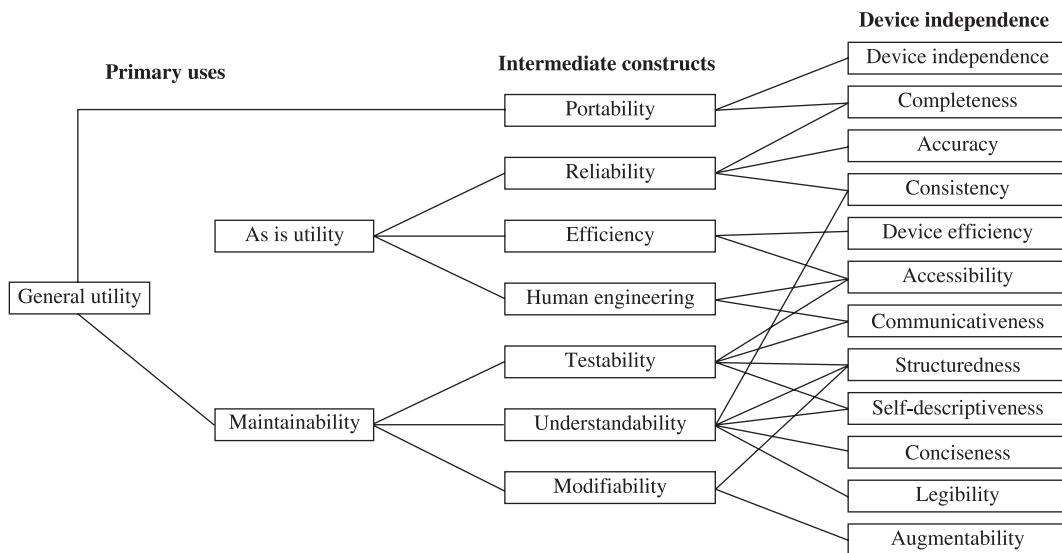


Figure 8.7 Boehm's software quality model

8.2.3 ISO 9000

The International Organization for Standardization (ISO) made different attempts to improve the quality with ISO 9000 series. It is a well-known and widely used series. The ISO 9000 series of standards is not only for software, but it is a series of five related standards that are applicable to a wide range of applications such as industrial tasks including design, production, installation and servicing. **ISO 9001 is the standard that is applicable to software quality**. The aim of ISO 9001 is to **define, understand, document, implement, maintain, monitor, improve and control the following processes:**

1. Management responsibility
2. Quality system
3. Contract review
4. Design control
5. Document control
6. Purchasing
7. Purchaser-supplied product
8. Product identification and traceability
9. Process control
10. Inspection and testing
11. Inspection, measuring and test equipment
12. Inspection and test status
13. Control of nonconforming product
14. Corrective action
15. Handling, storage, packaging and delivery
16. Quality records

17. Internal quality audits
18. Training
19. Servicing
20. Statistical techniques

In order to obtain ISO certification, a formal audit on the above 20 clauses is to be done and the result of the audit should be positive for all the 20 clauses. The failure of software companies during initial audit ranges from 60% to 70%.

8.2.4 ISO 9126

ISO 9126 is an international standard that is guided by ISO/IEC. It consists of six characteristics given below and shown in Figure 8.8:

1. **Functionality**: It is an essential feature of any software product that achieves the **basic purpose** for which the software is developed. For example, the LMS should be able to maintain book details, maintain member details, issue book, return book, reserve book, etc. Functionality includes the essential features that a product must have. It includes suitability, accuracy, interoperability and security.
2. **Reliability**: Once the functionality of the software has been completed, the reliability is defined as the **capability of defect-free operation of software for a specified period of time and given conditions**. One of the important features of reliability is **fault tolerance**. For example, if the system crashes, then when it recovers the system should be able to continue its normal functioning. Other features of reliability are maturity and recoverability.
3. **Usability**: The ease with which the software can be used for each specified function is another attribute of ISO 9126. The ability to **learn, understand and operate the system is the sub-characteristics of usability**. For example, the ease with which the operation of cash withdrawal function of an ATM system can be learned is a part of usability attribute.
4. **Efficiency**: This characteristic concerns with **performance** of the software and **resources used** by the software under specified conditions. For example, if a system takes 15 minutes to respond, then the system is not efficient. Efficiency includes time behaviour and resource behaviour.
5. **Maintainability**: The ability to **detect and correct faults** in the maintenance phase is known as maintainability. Maintainability is affected by the **readability, understandability and modifiability** of the source code. The ability to diagnose the system for identification of cause of failures (analysability), the effort required to test a software (testability) and the risk of unexpected effect of modifications (stability) are the sub-characteristics of maintainability.
6. **Portability**: This characteristic refers to the ability to **transfer** the software from **one** platform or environment **to another** platform or **environment**.

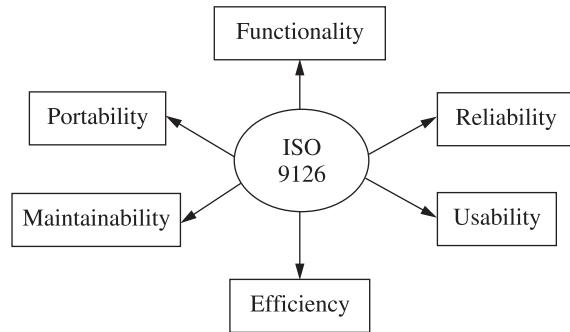


Figure 8.8 ISO 9126 model.

The major difference between ISO 9126 and McCall and Boehm quality models is that ISO 9126 follows a strict hierarchy. Each characteristic is related to one attribute only.

8.2.5 Capability Maturity Model

The initial version of the Capability Maturity Model (CMM) was developed at Software Engineering Institute (SEI), Carnegie Mellon University, USA. The CMM provides a framework which defines the key elements of an effective and efficient software process. The CMM defines the growth path of an organization from an undisciplined, immature approach to a disciplined, mature approach. The CMM covers key practices that help to plan, improve and manage software development life cycle activities. These software development practices, when followed by organizations, improve the resource utilization and quality of the software.

The CMM is categorized into five maturity levels as shown in Figure 8.9: initial, repeatable, defined, managed and optimizing.

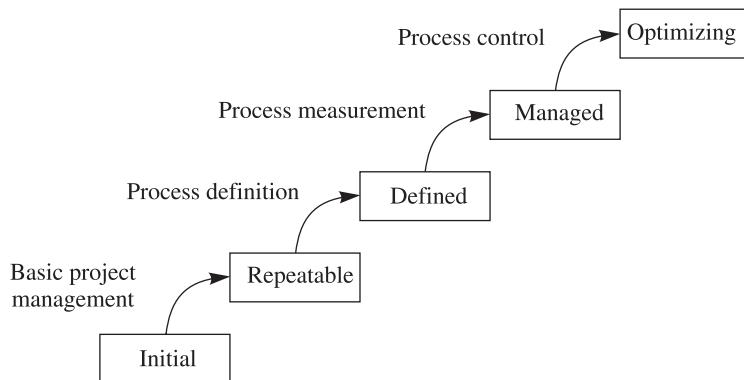


Figure 8.9 Levels of CMM.

Maturity Level 1: Initial

This is the lowest level and at this level, organizations do not have a stable environment for following software engineering and management practices. This leads to ineffective planning

and management of systems. At this level, everything is carried out on an ad hoc basis. The success of a project depends on a competent manager and a good software project team. The absence of sound management principles cannot even be covered by strong engineering principles. The capability of software process is undisciplined at this level as the processes are continuously changing or being modified as the software development progresses. The performance completely depends upon individuals rather than on the organization's abilities. As the process totally depends upon the individual, it changes when the staff changes. Thus, time and cost of development are unpredictable.

Maturity Level 2: Repeatable

At level 2, procedures and policies for managing the software product are established. Planning and managing of new software projects are not ad hoc rather they are based on the past similar software projects. The aim is to develop effective software processes and practices that enable the organization to repeat the successful past practices.

An effective process is one which is efficient, defined, documented and measured, and has the ability for improvement. Unlike level 1, in this level, the managers identify problems and take corrective actions to prevent these problems from converting into crisis. The project managers keep track of cost and time. The standards are defined and the organizations ensure that these standards are actually followed by the project team. Hence, the process maturity level 2 can be described as disciplined because it is well planned, stable and past successful practices are repeatable.

Maturity Level 3: Defined

At maturity level 3, the established processes are documented. Software processes established at this level are used by project managers and technical staff to improve the performance more effectively. These are known as standard software processes. The organizations also conduct a training programme so that the staff and managers are well aware of the sources and understand their assigned tasks.

Different projects in the organization modify the standard processes and practices to construct their own defined software processes and practices. These defined software processes are specific to the requirement of individual characteristics of the given project.

A well-defined process includes inputs, standards, practices and procedures to carry out the work, verification procedures, outputs and criteria of completion. The maturity level of a process at level 3 can be described as standard and completed.

Maturity Level 4: Managed

At this level, goals for quality of the software product are set. The organization's measurement program measures the quality of the software process and standards. Hence, the software processes and standards are evaluated using measures and this helps in giving indication of the trends of quality in processes and standards followed in the organization. The limits specifying thresholds are established; when these limits are exceeded, then corrective action is to be taken.

Maturity Level 5: Optimizing

At level 5, the focus of the organization is on continuous improvement of the software processes. The software projects identify the causes of the software defects and evaluate software processes to prevent defects from reoccurring. The process maturity at level 5 can be defined as ‘continuously improving’. Improvement occurs due to advances in existing processes and innovations using new technologies.

The maturity levels with their corresponding key process areas (KPAs) are shown in Table 8.6.

Table 8.6 Key process areas of CMM

Maturity level	Characterization	Key process areas
1—Initial	Ad hoc process	No KPA
2—Repeatable	Disciplined and mature process	<ul style="list-style-type: none"> • Requirements management • Software project planning • Software project tracking and oversight • Software subcontract management • Software quality assurance • Software configuration management
3—Defined	Standard process	<ul style="list-style-type: none"> • Organization process definition • Training program • Integrated software management • Software product engineering • Inter-group coordination • Peer reviews
4—Managed	Predictable process	<ul style="list-style-type: none"> • Quantitative process management • Software quality management
5—Optimizing	Improving process	<ul style="list-style-type: none"> • Defect prevention • Technology change management • Process change management

The past experience reports show that moving from level 1 to level 2 may take 3 to 5 years. The CMM is becoming popular and many software organizations are aspiring to achieve CMM level 5. The acceptability and pervasiveness of the CMM activities are helping the organizations to produce a quality software.

8.3 Measurement Basics

A key element of any engineering process is measurement. Measures are used to better understand the attributes of the model that we create. But, most important, we use measurements to assess the quality of the engineered product or the process used to build it. Unlike other engineering disciplines, software engineering is not grounded by the basic quantitative laws of physics. Absolute measurements, such as voltage, mass, velocity or temperature, are uncommon in the software world. Instead, we attempt to derive a set of indirect measures that lead to metrics that provide an indication of the quality of some representation of software.

The following questions arise during software development:

1. How to measure the cost and development duration of the software?
2. How to estimate the size of the software?
3. How many faults can we expect?
4. What is the complexity of the module/class?
5. What is the reliability of the software?
6. Is the test suite weak or strong?
7. Are the processes appropriate?
8. Which technique is more effective?
9. When to stop testing?
10. When to release the software?
11. What about the cohesion and coupling of the module/class?
12. What is the reusability of the module/class?

The above information can only be gathered by employing metrics. These metrics will guide the project managers in improving software product and process during the project development.

8.3.1 What are Software Metrics?

Software metrics are initiated with the belief that they will improve software engineering, estimation and management practices. The rationale arises from the notion that “you cannot control what you cannot measure” (DeMarco, 1982). Software measures can help address the most critical issues in software development and provide support for planning, predicting, controlling and evaluating the quality of both software products and processes.

Goodman (1993) defined software metrics as:

The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.

The above definition covers a lot in itself. Software metrics are related to measures which, in turn, involve numbers for quantification. These numbers are used to produce a better product and improve its related process. Software metrics should be gathered from the beginning, when the cost and effort of the product is to be estimated, to the monitoring of the reliability and maintainability of the end product.

8.3.2 Application Areas of Metrics

Software metrics have wide areas of applications. One of the most established areas of measurement is cost and effort estimation. The project cost and effort are predicted in the early phases of software development. There are many models available for this purpose and these models include function point method (Albrecht and Gaffney, 1983), COCOMO81, COCOMOII (Boehm, 1981), and Putnam resource allocation model (Putnam and Putnam, 1984). These models are used to calculate effort (number of persons per month) or size (number of function points or number of lines of code produced). There are various research investigations to derive models in order to estimate the cost and effort of a project.

SRS is an important document produced during the software development life cycle. The metrics can be used to measure the readability, faults found during SRS verification, change request frequency, etc. In the design phase, the coupling and strength of modules/classes can also be measured using metrics.

Managing and controlling software process and product is an important area of software development. The measurement-based graphs and analysis provide customers and developers the direction in which the software should move. Many software developers use metrics in early phases of software development to measure the quality of the systems.

The need of prediction of software quality attributes has led to construction of various quality models in order to measure reliability, maintainability, testability, complexity, etc. The real benefits of these models are in large-scale systems where the testing persons are asked to focus their attention and resources on problematic and critical areas of the system. From the design phase, we can make software metrics captured in these models and then predict which modules/classes will need extra attention during the remainder of development. This can help management to focus resources on those modules/classes that cause most of the problems.

The use of metrics to provide measures of object-oriented constructs such as inheritance, polymorphism, etc. is also a well-established area. A large number of object-oriented metrics that are related with external software quality attributes have been designed and developed over the last decade.

Statement coverage metrics are also available that calculate the percentage of statements in the source code covered during testing. The effectiveness of test suite may be measured. The testing metrics include the number of failures experienced per unit of time, the number of paths in a control flow graph, the number of independent paths, percentage of statement coverage and percentage of decision statements covered.

The metrics can also be used to provide meaningful and timely information to the management. These metrics can help them to assess software quality, measure process effectiveness and productivity of the software. Hence, this will aid the management in effective decision making. The effective application of metrics can help the software developers to produce the quality software within budget and on time.

8.3.3 Categories of Metrics

The metrics can be categorized by the entities we need to measure. In software development, there are two entities that need to be measured:

1. Products are deliverables or documents produced during the software development.
2. Processes are set activities that are used to produce a product.

Processes use products produced from an activity and produce products that can be used by another activity. For example, a requirement document is a deliverable produced by the requirement phase and it serves as an input to the design activity. The metrics related to the product are known as product metrics and the metrics related to the process are known as process metrics. The process metrics measure the effectiveness of the process. These metrics can be applied at all the phases of software development.

The process metrics can be used to:

1. measure the cost of the process.
2. measure the time taken to complete the process.
3. estimate the efficiency and effectiveness of the process in order to determine which one is effective.
4. compare various processes in order to determine which one is effective.
5. guide future projects.

For example, during reviews we can measure the average cost consumed to find each fault by computing the ratio of cost to the number of faults occurred in reviews.

The product metrics can be used to assess the document or a deliverable produced during the software development life cycle. The examples of product metrics include size, functionality, complexity and quality. Documents such as SRS, user manual, etc. can be measured for correctness, readability and understandability.

The process and product metrics can be further categorized as internal or external attributes:

1. Internal attributes are those that are related to the internal structure of the product or the process.
2. External attributes are those that are related to the behaviour of the product or the process.

For example, the size of a class, structural attributes including complexity of the source code, number of independent paths, number of statements, and number of branches can be determined without executing the source code. These are known as internal attributes of a product or process. When the source code is executed, the number of failures encountered, user-friendliness of the forms and navigational items or response time of a module describe the behaviour of the software. These are known as external attributes of a process or product. The external attributes are related to the quality of the system. Figure 8.10 summarizes the categorization of metrics along with their examples.

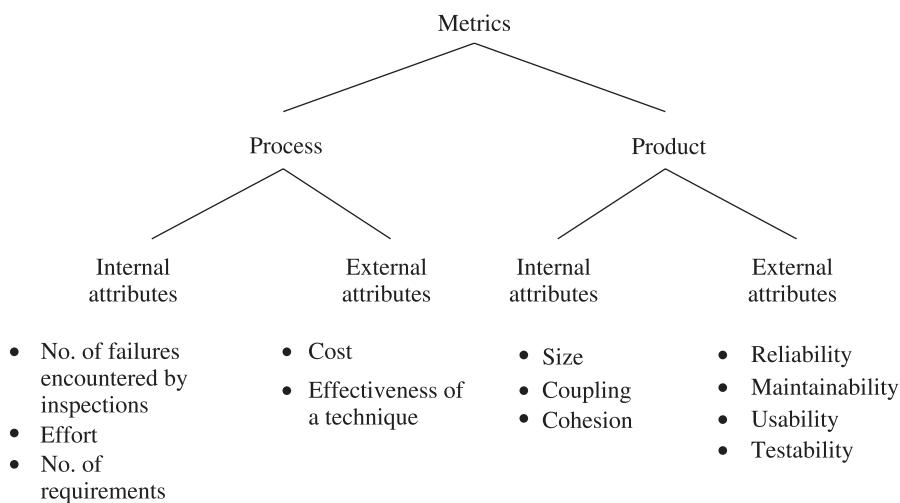


Figure 8.10 Categories of metrics.

8.3.4 Measurement Scale

There are two types of data—non-metric and metric. Non-metric type of data is of categorical or discrete type. For example, gender of a person can be male or female. In contrast, metric type of data represents amount or magnitude such as lines of source code.

Non-metric measurements can be measured either on a nominal scale or on an ordinal scale. In the nominal scale, the categories or classes of a variable are described. A metric has nominal scale when it can be divided into categories or classes and there is no ranking or ordering amongst the categories or classes. Thus, the number indicates the presence or absence of the attribute value. For example,

$$x = \begin{cases} 0, & \text{if not faulty} \\ 1, & \text{if faulty} \end{cases}$$

Class x is faulty or not faulty. Thus, there are two categories of faults—either they are present or they are not present.

Similar to nominal scales, a metric having ordinal scale can be divided into categories or classes; however, it involves ranking or ordering. In the ordinal scale, each category can be compared with another in terms of “higher than” or “lower than” relationship. For example, fault impact can be categorized as high, medium, and low. Numbers can be used to represent each category shown as follows:

$$\text{fault impact} = \begin{cases} 1, & \text{high} \\ 2, & \text{medium} \\ 3, & \text{low} \end{cases}$$

The metric scale provides higher precision permitting various arithmetic operations to be performed. Interval, ratio and absolute scales are of metric type. In the interval scale, on any part of the scale, the difference between two adjacent points is equal. The interval scale has an arbitrary zero point. Thus, on an interval scale, it is not possible to express any value in multiple of some other value on the scale. For example, a day with 100°F temperature cannot be said as twice as hot as a day with 50°F temperature. The reason is that on Celsius scale, 100°F is 37.8 and 50°F is 10. Thus, this relationship can be expressed as:

$$C = 5 \times \frac{F - 32}{9}$$

Ratio scales give more precision since they have all advantages of other scales plus an absolute zero point. For example, if weight of a person is 100 kg, then he/she is twice heavier compared to a person with 50 kg weight. Absolute scale simply represents counts. For example, the number of faults encountered during inspections can only be measured in one way by counting the faults encountered. Table 8.7 summarizes the differences between measurement scales with examples.

Table 8.7 Summary of measurement scales

Measurement type	Measurement scale	Characteristics	Transformation	Examples
Non-metric	Nominal	<ul style="list-style-type: none"> Order not defined Arithmetic not involved 	One-to-one mapping	Categorical classifications like type of language (C++, Java)
	Ordinal	<ul style="list-style-type: none"> Order defined Monotonic increasing function ($=, <, >$) Arithmetic not involved 	Increasing function $P(x) > P(y)$	Student grades, customer satisfaction level, employee capability levels
	Metric	<ul style="list-style-type: none"> $=, <, >$ No ratios Addition, subtraction Arbitrary zero point 	$P = xP' + y$	Temperatures, date and time
Metric	Interval	<ul style="list-style-type: none"> Absolute zero point All arithmetic operations possible 	$P = xP'$	Weight, height, length
	Ratio			
	Absolute	<ul style="list-style-type: none"> Simple count values 	$P = P'$	Number of faults encountered in testing

EXAMPLE 8.1 Consider the maintenance effort in terms of lines of source code added, deleted, and changes during maintenance phase classified between 1 and 5, where 1 means very high, 2 means high, 3 means medium, 4 means low and 5 means very low.

- What is the measurement scale for this definition of maintenance effort?
- Give an example to determine the criteria for estimating the maintenance effort levels for a given class.

Solution

- The measurement scale is ordinal since the variable maintenance effort is categorical (consisting of classes) and involves ranking or ordering amongst categories.
- For a given project, the following categorization may be used for maintenance effort:

$$\text{Maintenance effort} = \begin{cases} \text{very high, SLOC} > 10,000 \\ \text{high, } 7000 \leq \text{SLOC} < 10,000 \\ \text{medium, } 5000 \leq \text{SLOC} < 7000 \\ \text{low, } 1000 \leq \text{SLOC} < 5000 \\ \text{very low, SLOC} < 1000 \end{cases}$$

where SLOC stands for source lines of code.

8.3.5 Axiomatic Evaluation of Metrics on Weyuker's Properties

Several researchers recommend properties that software metrics should possess to increase their usefulness. For instance, Basili and Reiter (1979) suggest that metrics should be sensitive

to externally observable differences in the development environment, and must correspond to notions about the differences between the software artifacts being measured. However, most recommended properties tend to be informal in evaluation of metrics. It is always desirable to have a formal set of criteria with which the proposed metrics can be evaluated. Weyuker (1998) has developed a formal list of properties for software metrics and has evaluated a number of existing software metrics against these properties. Although many authors have criticized this approach (Briand et al., 1999; Zuse, 1990), yet it is a widely known formal analytical approach.

Weyuker's first four properties address how sensitive and discriminative the metric is. The fifth property requires that when two classes are combined, their metric value should be greater than the metric value of each individual class. The sixth property addresses the interaction between two programs/classes. It implies that the interaction between program/class A and program/class B is different than the interaction between program/class C and program/class B given that the interaction between program/class A and program/class C is the same. The seventh property requires that a measure to be sensitive to statement order within a program/class is the same. The eighth property requires that renaming of variables does not affect the value of a measure. The last property states that the sum of the metric values of a program/class could be less than the metric value of the program/class when considered as a whole (Henderson-Sellers, 1996).

Let u be the metric of program/class P and Q .

Property 1: This property states that

$$(\exists P), (\exists Q) [u(P) \neq u(Q)]$$

It ensures that no measure rates all programs/classes to be of the same metric value.

Property 2: Let c be a non-negative number. Then there are finite numbers of programs/classes with metric c . This property ensures that there is sufficient resolution in the measurement scale to be useful.

Property 3: There are distinct programs/classes P and Q such that $u(P) = u(Q)$.

Property 4: For an object-oriented system, two programs/classes having the same functionality could have different values.

$$(\exists P) (\exists Q) [P \equiv Q \text{ and } u(P) \neq u(Q)]$$

Property 5: When two programs/classes are concatenated, their metric should be greater than the metrics of each of the parts.

$$(\forall P)(\forall Q) [u(P) \leq u(P + Q) \text{ and } u(Q) \leq u(P + Q)]$$

Property 6: This property suggests non-equivalence of interaction. If there are two program/classes bodies of equal metric value, they when separately concatenated to a same third program/class, yield a program/class of different metric value.

For programs/classes P, Q, R

$$(\exists P)(\exists Q)(\exists R) [u(P) = u(Q) \text{ and } u(P + R) \neq u(Q + R)]$$

Property 7: This property is not applicable to object-oriented metrics (Chidamber and Kamerer, 1994).

Property 8: It specifies that “if P is a renaming of Q , then $u(P) = u(Q)$ ”.

Property 9: This property is not applicable to object-oriented metrics (Chidamber and Kamerer, 1994).

8.4 Analysing the Metric Data

The role of statistics is to function as a tool in analysing research data and drawing conclusions from it. The research data must be suitably reduced so that the same can be read easily and can be used for further analysis. Metric data can be analysed using one or many statistical techniques and meaningful inferences can be drawn.

8.4.1 Summary Statistics for Preexamining Data

The role of statistics is to function as a tool in analysing research data and drawing conclusions from it. The research data must be suitably reduced to be read easily and used for further analysis. Descriptive statistics concern development of certain indices or measures to summarize data. Data can be summarized by using measures of central tendency (mean, median and mode) and measures of dispersion (standard deviation, variance, and quartile).

Measures of Central Tendency

Measures of central tendency include mean, median and mode. These measures are known as measures of central tendency as they give us the idea about the central values of the data around which all the other data points have a tendency to gather. Mean can be computed by taking average values of the data set and is given as:

$$\text{Mean}(\mu) = \sum_{i=1}^N \frac{x_i}{N}$$

Median gives the middle value in the data set which means half of the data points are below the median value and half of the data points are above the median value. It is calculated as $(1/2)(n + 1)$ th value of the data set, where n is the number of data points in the data set.

The most frequently occurring value in the data set is denoted by mode. The concept has significance for nominal data.

EXAMPLE 8.2 Consider the following data set consisting of SLOC for a given project:

107, 128, 186, 222, 322, 466, 657, 706, 790, 844, 1129, 1280, 1411, 1532, 1824, 1882, 3442

Calculate mean, median and mode for it.

Solution Mean of SLOC (taking the sum of all the numbers and dividing by total data points) = 995.7647

Median of SLOC = 790

Since each value occurs only once in the data set, there is no mode for SLOC.

Measures of Dispersion

Measures of dispersion include standard deviation, variance and quartiles. Measures of dispersion tell us how the data is scattered or spread. Standard deviation calculates the distance of the data point from the mean. If most of the data points are closer to the mean, then the standard deviation of the variable is large. The standard deviation is calculated as given below:

$$\sigma_x = \sqrt{\frac{\sum(x - \mu)^2}{N}}$$

Variance is a measure of variability and is the square of standard deviation.

The quartile divides the metric data into four equal parts. For calculating quartile, the data is first arranged in ascending order. The 25% of the metric data is below the lower quartile (25 percentile), 50% of the metric data is below the median value and 75% of the metric data is below the upper quartile (75 percentile). Figure 8.11 shows the division of data set into four parts by using quartiles.

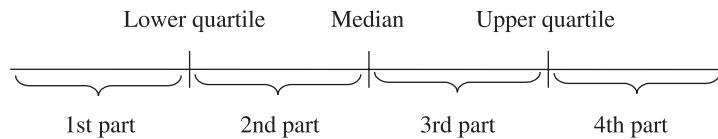


Figure 8.11 Quartiles.

The lower quartile (Q_1) is computed by:

1. finding the median of the data set
2. finding the median of the lower half of the data set.

The upper quartile (Q_3) is computed by:

1. finding the median of the data set
2. finding the median of the upper half of the data set.

The interquartile range is calculated as the difference between the upper quartile and the lower quartile and is given as

$$\text{Interquartile range (IQR)} = Q_3 - Q_1$$

EXAMPLE 8.3 Consider data set consisting of lines of the SLOC given in Example 8.2. Calculate standard deviation, variance and quartile for it.

107, 128, 186, 222, 322, 466, 657, 706, **790**, 844, 1129, 1280, 1411, 1532, 1824, 1882, 3442

Solution Lower quartile is the fifth value, i.e. $\frac{1}{2}(8 + 1)\text{th} = 322$

$$\text{Median} = 790$$

Upper quartile is the thirteenth value = 1411.

8.4.2 Metric Data Distribution

In order to understand the metric data, the starting point is to analyse the shape of the distribution of the data. There are a number of methods available to analyse the shape of the data. One of them is histogram through which a researcher can gain insight into the normality of the data. Histogram is a graphical representation of frequency of occurrences of the values of a given variable.

Figure 8.12 shows the frequency of continuous metric variable LOC in the form of bar charts. The bars show the frequency of values of LOC metric. The normal curve is superimposed on the distribution of values to determine the normality of the data values of LOC. Most of the data is left skewed or right skewed. These curves will not be applicable for discrete data (nominal or ordinal). For example, the classes may be faulty or non-faulty. Thus, the distribution will not be normal.

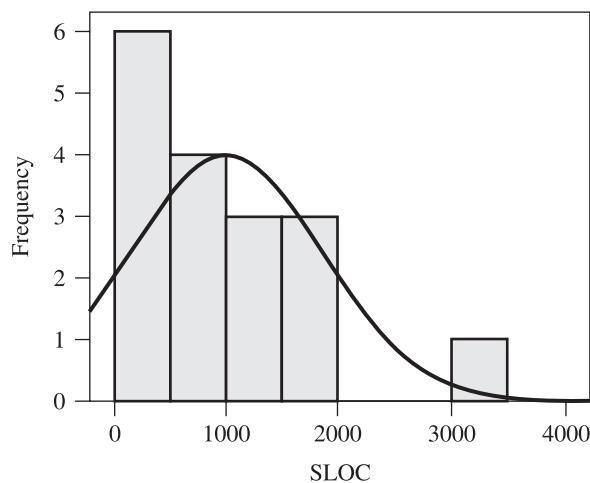


Figure 8.12 Normality curve for LOC variable.

The measures of central tendency such as mean, median and mode are all equal for normal curves. The normal curve is like a bell-shaped curve and within three standard deviations of the means, 96% of data occurs (Fenton and Pfleeger, 2004).

EXAMPLE 8.4 Consider the data sets given for three metric variables in Table 8.8. Determine the normality of these variables.

Table 8.8 Data set

Fault count	Cyclomatic complexity	Branch count
470.00	26.00	826.00
128.00	20.00	211.00
268.00	14.00	485.00
19.00	10.00	29.00

(Contd.)

Table 8.8 Data set (*Contd.*)

Fault count	Cyclomatic complexity	Branch count
404.00	15.00	405.00
127.00	11.00	240.00
263.00	14.00	464.00
94.00	10.00	187.00
207.00	13.00	344.00
42.00	7.00	83.00
24.00	10.00	47.00
94.00	6.00	163.00
34.00	9.00	67.00
286.00	10.00	503.00
104.00	12.00	175.00
82.00	8.00	147.00
20.00	7.00	39.00

Solution Figure 8.13 shows the bar charts to check the normality of the variables. The mean, median and standard deviation of the variables are given in Table 8.9. The normality of the data can also be checked by calculating mean and median. The mean, median and mode should be the same for a normal distribution. In Table 8.9, we compare the values of mean and median. The values of median are less than those of mean for variables fault count and branch count. Thus, these variables are not normally distributed. However, the mean and median of cyclomatic complexity do not differ by a larger value.

8.4.3 Outlier Analysis

Data points, which are located in an empty part of the sample space, are called *outliers*. These are the data values that are numerically distant from the rest of the data. For example, suppose one calculates the average weight of 10 students in a class, where most are between 51 pounds and 61 pounds, but the weight of one student is 210 pounds. In this case, the mean will be 72 pounds and the median will be 58 pounds. Hence, the median better reflects the weight of the students than the mean. Thus, the data point with the value 210 is an outlier, that is, it is located away from other values in the data set. Outlier analysis is done to find data points that are over influential and removing them is essential.

Once the outliers are identified, the decision about the inclusion or exclusion of the outlier must be made. The decision depends upon the reason why the case is identified as outlier. There are three types of outliers: univariate, bivariate and multivariate. Univariate outliers are those exceptional values that occur within a single variable. Bivariate outliers occur within the combination of two variables and multivariate outliers are present within the combination of more than two variables.

Box plots and scatter plots are two popular methods that are used for univariate and bivariate outlier detections. Box plots are based on median and quartiles. The upper and lower quartile statistics are used to construct a box plot. The median value is the middle value of the data

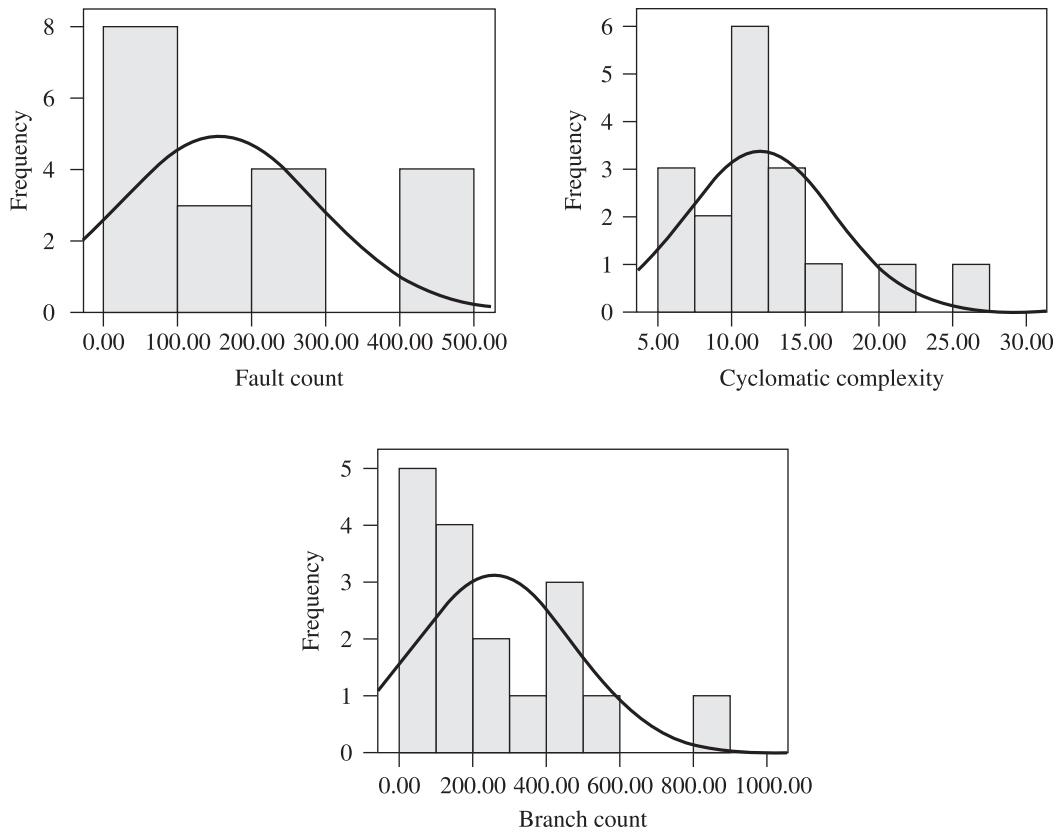


Figure 8.13 Normality curves.

Table 8.9 Descriptive statistics

Metric	Mean	Median	Std. deviation
Fault count	156.8235	108	137.7599
Cyclomatic complexity	11.88235	10	5.035901
Branch count	259.7059	187	216.9976

set; half of the values are less than this value and half of the values are greater than this value (Figure 8.14).

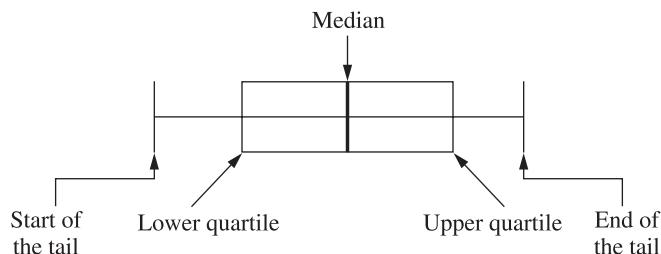


Figure 8.14 Example box plot.

The box starts at the lower quartile and ends at the upper quartile. The distance between the lower quartile and the upper quartile is called box length. The tails of the box plot specify the bounds between which all the data points must lie. The start of the tail is $Q_3 - 1.5 \times \text{IQR}$ and the end of the tail is $Q_3 + 1.5 \times \text{IQR}$. These values are truncated to the nearest values of the actual data points in order to avoid negative values. Thus, the actual start of the tail is the lowest value in the variable above $(Q_3 - 1.5 \times \text{IQR})$ and the actual end of the tail is the highest value below $(Q_3 + 1.5 \times \text{IQR})$.

Any value outside the start of the tail and the end of the tail is outlier and these data points must be identified as they are unusual occurrences of data values which must be considered for inclusion or exclusion. The box plots also tell us whether the data is skewed or not. If the data is not skewed, the median will lie in the centre of the box. If the data is left or right skewed, then the median will be away from the centre. For example, consider the LOC values for a sample project given below:

17, 25, 36, 48, 56, 62, 78, 82, 103, 140, 162, 181, 202, 251, 310, 335, 508

The box plot for the given data set is shown in Figure 8.15 with an outlier case number 17. The median of the data set is 103, lower quartile is 56 and upper quartile is 202. The interquartile range is 146. The start of the tail is $202 - 1.5 \times 146 = -17$ and end of the tail is $202 + 1.5 \times 146 = 421$. The actual start of the tail is the lowest value above -17, i.e. 17 and the actual end of the tail is the highest value below 421, i.e. 335, as shown in Figure 8.15. Thus, the case number 17 with value 508 is above the end of the tail and hence is an outlier.

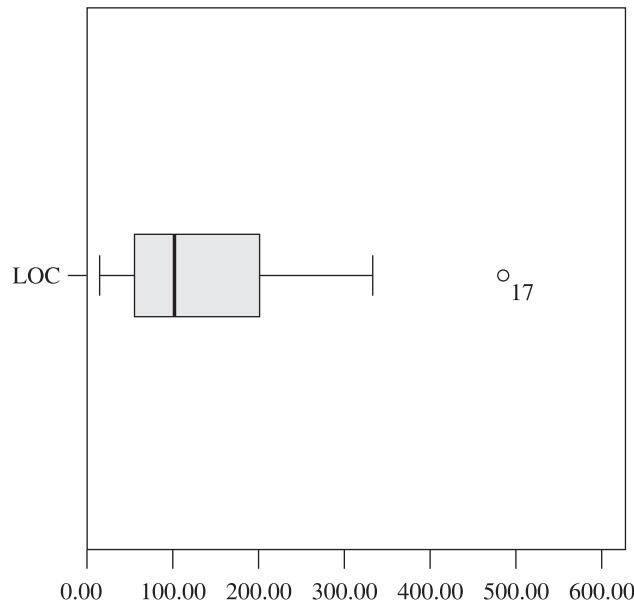


Figure 8.15 Box plot of LOC metric.

EXAMPLE 8.5 Consider the data set given in Example 8.3. Construct box plots and identify univariate outliers for all the variables in the data set.

Solution The box plots for LOC, fault count, cyclomatic complexity and branch count using univariate analysis are shown in Figures 8.16–8.19.

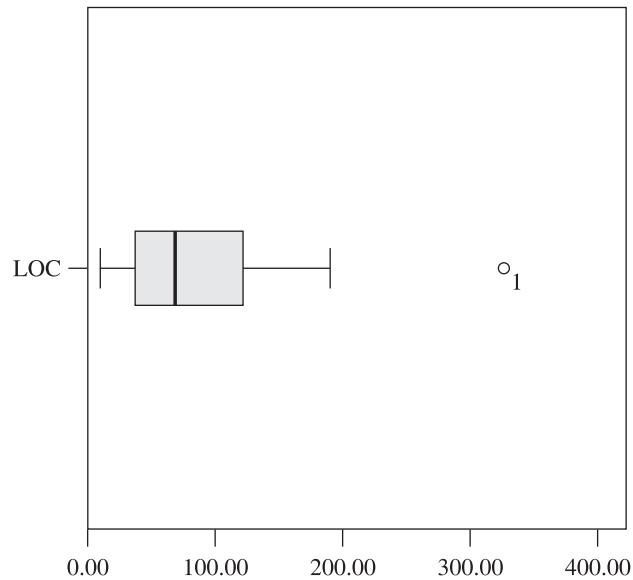


Figure 8.16 Box plot for LOC metric.

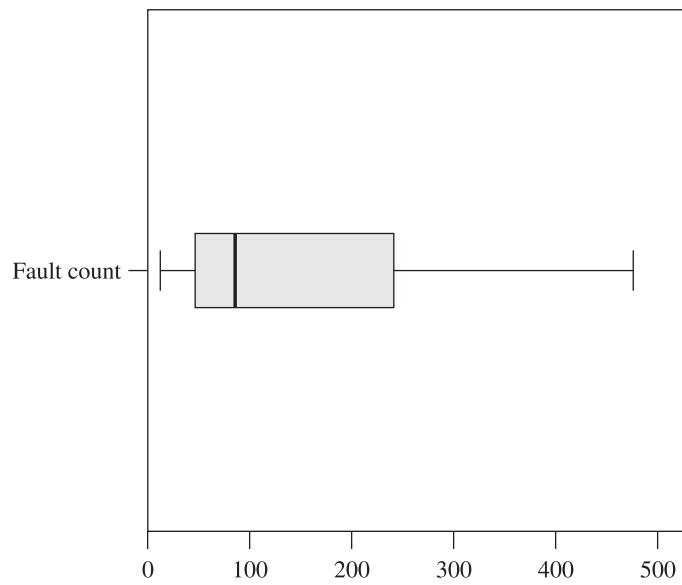


Figure 8.17 Box plot for fault count metric.

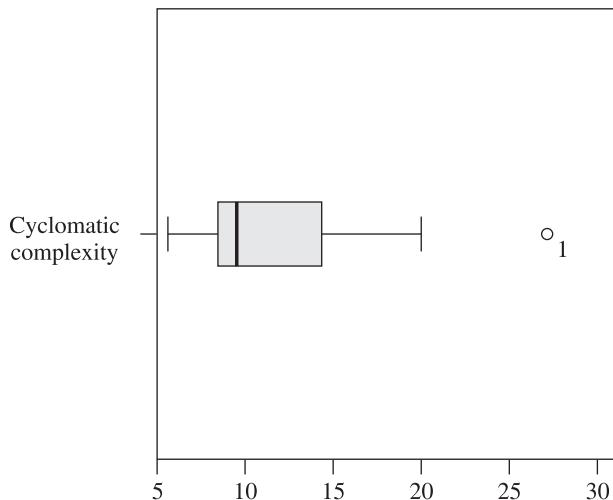


Figure 8.18 Box plot for cyclomatic complexity metric.

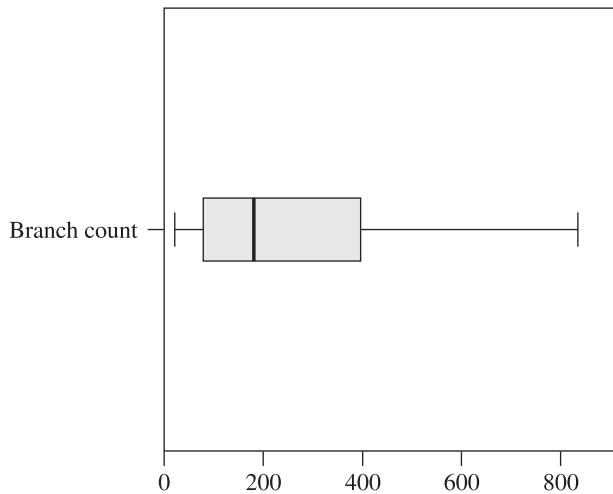


Figure 8.19 Box plot for branch count metric.

The outliers should be analysed by the researchers to make a decision about their inclusion or exclusion in the data analysis. There may be many reasons for an outlier: (i) error in the entry of the data, (ii) some extra information that represents extraordinary or unusual event and (iii) an extraordinary event that is unexplained by the researcher.

Outlier values may be present due to the combination of data values present across more than one variable. These outliers are called multivariate outliers. Scatter plot is another visualization method to detect outliers. In scatter plots, we simply represent graphically all the data points. The scatter plot allows us to examine more than one metric variable at a given time. The univariate outliers can also be determined by calculating the z -score value of the data points of the variable. The data values exceeding ± 2.5 are considered to be outliers (Hair et al., 2006).

EXAMPLE 8.6 Consider the data set given in Example 8.3. Identify bivariate outliers between dependent variable fault count and other variables.

Solution The scatter plots are shown in Figures 8.20–8.22.

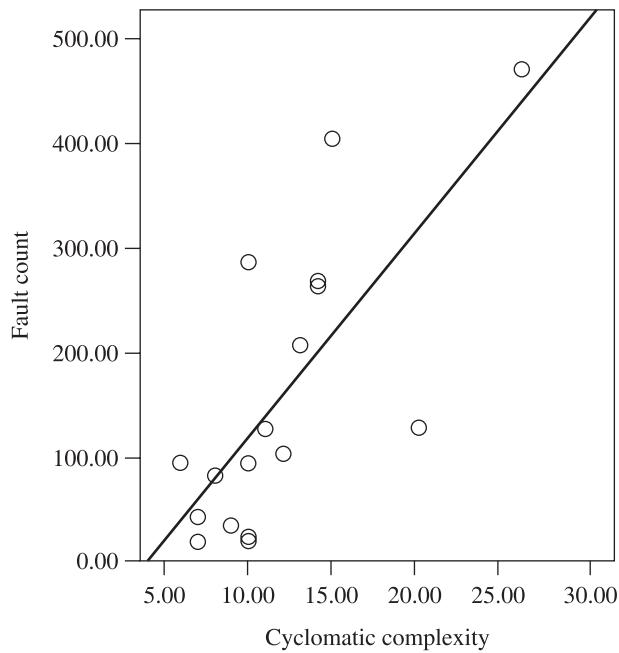


Figure 8.20 Scatter plot for cyclomatic complexity metric.

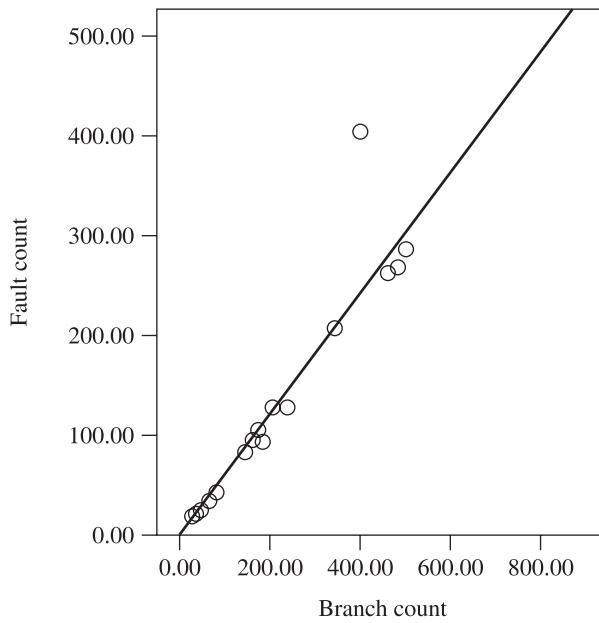


Figure 8.21 Scatter plot for branch count metric.

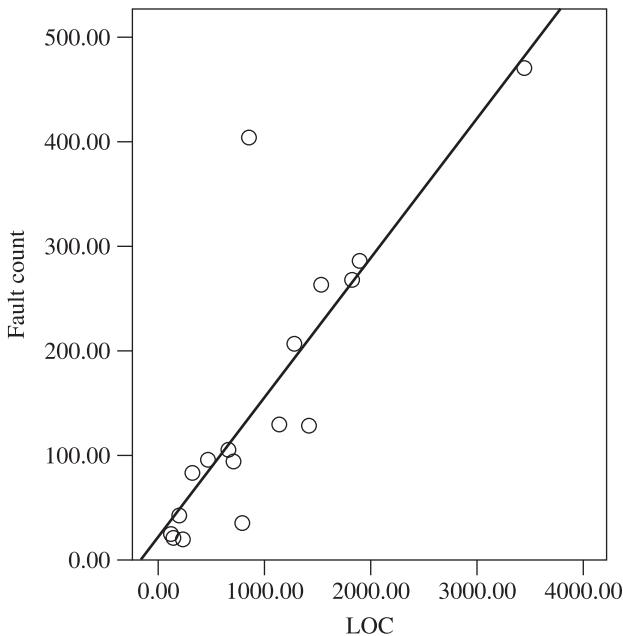


Figure 8.22 Scatter plot for LOC metric.

EXAMPLE 8.7 Consider the data set given in Example 8.3. Identify outliers by computing z -score values.

Solution After computing the z -score values for all the given metrics, the following case numbers are found to be influential (Table 8.10).

Table 8.10 Influential case numbers

Variable	Case number
Cyclomatic complexity	1
LOC	1
Fault count	None
Branch count	1

8.4.4 Correlation Analysis

Correlation analysis studies the variation of two or more variables for determining the amount of correlation between them. Hopkins (2003) calls a correlation coefficient value between 0.5 and 0.7 large, 0.7 and 0.9 very large, and 0.9 and 1.0 almost perfect. Correlation analysis can be used to find the relationship among the metrics. This tells whether the metrics are independent or contain redundant information.

8.4.5 Exploring Analysis

The metric variables can be of two types—Independent variables and dependent (target) variables. The effect of independent variables on the dependent variables can be explored by

using various statistical and machine learning methods. The choice of the method for analysing the relationship between independent and dependent variables depends upon the type of the dependent variable (continuous or discrete).

If the dependent variable is continuous, then the widely known statistical method (linear regression method) may be used, whereas if the dependent variable is of discrete type, then logistic regression method may be used for analysing relationships.

8.5 Metrics for Measuring Size and Structure

There are a range of metrics available to measure the size and structure of a software system. Size metrics can be used to estimate the size of the software, input to estimation models and can be used to monitor the progress during software development. The structural metrics helps us to analyse the product and increase the understanding of the product. They may also provide insight into the complexity of the software. In the below sections, we define size and structure metrics.

8.5.1 Size Estimation

The measurement of size is a very important and difficult area in software development. **Size metrics** give an indication about the length of the source code. The physical quantities such as mass, velocity or temperature are easily measurable, but measuring the size of the software is difficult. The most commonly used size metric is **lines of source code (LOC)**. There are many ways in which they can be counted. For example, many programs use comment lines and blank lines to make their programs more understandable and readable.

In true sense, programming effort is required in order to include blank lines and comment lines. Thus, these lines may be excluded from the count of lines of source code. Similarly, unexecutable statements are also not included by some programmers in the LOC count. Hence, the programmer must be careful while selecting the method for counting LOC. Consider the operation to check the validity of a triangle given in Figure 8.23.

```
/*This function checks for validity of a triangle*/
int triangle(int a, int b, int c)
{
    int valid;
    if((a+b)>c)&&((c+a)>b)&&((b+c)>a)
        //checks validity
    {
        valid=1;
    }
    else
    {
        valid=0;
    }
    return valid;
}
```

Figure 8.23 Operation to check validity of a triangle.

The function in Figure 8.23 consists of 15 LOC, if we simply count the number of LOC.

Most programmers and researchers exclude blank lines and comment lines as these lines do not consume any effort and only give illusion of high productivity of the staff that is measured in terms of LOC/PM (lines of source code/person month). The LOC count for the function shown in Figure 8.23 after excluding the blank and comment lines is 13. This value of LOC follows the definition of LOC given by Conte et al. (1986).

LOC can be used to provide estimation of programming time and also serves as input to many estimation models such as COCOMO, COCOMO II, etc. However, their usefulness is limited to other non-functional tasks such as reliability, efficiency, usability, etc. The number of classes is also an important count for measuring size of object-oriented software. The size metrics related to object-oriented software are described in Section 8.7.

The functional units can be counted in the early phases of software development. In object-oriented software development, the functionality of a software can be depicted in terms of use cases and classes. The use case point and class point method are used to count the functional units in object-oriented software development and have been explained in Chapter 4 (refer to Sections 4.3 and 4.4).

8.5.2 Halstead Software Science Metrics

Halstead developed a suite of software science measures in the early 1970s. These metrics see a program in terms of tokens that are classified as operators or operands. Any keyword or symbol in a program is considered as an operator, while any symbol used to represent data is considered as an operand. Variables, constants and labels are considered to be operands, whereas arithmetic operators such as +, -, /, * and keywords such as ‘int’, ‘printf’, ‘for’, ‘while’ and function names are counted as operators. Operators also include braces, parameters and even names of the operations (reserved keywords). The basic metrics to count the vocabulary of a program are given as follows:

$$\begin{aligned}\eta_1 &= \text{number of unique operators} \\ \eta_2 &= \text{number of unique operands} \\ \eta &= \text{vocabulary of a program} \\ \eta &= \eta_1 + \eta_2\end{aligned}$$

Program length is calculated in terms of the total number of tokens given as:

$$\begin{aligned}N_1 &= \text{total occurrences of operators} \\ N_2 &= \text{total occurrences of operands}\end{aligned}$$

$$\text{Length: } N = N_1 + N_2$$

Program length can also be expressed in terms of LOC as:

$$N = 2 \times \text{LOC}$$

The volume of the program is interpreted as the number of mental comparisons needed to write as the given program and is given as:

$$\text{Volume: } V = N \times \log_2 (n_1 + n_2)$$

The unit of measurement of volume is ‘bits’. A program can be implemented in various ways, potential volume (V^*) represents the program with minimal value. The program level is given as:

$$\text{Level: } L = V^*/V$$

The difficulty is the inverse of program level:

$$\text{Difficulty: } D = 1/L$$

The potential volume of an algorithm is implemented as a procedure call and is given below:

$$V^* = (2 + n_1^*) \log_2(2 + n_2^*)$$

The first term is 2 which represents two unique operators for the procedure call. The second term n_2^* represents the number of conceptually unique input and output parameters of a procedure.

The estimated program level and volume are given as:

$$\bar{L} = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

$$\text{Difficulty: } D = \frac{1}{\bar{L}}$$

$$= \frac{\eta_1 N_2}{2 \eta_2}$$

$$\text{Effort: } E = \frac{V}{\bar{L}}$$

$$\text{Program time: } T = \frac{E}{\beta}$$

β is normally equal to 18. The metrics proposed by Halstead can also be applied to object-oriented software.

EXAMPLE 8.8 Consider the function to check the validity of a triangle given in Figure 8.23. List out the operators and operands and also calculate the values of software science metrics.

Solution The list of operators and operands is given in Table 8.11.

Table 8.11 Tokens for the program given in Figure 8.23

Operators	Occurrences	Operands	Occurrences
int	4	triangle	1
{ }	3	a	4
if	1	b	4
+	3	c	4
>	3	valid	4
&&	2	1	1
=	2	0	1

(Contd.)

Table 8.11 Tokens for the program given in Figure 8.23 (*Contd.*)

Operators	Occurrences	Operands	Occurrences
else	1		
return	1		
;	4		
,	2		
()	7		
$\eta_1 = 12$	$N_1 = 33$	$\eta_2 = 7$	$N_2 = 19$

$$\begin{aligned}
 \eta &= \eta_1 + \eta_2 \\
 &= 12 + 7 = 19 \\
 N &= N_1 + N_2 \\
 &= 33 + 19 = 52 \\
 V &= N \times \log_2(n_1 + n_2) \\
 &= 52 \times \log_2 19 \\
 &= 52 \times 4.249 = 220.96 \text{ bits} \\
 N &= \eta_1 \log_2 n_1 + \eta_2 \log_2 n_2 \\
 &= 12 \log_2 12 + 7 \log_2 7 \\
 &= 43.02 + 19.65 = 62.67 \\
 V^* &= (2 + n_1^*) \log_2(2 + n_2^*) \\
 &= (2 + 4) \log_2(2 + 4) \\
 &= 6 \log_2 6 = 15.5 \\
 \bar{L} &= \frac{2}{n_1} \times \frac{n_2}{N_2} \\
 &= \frac{2}{12} \times \frac{7}{19} = 0.0614 \\
 \bar{D} &= \frac{1}{\bar{L}} = 16.29 \\
 E &= V \times \bar{D} \\
 &= 220.96 \times 16.29 = 3599.44 \\
 T &= \frac{E}{\beta} = \frac{3599.44}{18} = 199.97 \text{ s} \\
 &= 3.33 \text{ min}
 \end{aligned}$$

8.5.3 Information Flow Metrics

Information flow metrics represent the coupling (degree of interdependence between classes) in the system. A system is said to be highly coupled if its classes accept information from and/or pass information to other classes. The aim of the developer should be to minimize coupling, as high-coupled systems tend to be less reliable and maintainable. Fan-in and fan-out refer to the number of classes collaborating with each other and are defined as follows:

1. Fan-in: It counts the number of other classes that call a given class A.
2. Fan-out: It counts the number of other classes called by a given class A.

Figure 8.24 shows values of fan-in and fan-out for a small system consisting of six classes.

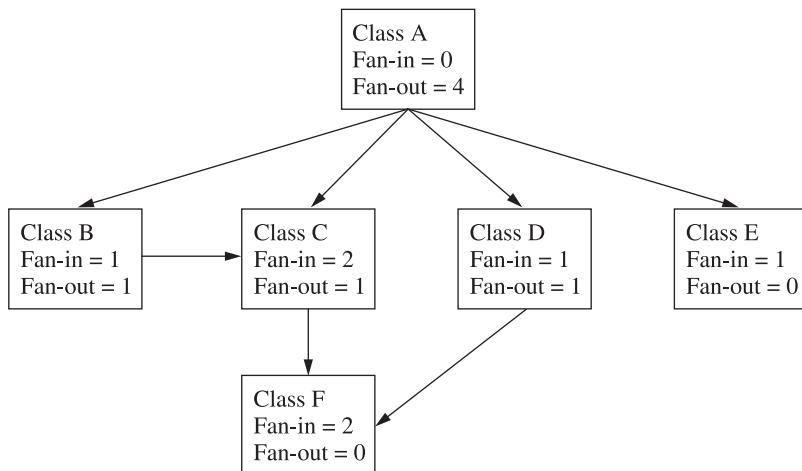


Figure 8.24 Fan-in and fan-out metrics for six systems.

A low fan-out value is desirable since high fan-out values represent large amount of coupling present in the system.

8.6 Measuring Software Quality

Software quality should be an essential practice in software development and thus arises the need of measuring the aspects of software quality. Measuring quality attributes will guide the software professionals about the quality of the software. Software quality must be measured throughout the software development life cycle phase.

8.6.1 Software Quality Metrics Based on Defects

According to the IEEE/ANSI standard, defect can be defined as “an accidental condition that causes a unit of the system to fail to function as required”. A fault can cause many failures; hence, there is no one-to-one correspondence between fault and a failure. The defect-based metrics can

be classified at product and process levels. The difference of these two terms fault and defect is unclear from the definitions. In practice, the difference in the terms is not significant and these terms are used interchangeably. The defect density and defect rate are two popular product metrics used for measuring defects.

Defect Density

Defect density can be measured as the ratio of the number of defects encountered to the size of the software. Size of the software is usually measured in terms of **thousands of lines of code (KLOC)** and is given as:

$$\text{Defect density} = \frac{\text{Number of defects}}{\text{KLOC}}$$

The number of defects counts the defects detected during testing or by using any verification technique.

Defect rate can be measured as the number of defects encountered over a period of time such as per year. The defect rate may be useful in predicting the cost and resources that will be utilized in maintenance phase of software development. Another useful metric is defect density during testing. It measures the defect density during the formal testing, i.e. after the source code has been formally completed and added in the software library. If the value of defect density metric during testing is high, then the tester should ask the following questions:

1. Is the software not well designed or developed?
2. Is the testing technique effective in detecting defects?

If the reason of higher defects is the first one, then the software should be tested thoroughly so that higher defects are revealed. If the reason of higher defects is the second one, then the quality perspective of the software is good.

Phase-based Defect Density

It is an extension of defect density metric. The defect density can be tracked at various phases of software development including verification activities such as reviews, inspections and formal reviews before the start of validation testing. This metric provides an insight into the processes and standards being used during software development. Some organizations even set ‘threshold values’ for these metrics in order to monitor and assess the quality of the software.

Defect Removal Effectiveness

Defect removal effectiveness (DRE) is defined as:

$$\text{DRE} = \frac{\text{Defects removed in a given life cycle phase}}{\text{Latent defects}}$$

Latent defects for a given phase is not known. Thus, they are estimated as the sum of defects removed during a phase and defects detected later. The higher the value of the metric, the more

efficient and effective is the process followed in a particular phase. The ideal value of DRE is 1. The DRE of a product can also be calculated by

$$\text{DRE} = \frac{D_B}{D_B + D_A}$$

where D_B is defects encountered before software delivery and D_A is defects encountered after delivery.

8.6.2 Usability Metrics

Usability metrics measure the ease of use, learnability and user satisfaction for a given software. Bevan (1995) measured usability under MUSIC project. In MUSIC project, a number of performance measures are proposed. The task effectiveness is defined as follows:

$$\text{Task effectiveness} = \frac{1}{100} \times (\text{quantity} \times \text{quality})\%$$

where quantity measures the amount of task completed by a user and quality measures the degree to which the output produced by the user satisfies the goals of the task.

Both quantity and quality are represented in percentages. For example, consider a task of proofreading a three-page document. The quantity specifies the percentage of ratio of the number of words proofread to the total number of words. The quality is calculated as the percentage of proportion of the correctly proofread document. If the quantity is 80% and the quality is 90%, then task effectiveness is 72%.

The other measures of use defined in MUSIC project are (Bevan, 1995):

$$\text{Temporal efficiency} = \frac{\text{Effectiveness}}{\text{Task time}}$$

$$\text{Productive period} = \frac{\text{Task time} - \text{Unproductive time}}{\text{Task time}} \times 100$$

$$\text{Relative user efficiency} = \frac{\text{User efficiency}}{\text{Expert efficiency}} \times 100$$

The other measures that can be used to measure the usability of the system are as follows:

1. Time required to learn a system
2. Total increase in productivity by the use of the system
3. Response time

Conducting a survey based on a questionnaire is an effective means to measure the satisfaction of the customer. The questionnaire should be developed by experience persons having technical knowledge. The sample size should be sufficient enough to build the confidence level on the survey results. The results rated on scale (for example, very satisfied, satisfied, neutral, dissatisfied, very dissatisfied) may be used for measuring satisfaction level of the customer.

The quality charts such as bar charts, scatter plots, line charts, etc. should be built to analyse the satisfaction level of the customer. The customer satisfaction must be continuously monitored over time.

8.6.3 Testing Metrics

The testing metrics can be used to monitor the status of testing and provides indication about the quality of the product.

Testing coverage metrics can be used to monitor the amount of testing being done. These include the following basic coverage metrics:

1. **Statement** coverage metric describes the degree to which statements are covered while testing.
2. **Branch** coverage metric determines whether each branch in the source code has been tested.
3. **Operation** coverage metric determines whether every operation of a class has been tested.
4. **Condition** coverage metric determines whether each condition is evaluated both for true and for false.
5. **Path** coverage metric determines whether each path of the control flow graph has been exercised or not.
6. **Loop** coverage metric determines how many times a loop is covered.
7. **Multiple condition** coverage metric determines whether every possible combination of conditions is covered.

NASA has given a test focus metric which determines the amount of effort spent in finding and fixing ‘real’ faults in the software versus the amount of effort spent in finding and fixing ‘fake’ faults in the software. The test focus (TF) metric is given as (Stark et al., 1992):

$$TF = \frac{\text{Number of STRs fixed and closed}}{\text{Total number of STRs}}$$

where STR is software trouble report.

The fault coverage metric (FCM) is given as:

$$FCM = \frac{\text{Number of faults addressed} \times \text{Severity of faults}}{\text{Total number of faults} \times \text{Severity of faults}}$$

Some of the basic process metrics are given below:

1. Number of test cases designed
2. Number of test cases executed
3. Number of test cases passed
4. Number of test cases failed
5. Test case execution time
6. Total execution time
7. Time spent for the development of a test case
8. Total time spent for the development of all test cases

On the basis of the above direct measures, we may design the following additional metrics so that more useful information can be gained from the basic metrics:

1. Percentage of test cases executed
2. Percentage of test cases passed
3. Percentage of test cases failed
4. Total actual execution time/total estimated execution time
5. Average execution time of a test case

8.7 Object-Oriented Metrics

Object-oriented systems are rapidly increasing in the market. Object-oriented software engineering leads to better design, higher quality and maintainable software. As the object-oriented development is growing, the need for object-oriented metrics that can be used across the software industry is also increasing. The traditional metrics, although applicable to object-oriented systems, do not measure object-oriented constructs such as inheritance and polymorphism. This need has led to the development of object-oriented metrics. In the following subsections, we define coupling, cohesion, inheritance, reuse and size metrics for object-oriented systems.

8.7.1 Coupling Metrics

The degree of interdependence between classes is defined by coupling. During object-oriented analysis and design phases, coupling is measured by counting the relationship a class has with other classes or systems. Coupling increases complexity and decreases maintainability, reusability and understandability. Hence, coupling should be reduced amongst classes and the classes should be designed with the aim of weak coupling. The developer's goal should be to eliminate extraneous coupling (Henderson-Sellers, 1996). Chidamber and Kemerer (1994) defined coupling as:

Two classes are coupled when methods declared in one class use methods or instance variables of the other classes.

This definition also includes coupling based on inheritance. In 1994, Chidamber and Kemerer defined coupling between objects (CBO). In their paper, they defined CBO as the count of the number of other classes to which a class is coupled. The CBO definition given in 1994 includes inheritance-based coupling. For example, consider Figure 8.25. Two variables of other classes (class B and class C) are used in class A; hence, the value of CBO for class A is 2. Similarly, for class B and class C, the value of CBO is zero.

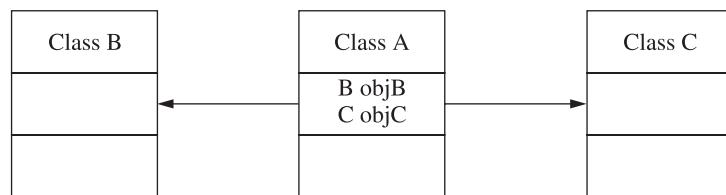


Figure 8.25 Values of CBO metric for a small program.

Li and Henry (1993) used the data abstraction technique for defining coupling. Data abstraction provides the ability to create user-defined data types called abstract data types (ADTs). Li and Henry defined data abstraction coupling (DAC) as:

$$\text{DAC} = \text{Number of ADTs defined in a class}$$

In Figure 8.25, class A has two ADTs (i.e. two non-simple attributes) objB and objC. Li and Henry defined another coupling metric known as message passing coupling (MPC) as “the number of unique sent statements in a class”. Hence, if four different methods in class A access the same method in class B, then MPC is 4 as shown in Figure 8.26.

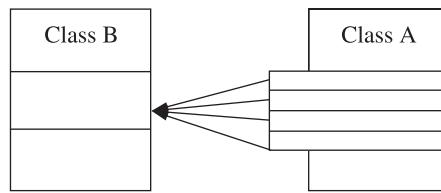


Figure 8.26 Example of MPC metric.

Chidamber and Kemerer (1994) defined response for a class (RFC) metric as a set of methods defined in a class and called by a class. It is given by $\text{RFC} = | \text{RS} |$, where RS, the response set of the class, is given by:

$$\text{RS} = M_i \cup \text{all } j\{\text{R}_{ij}\}$$

where M_i = set of all methods in a class (total n) and $\text{R}_i = \{\text{R}_{ij}\}$ = set of methods called by M_i . For example, in Figure 8.27, the RFC value for class A is 9 and the method for calculating it is explained in the given example.

Class consists of four functions: A_1, A_2, A_3, A_4 A_1 calls B_1 and B_2 of class B A_2 calls C_1 of class C A_3 calls D_1 and D_2 of class D Thus, $\text{RFC} = \text{RS} $ $M_i = \{A_1, A_2, A_3, A_4\}$ $R_{ij} = \{B_1, B_2, C_1, D_1, D_2\}$ $\text{RS} = \{A_1, A_2, A_3, A_4, B_1, B_2, C_1, D_1, D_2\}$ $\text{RFC} = 9$

Figure 8.27 Example for RFC metric.

A system level metric, coupling factor, is also defined by Harrison et al. (1998). In 1997, Briand et al. (1997) gave a suite of 18 metrics that measured different types of interaction between classes. These metrics may be used to guide software developers about which type of coupling affects maintenance cost and reduces reusability. Briand et al. (1997) observed that the coupling between classes can be divided into different facets:

1. *Relationship:* It refers to the type of relationship between classes—friendship, inheritance, or other.
2. *Export or import coupling:* This identifies whether class A uses methods/variables of other class B (import).
3. *Type of interaction:* Briand et al. identified three types of interaction between classes—class-attribute, class-method and method-method.
 - (i) *Class-attribute (CA) interaction:* For example, consider Figure 8.28, class A consists of two non-simple attributes objA and objB of type class B and class C, respectively. Thus, if there is any change in class B and/or class C, class A will be affected.
 - (ii) *Class-method (CM) interaction:* If the parameter of class B is passed as argument to method of class A, the type of interaction is said to be class-method. For example, consider class A given in Figure 8.28. There is a CM interaction between method A1 and class B as method A1 uses the object of class B as parameter.
 - (iii) *Method-method (MM) interaction:* If a method M_i of class C_i calls method M_j of class C_j or method M_i of class C_i consists of reference of method M_j of class C_j as arguments, then there is MM type of interaction between class C_i and class C_j . For example, as shown in Figure 8.28, the method of class B calls method A1 of class A; hence, there is an MM interaction between class B and class A.

```

class A
{
B objB;
C objC;
public:
void A1(B obj1)
{
}
};
class B
{
public:
void B1()
{
A objA
A1();
}
};
class C
{
void A2(B::B1)
{
}
};

```

Figure 8.28 Example for computing type of interaction.

The metrics for CA are FCAIC, ACSIC, OCAIC, FCAEC, DCAEC and OCEEC. In these metrics, the first one/two letters signify the type of relationship (IF signifies inverse friendship, A signifies ancestors, D signifies descendant, F signifies friendship, and O signifies others). The next two letters signify the type of interaction (CA, CM, MM). Finally, the last two letters signify import coupling (IC) or export coupling (EC).

Lee et al. (1995) differentiate between inheritance-based and non-inheritance-based coupling by the corresponding measures: Non-inheritance information flow-based coupling (NIH-ICP), and information flow-based inheritance coupling (IH-ICP). Information flow-based coupling (ICP) metric was the sum of NIH-ICP and IH-ICP metrics. Lee et al. emphasized that their ICP metric, based on method invocation, take polymorphism into account.

8.7.2 Cohesion Metrics

Cohesion is a **desirable property of a class and should be maximized as it supports the concept of data hiding**. Low cohesive class is more complex and is more prone to faults in the development life cycle. Chidamber and Kemerer (1994) proposed **lack of cohesion in methods (LCOM)** metric in 1994. The LCOM metric measures the dissimilarity of methods in a class by finding the attributes used by the methods.

It calculates the difference between the number of methods that have similarity zero and the number of methods that have similarity greater than zero. If the difference is negative, then the value of LCOM is zero. The larger the similarity between methods, the more cohesive is the class. For example, a class consists of four attributes (a_1, a_2, a_3 and a_4). The method usage of the class is given in Figure 8.29.

$M_1 = \{a_1, a_2, a_3, a_4\}$ $M_2 = \{a_1, a_2\}$ $M_3 = \{a_3\}$ $M_4 = \{a_3, a_4\}$ $M_1 \cap M_2 = 1$ $M_1 \cap M_3 = 1$ $M_1 \cap M_4 = 1$ $M_2 \cap M_3 = 0$ $M_2 \cap M_4 = 0$ $M_3 \cap M_4 = 1$ $LCOM = 2 - 4$, hence, $LCOM = 0$

Figure 8.29 Example of LCOM metric.

Henderson-Sellers (1996) observed some problems in the definition of LCOM metric proposed by Chidamber and Kemerer in 1994. The problems were as follows:

1. A number of real examples showed the value of LCOM as zero due to the presence of dissimilarity amongst methods. Hence, while a large value of LCOM suggests low cohesion, the zero value does not necessarily suggest high cohesion.
2. No guideline for interpretation of the value of LCOM was given by Chidamber and Kemerer. Thus, Henderson-Sellers revised the LCOM value. Consider m methods

accessing a set of attributes D_i ($i = 1, \dots, n$). Let the number of methods that access each datum be $\mu(D_i)$. The revised LCOM1 metric is given as:

$$\text{LCOM1} = \frac{\frac{1}{N} \sum_{i=1}^n \mu(D_i) - m}{1 - m}$$

Beiman and Kang (1995) defined two cohesion metrics: **tight class cohesion (TCC)** and **loose class cohesion (LCC)**. The TCC metric is defined as the percentage of pairs of directly connected public methods of the class with common attribute usage. The LCC metric is the same as TCC, except that it also considers indirectly connected methods. A method M_1 is said to be indirectly connected with method M_3 , if M_1 is connected to method M_2 and method M_2 is connected to method M_3 . Hence, indirectly connected methods represent transitive closure of directly connected methods. Consider the source code of a queue class given in Figure 8.30 (Beiman and Kang, 1995).

```
class queue
{
private:
int *a;
int rear;
int front;
int n;
public:
queue(int s)
{
n=s;
rear=0;
front=0;
a=new int[n];
}
int empty()
{
if(rear==0)
{
    return 1;
}
else
{
    return 0;
}
}
void insert(int);
int remove();
int getsize();
```

Figure 8.30 (Contd.)

```

{
    return n;
}
void display();
};
void queue::insert(int data)
{
    if(rear==n)
    {
        cout<<"Queue overflow";
    }
    else
    {
        a[rear++]=data;
    }
}
int queue::remove()
{
    int element,i;
    if(empty())
    {
        cout<<"Queue underflow";
        getch();
    }
    else
    {
        element=a[front];
        for(i=0;i<rear-1;i++)
        {
            a[i]=a[i+1];
        }
        rear--;
    }
    return element;
}
void queue::display()
{
    int i;
    for(i=0;i<rear;i++)
    {
        cout<<a[i]<<" ";
    }
    getch();
}

```

Figure 8.30 Source code for implementing queue class.

Figure 8.31 shows the attribute usage of methods. The pair of public functions with common attribute usage is given below:

{(empty, insert), (empty, remove), (empty, display), (getsize, insert), (insert, remove),
(insert, display), (remove, display)}

Thus,

$$\text{TCC}(\text{Queue}) = \frac{7}{10} \times 100 = 70\%$$

The methods empty and getsize are indirectly connected, since empty is connected to insert and getsize is also connected to insert. Thus, by transitivity, empty is connected to getsize. Thus,

$$\text{LCC}(\text{Queue}) = \frac{10}{10} \times 100 = 100\%$$

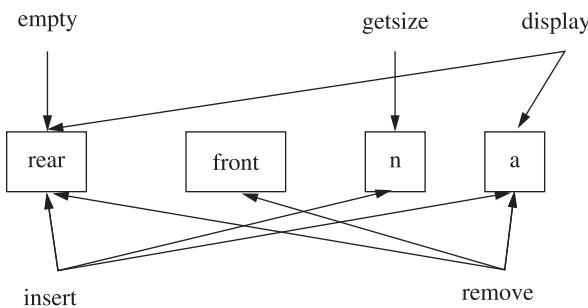


Figure 8.31 Attribute usage of methods of class queue.

Lee et al. (1995) proposed information flow-based cohesion (ICH) metric. ICH for a class is defined as the weighted sum of the number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method. In Figure 8.31, the method remove invokes the method empty which does not consist of any arguments. Thus, $\text{ICH}(\text{Queue}) = 1$.

8.7.3 Inheritance Metrics

The inheritance is measured in terms of **depth of inheritance** hierarchy by many authors in the literature. The depth of a class within an inheritance hierarchy is measured by depth of inheritance tree (DIT) metric given by Chidamber and Kemerer (1994). It is measured as the **number of steps from the class node to the root node of the tree**. In the case of involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree. Consider Figure 8.32. The value of the DIT metric for class D is 2.

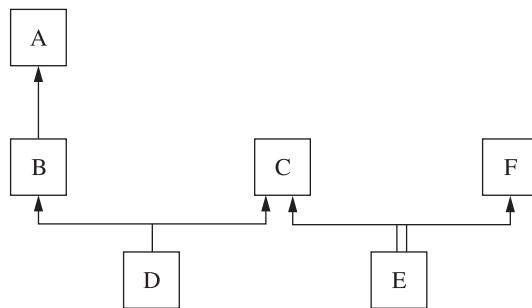


Figure 8.32 Inheritance hierarchy.

The average inheritance depth (AID) is calculated as (Yap and Henderson-Sellers, 1993):

$$AID = \frac{\sum \text{Depth of each class}}{\text{Total number of classes}}$$

In Figure 8.32, the depth of subclass D is 1.5 $[(2 + 1)/2]$.

The AID of overall inheritance structure is: $0(A) + 1(B) + 0(C) + D(1.5) + E(1) + 0(F) = 3.5$. Finally, dividing by the total number of classes, we get $3.5/6 = 0.58$.

Number of children (NOC) metric counts the number of immediate subclasses of a class in a hierarchy. In Figure 8.32, the NOC value for class A is 1 and for class E it is 2. Lorenz and Kidd developed number of parents (NOP) metric which counts the number of classes that a class directly inherits (i.e. multiple inheritance) and number of descendants (NOD) as the number of subclasses of a class (both directly and indirectly). Number of ancestors (NOA) given by Tegarden and Sheetz (1992) counts the number of base classes of a class (both directly and indirectly). Hence, in Figure 8.32, NOA(D) = 3 (A, B, C), NOP(D) = 2 (B, C) and NOD(A) = 2 (B, D).

Lorenz and Kidd (1994) gave three measures: number of methods overridden (NMO), number of methods added (NMA) and number of methods inherited (NMI). When a method in a subclass has the same name and type (signature) as in the superclass, then the method in the superclass is said to be overridden by the method in the subclass. NMA counts the number of new methods (neither overridden nor inherited) added in a class. NMI counts the number of methods a class inherits from its superclasses. Finally, Lorenz and Kidd use NMO, NMA and NMI metrics to calculate specialization index (SIX) as given below:

$$SIX = \frac{NMO \times DIT}{NMO + NMA + NMI}$$

Consider the source code given in Figure 8.33. The class student overrides two methods of class person—readperson() and displayperson(). Thus, the value of the NMO metric for class student is two. One new method is added in this class (getaverage). Hence, the value of NMA metric is 1.

Similarly, for class GradStudent, the value of NMO is 2, NMA is 1 and NMI is 1 (getaverage()). The value of SIX for class GradStudent is given by

$$SIX = \frac{2 \times 2}{2 + 1 + 1} = \frac{4}{4} = 1$$

The maximum number of levels in the inheritance hierarchy which is below the class is measured through class to leaf depth (CLD). The value of CLD for class person is 2.

8.7.4 Reuse Metrics

An object-oriented development environment supports design and code reuse, the most straightforward type of reuse being the use of a library class (of code), which perfectly suits the requirements. Yap and Henderson-Sellers (1993) discuss two measures designed to evaluate the level of reuse possible within classes.

```

class Person {
protected:
char name[25];
int age;
public:
void readperson();
void displayperson();
};

class Student extends Person{
protected:
roll_no[10];
float average;
public:
void readstudent();
void displaystudent();
float getaverage();
};

class GradStudent extends Student{
private:
char subject[25];
char working[25];
public:
void readit();
void displaysubject();
void workstatus();
};

```

Figure 8.33 Source code to demonstrate use of inheritance.

The reuse ratio (U) is given by

$$U = \frac{\text{Number of superclasses}}{\text{Total number of classes}}$$

Consider Figure 8.33, the value of U is $2/3$. Another metric is specialization ratio (S), which is given by

$$S = \frac{\text{Number of subclasses}}{\text{Number of superclasses}}$$

In Figure 8.33, number of subclasses = {Student, GradStudent} and number of superclasses = {Person, Student}. Thus, $S = 1$.

Aggarwal et al. (2009) proposed another set of metrics for measuring reuse by using generic programming in the form of templates. The metric function template factor (FTF) is defined as the ratio of the number of functions using function templates to the total number of functions as shown below:

$$\text{FTF} = \frac{\text{Number of functions using function templates}}{\text{Total number of functions}}$$

Consider a system, with methods F_1, \dots, F_n . Then,

$$\text{FTF} = \frac{\sum_{i=1}^n \text{uses_FT}(F_i)}{\sum_{i=1}^n F_i}$$

where, $\text{uses_FT}(F_i) = \begin{cases} 1, & \text{iff function uses function template} \\ 0, & \text{otherwise} \end{cases}$

```
void function1(){
....}
template<class T>
void function2(T &x, T &y){
....}
void function3(){
....}
```

Figure 8.34 Source code for calculating metric FTF.

In Figure 8.34, the value of metric FTF = 1/3.

The metric class template factor (CTF) is defined as the ratio of the number of classes using class templates to the total number of classes as shown below:

$$\text{CTF} = \frac{\text{Number of classes using class templates}}{\text{Total number of classes}}$$

Consider a system, with classes C_1, \dots, C_n . Then,

$$\text{CTF} = \frac{\sum_{i=1}^n \text{uses_CT}(C_i)}{\sum_{i=1}^n C_i}$$

where, $\text{uses_CT}(C_i) = \begin{cases} 1, & \text{iff class uses class template} \\ 0, & \text{otherwise} \end{cases}$

```
class A{
....};
template<class T, int size>
class B{
T arr[size];
....};
```

Figure 8.35 Source code for calculating metric CTF.

In Figure 8.35, the value of metric CTF = 1/2.

8.7.5 Size Metrics

Several traditional metrics are applicable to object-oriented systems. The traditional LOC metric is a measure of the size of a class (refer to Section 8.3.1). Halstead's software science and McCabe's measures for measuring size are also applicable to object-oriented systems; however, the object-oriented paradigm defines a different way of doing things. This has led to the development of size metrics applicable to object-oriented constructs. Chidamber and Kemerer defined weighted methods per class (WMC) metric given by:

$$WMC = \sum_{i=1}^n C_i$$

where M_1, \dots, M_n are methods defined in class K_1 and C_1, \dots, C_n are the complexity of the methods.

The number of attributes (NOA), given by Lorenz and Kidd, is defined as the sum of the number of instance variables and number of class variables. Number of methods (NOM) given by Li and Henry (1993) is defined as the number of local methods defined in a class. They also gave two additional size metrics SIZE1 and SIZE2 besides the LOC metric given as:

$$SIZE1 = \text{number of semicolons in a class}$$

$$SIZE2 = NOA + NOM$$

8.7.6 Popular Metric Suites

The number of recognized metric suites is limited in the literature. The object-oriented metrics that have widely gained attention are summarized as follows:

1. Metric suite consisting of six metrics is proposed by Chidamber and Kemerer (1994). This metric suite has received widest attention in the literature. The metric suite along with the constructs it measures is summarized in Table 8.12.

Table 8.12 Chidamber and Kemerer metrics

Metric	Definition	Construct being measured
CBO	Coupling between objects	Coupling
WMC	Weighted methods per class	Size
RFC	Response for a class	Coupling
LCOM	Lack of cohesion in methods	Cohesion
NOC	Number of children	Inheritance
DIT	Depth of inheritance	Inheritance

2. Li and Henry (1993) investigated Chidamber and Kemerer metrics and proposed a suite of five metrics. These metrics are summarized in Table 8.13.

Table 8.13 Li and Henry metric suite

Metric	Definition	Construct being measured
DAC	Data abstraction coupling	Coupling
MPC	Message passing coupling	
NOM	Number of methods	Size
SIZE1	Number of semicolons	
SIZE2	Number of attributes + NOM	

3. Beiman and Kang (1995) proposed two cohesion metrics LCC and TCC.
4. Lorenz and Kidd (1994) proposed a suite of 11 metrics. These metrics address size, coupling, inheritance, etc. and are summarized in Table 8.14.

Table 8.14 Lorenz and Kidd metric suite

Metric	Definition	Construct being measured
NOP	Number of parents	Inheritance
NOD	Number of descendants	
NMO	Number of methods overridden	
NMI	Number of methods inherited	
NMA	Number of methods added	
SIX	Specialization index	

5. Briand et al. (1997) proposed a suite of 18 coupling metrics. These metrics are summarized in Table 8.15.

Table 8.15 Briand et al. metric suite

IFCAIC	These coupling metrics count the number of interactions between classes.
ACAIC	The metrics distinguish the relationship between the classes (friendship, inheritance, none), different types of interactions, and the locus of impact of the interaction.
OCAIC	
FCAEC	The acronyms for the metrics indicate what interactions are counted:
DCAEC	• The first or first two characters indicate the type of coupling relationship between classes [A: Ancestor, D: Descendants, F: Friend classes, IF: Inverse Friends (classes that declare a given class a as their friend), O: Others, i.e. none of the other relationships].
OCAEC	• The next two characters indicate the type of interaction:
IFCMIC	CA: There is a Class-Attribute interaction if class x has an attribute of type class y.
ACMIC	CM: There is a Class-Method interaction if class x consists of a method that has parameter of type class y.
DCMIC	MM: There is a Method-Method interaction if class x calls method of another class y, or class x has a method of class y as a parameter.
FCMEC	• The last two characters indicate the locus of impact:
DCMEC	IC: Import coupling, counts the number of other classes called by class x.
OCMEC	EC: Export coupling, counts the number of other classes using class y.
IFMMIC	
AMMIC	
OMMIC	
FMMEC	
DMMEC	
OMMEC	

6. Teagarden and Sheetz have proposed a large suite of metrics.
7. Lee et al. (1995) have given four metrics: one for measuring cohesion and three metrics for measuring coupling (see Table 8.16).

Table 8.16 Lee et al. metric suite

Metric	Definition	Construct being measured
ICP	Information flow-based coupling	Coupling
IHICP	Information flow-based inheritance coupling	
NIHICP	Information flow-based non-inheritance coupling	
ICH	Information based cohesion	Cohesion

8. The system level polymorphism metrics are measured by Benlarbi and Melo (1999). These metrics are used to measure static and dynamic polymorphism and are summarized in Table 8.17.

Table 8.17 Polymorphism metrics

Metric	Definition
SPA	Static polymorphism in ancestors
DPA	Dynamic polymorphism in ancestors
SP	Static polymorphism in inheritance relations; $SP = SPA + SPD$
DP	Dynamic polymorphism in inheritance relations; $DP = DPA + DPD$
NIP	Polymorphism in non-inheritance relations
OVO	Overloading in stand-alone classes
SPD	Static polymorphism in descendants
DPD	Dynamic polymorphism in descendants

9. Yap and Henderson-Sellers (1993) have proposed a suite of metrics to measure cohesion and reuse in object-oriented systems.
10. Aggarwal et al. (2006) have proposed a set of two metrics (FTF and CTF) to measure reuse in object-oriented systems.

Review Questions

1. What is software quality? Explain the various software quality attributes.
2. Define the following software quality attributes:
 - (a) Functionality
 - (b) Reliability
 - (c) Maintainability
 - (d) Adaptability
3. Establish the relationship between reliability and maintainability. How does one affect the effectiveness of the other?

4. Describe the elements of a quality system. Why should customer's satisfaction be the prime concern for developers?
5. Design the table of associations between elements of a quality system and life cycle phases. Which life cycle phases require all elements of a quality system and why?
6. Explain the McCall's software quality model. Discuss the significance and importance of product quality factors.
7. Establish the relationship between quality factors and quality criteria in McCall's software quality model.
8. What is defect management and trend analysis of a quality system? Explain with the help of a format for software defect report.
9. Define the following terms:
 - (a) Configuration management
 - (b) Risk management activities
 - (c) Reviews and audits
 - (d) Standards, processes and metrics
10. Explain the Boehm's software quality model with software quality attributes in the form of a hierarchy.
11. What is ISO 9000? List the processes which can be improved and better maintained by the implementation of ISO 9000.
12. Discuss ISO 9126. How is it different from ISO 9000? Highlight a few characteristics which are part of this model.
13. What is CMM? How does it encourage the continuous improvement of the software processes?
14. Explain the various KPAs at different maturity levels. List some of the difficulties of achieving CMM level 5.
15. Compare CMM and ISO 9126 models. Which is more popular and why?
16. What are software metrics? Discuss their areas of applications.
17. What are the various categories of software metrics? Explain each category with the help of examples.
18. Define nominal scale and ordinal scale in the measurements and also discuss both the concepts with examples.
19. Explain the role and significance of Weyuker's properties in software metrics.
20. Describe the measures of central tendency. Discuss the concepts with examples.
21. Consider the following data set consisting of lines of source code for a given project:
100, 150, 156, 187, 206, 209, 235, 270, 275, 280, 305, 309, 351, 359, 375, 400, 406, 410, 450, 475, 500
Calculate mean, median and mode.

22. Describe the measures of dispersion. Explain the concepts with examples.
23. Consider the data set consisting of lines of source code given in Question 21. Calculate the standard deviation, variance and quartile.
24. Consider the following data sets given for three metric variables. Determine the normality of these variables.

Fault count	Cyclomatic complexity	Branch count
302	21	605
278	20	570
206	16	210
110	14	98
98	12	91
90	11	86
65	11	82
36	10	40
18	8	36
11	9	32

25. What is outlier analysis? Discuss its importance in data analysis. Explain univariate, bivariate and multivariate.
26. Consider the data set given in Question 24. Construct box plots and identify univariate outliers for all variables in the data set.
27. Consider the data set given in Question 24. Identify bivariate outliers between the dependent variable fault count and other variables.
28. Explain the various size estimation metrics. Which one is most popular and why?
29. Explain Halstead's software science metrics. How can we estimate program length? Highlight some advantages of these metrics.
30. Define the role of fan-in and fan-out in information flow metrics.
31. What are the various measures of software quality? Discuss any one with examples.
32. What are the various usability metrics? What is the role of customer satisfaction?
33. Define the following testing metrics:
 - (a) Statement coverage metric
 - (b) Branch coverage metric
 - (c) Condition and path coverage metrics
 - (d) Fault coverage metric
34. What is software coupling? Explain some coupling metrics with examples.
35. What is software cohesion? Explain some cohesion metrics with examples.
36. How do we measure inheritance? Explain inheritance metrics with examples.

- 37.** Define the following inheritance metrics:
- DIT
 - AID
 - NOC
 - NOP
 - NOD
 - NOA
- 38.** Define the following:
- Reuse ratio
 - Specialization ratio
 - FTF and CTF
- 39.** Define the following size metrics:
- Weighted methods per class (WMC)
 - Number of attributes (NOA)
 - Number of methods (NOM)
- 40.** Explain the metric suite of Chidamber and Kemerer. Highlight the strengths and weaknesses of this metric suite.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

- A fault is nothing but a/an:

(a) Error	(b) Defect
(c) Mistake	(d) All of the above
- A fault may generate:

(a) One failure	(b) Two failures
(c) Many failures	(d) All of the above
- In terms of software quality, the software should conform to

(a) Functional requirements	(b) Non-functional requirements
(c) Both (a) and (b)	(d) Critical requirements only
- Which is **not** a software quality attribute?

(a) Functionality	(b) Accountability
(c) Maintainability	(d) Reliability
- Which is **not** an element of a quality system?

(a) Processes, metrics and standards	(b) Quality of developers
(c) Reviews and audits	(d) Safety and security
- ANSI stands for:

(a) American National Standards Institute	(b) American National System Information
(c) American National Software Institute	(d) American National Safety Institute

7. McCall's software quality model consists of:

 - (a) Product operation
 - (b) Product transition
 - (c) Product revision
 - (d) All of the above

8. Product operation may **not** include:

 - (a) Correctness
 - (b) Efficiency
 - (c) Portability
 - (d) Reliability

9. Product revision may include:

 - (a) Maintainability
 - (b) Flexibility
 - (c) Testability
 - (d) All of the above

10. Product transition may include:

 - (a) Portability
 - (b) Reusability
 - (c) Interoperability
 - (d) All of the above

11. McCall has designed a:

 - (a) Quality model
 - (b) Design model
 - (c) Test model
 - (d) Maintenance model

12. Intermediate constructs of Boehm's software quality model may include:

 - (a) Portability
 - (b) Reliability
 - (c) Testability
 - (d) All of the above

13. How many levels of characteristics are designed in Boehm's software quality model?

 - (a) 1
 - (b) 2
 - (c) 3
 - (d) 4

14. Which is **not** a characteristic in ISO 9126?

 - (a) Safety
 - (b) Portability
 - (c) Reliability
 - (d) Usability

15. The total number of maturity levels in CMM is:

 - (a) 1
 - (b) 3
 - (c) 5
 - (d) 7

16. CMM stands for:

 - (a) Cost maturity model
 - (b) Capability maturity model
 - (c) Confident maturity model
 - (d) Condition maturity model

17. Characteristic of CMM level 5 (optimizing) is:

 - (a) Process control
 - (b) Process measurement
 - (c) Process definition
 - (d) Process management

18. KPA in CMM stands for:

 - (a) Key product areas
 - (b) Key process areas
 - (c) Key program areas
 - (d) Key person areas

19. Measures of central tendency include:

 - (a) Mean
 - (b) Median
 - (c) Mode
 - (d) All of the above

20. Measures of dispersion include:

- (a) Standard deviation
- (b) Variance
- (c) Quartiles
- (d) All of the above

21. Which one is **not** a measure of size?

- (a) LOC
- (b) Halstead's program length
- (c) Cyclomatic complexity
- (d) Function point

22. After the design of SRS, we may like to estimate:

- (a) Cost
- (b) Development time
- (c) Size
- (d) All of the above

23. Defect density is defined as:

- | | |
|--|--|
| (a) $\frac{\text{Number of defects}}{\text{KLOC}}$ | (b) $\frac{\text{Number of failures}}{\text{KLOC}}$ |
| (c) $\frac{\text{Number of undefined variables}}{\text{KLOC}}$ | (d) $\frac{\text{Number of variables}}{\text{KLOC}}$ |

24. DRE is calculated by:

- | | |
|-----------------------------|-----------------------------|
| (a) $\frac{D_A}{D_B + D_A}$ | (b) $\frac{D_B}{D_B + D_A}$ |
| (c) $\frac{D_B + D_A}{D_B}$ | (d) $\frac{D_B + D_A}{D_A}$ |

25. Reuse ratio is calculated by:

- | | |
|--|--|
| (a) $\frac{\text{Number of superclasses}}{\text{Total number of classes}}$ | (b) $\frac{\text{Total number of classes}}{\text{Number of superclasses}}$ |
| (c) $\frac{\text{Number of subclasses}}{\text{Total number of classes}}$ | (d) All of the above |

26. Specialization ratio is calculated by:

- | | |
|--|--|
| (a) $\frac{\text{Number of superclasses}}{\text{Number of subclasses}}$ | (b) $\frac{\text{Number of subclasses}}{\text{Number of superclasses}}$ |
| (c) $\frac{\text{Number of subclasses}}{\text{Total number of classes}}$ | (d) $\frac{\text{Number of superclasses}}{\text{Total number of classes}}$ |

27. Which one is **not** a Chidamber and Kemerer metric?

- (a) CBO
- (b) WMC
- (c) NOP
- (d) DIT

28. Which one is **not** a Lorenz and Kidd metric?

- (a) NOD
- (b) NMO
- (c) NMI
- (d) DIT

29. Select the best maturity level in CMM:

 - (a) CMM level 1
 - (b) CMM level 2
 - (c) CMM level 3
 - (d) CMM level 5

30. The CMM concept is designed at:

 - (a) Harvard University
 - (b) Carnegie Mellon University
 - (c) City University
 - (d) Oxford University

Further Reading

Hortch presents full account of the element of software quality system:

Horch, John W., *Practical Guide to Software Quality Management*. Norwood, MA: Artech House, 2003.

An in-depth study of 18 different categories of software complexity metrics was provided by Zuse, where he tried to give the basic definition of metrics in each category:

Zuse, H., *Software Complexity: Measures and Methods*. Berlin: Walter De Gruyter, 1990.

Fenton and Pfleeger's book is a classic and useful reference, and it gives detailed discussion on measurement and key definition of metrics:

Fenton, N. and Pfleeger, S., *Software Metrics: A Rigorous & Practical Approach*. Boston: PWS Publishing Company, 1997.

A detailed description on software reliability and contributions from many of the leading researchers may be found in Lyu's book:

Lyu, M., *Handbook of Software Reliability Engineering*. Los Angeles: IEEE Computer Press, 1996.

Aggarwal presents a good overview of software reliability models. Musa et al. provides a detailed description particularly on software reliability models:

Aggarwal, K.K., *Reliability Engineering*. New Delhi: Kluwer, 1993.

Musa, J.D., Lannino, A. and Okumoto, K., *Software Reliability: Measurement, Prediction and Applications*. New York: McGraw Hill, 1987.

The first significant OO design metrics suite was proposed by Chidamber and Kemerer in 1991.

Then came another paper by Chidamber and Kemerer defining and validating metrics suite for OO design in 1994. This metric suite has received the widest attention in empirical studies:

Chidamber, S., and Kemerer, C., A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, **20**(6): 476–493, 1994.

There are several books on research methodology and statistics with their applications:

Kothari, C.R., *Research Methodology: Methods and Techniques*. New Delhi: New Age International Limited, 2004.

Hopkins, W.G., A new view of statistics. *Sport Science*, 2003.

The details on outlier analysis can be obtained from:

Barnett, V. and Price, T., *Outliers in Statistical Data*. Chichester: John Wiley & Sons, 1995.

For a detailed account of the statistics needed for model prediction using logistic regression (notably how to compute maximum likelihood estimates, R^2 , significance values), see the following textbook and research paper:

Hosmer, D., and Lemeshow, S., *Applied Logistic Regression*. New York: John Wiley & Sons, 1989.

Basili, V., Briand, L., and Melo, W., "A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10): 751–761, 1996.

Detailed description on object-oriented metrics can be obtained from:

Lorenz, M. and Kidd, J., *Object-Oriented Software Metrics*. Englewood Cliff, NJ: Prentice-Hall, 1994.

Henderson-Sellers, B., *Object-oriented Metrics: Measures of Complexity*. Englewood Cliff, NJ: Prentice-Hall, 1996.

Various object-oriented metrics are explained with real-life examples in:

Aggarwal, K.K., Singh, Y., Kaur, A. and Malhotra, R., Empirical study of object-oriented metrics. *Journal of Object Technology*, 5(8): 149–173, 2006.

Relevant publications on object-oriented metrics can be obtained from:

<http://www.sbu.ac.uk/~csse/publications/OOMetrics.html>

Complete bibliography on object-oriented metrics can be obtained from:

Object-Oriented Metrics: An Annotated Bibliography, <http://dec.bournemouth.ac.uk/ESERG/bibliography.html>

9

Software Testing

Software testing is a very important, challenging and essential activity. It starts along with the design of SRS document and ends at the delivery of software product to the customer. It consumes significant effort and maximum time of software development life cycle (without including the maintenance phase). The significant effort may be from one third to one half of the total effort expended till the delivery of the software product. We cannot imagine to deliver a software product without adequate testing. The existing testing techniques help us to do adequate testing. However, adequate testing has different meaning to different software testers.

9.1 What is Software Testing?

The common perception about testing is to execute a program with given input(s) and note the observed output(s). The observed output(s) is/are matched with expected output(s) of the program. If it matches, the program is correct and if it does not match, the program is incorrect for the given input(s). There are many definitions of testing and some of them are given as follows:

- (i) The aim of testing is to **show that a program performs its desired functions correctly**.
- (ii) Testing is the process of demonstrating that errors are not present.
- (iii) Testing is the process of establishing confidence that a program does what it is supposed to do.

The purpose of testing as per the above definitions is to show the correctness of the program. We may select many inputs and execute the program and also demonstrate its correctness without touching critical and complex portions of the program. During testing, our objective should be to find faults and find them as early as possible. Hence, a more appropriate definition of testing is given by Myers (2004) as:

Testing is the process of executing a program with the intent of finding faults.

This definition motivates us to select those inputs which have higher probability of finding faults, although the definition is also not complete because it focuses only on the execution of the program. Nowadays, attention is also given to the activities like reviewing the documents and programs. Reviewing the documents (like SRS and SDD) helps us to find a good number of faults in the early phases of software development. Hence, testing is divided into two parts: verification and validation. Therefore, the most appropriate definition of software testing is:

Software testing is the process of verifying the outcomes of every phase of software development and validating the program by executing it with the intention of finding faults.

In the initial days of programming, testing was primarily validation oriented but nowadays both are equally important and carried out in most of the software organizations.

9.1.1 Verification

Software verification is also known as static testing where testing activities are carried out without executing the program. It may include inspections, walkthroughs and reviews where documents and programs are reviewed with the purpose of finding faults. Hence, verification is the process of reviewing (rechecking) documents and program with the intention of finding faults. As per the IEEE (2001):

The verification is the process of evaluating the system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

9.1.2 Validation

Software validation is also known as dynamic testing where the program is executed with given input(s). Hence, validation involves execution of the program and usually exposes symptoms of errors. As per the IEEE (2001):

The validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements.

Validation activities are carried out with the execution of the program. We experience failures and reasons of such failures are identified.

Software testing includes both verification and validation. Both are important and complementary to each other. Effective verification activities find a good number of faults in the early phases of software development. Removing such faults will definitely provide better foundations for the implementation/construction phase. Validation activities are possible only after the implementation of module/program, but verification activities are possible in every phase of software development.

9.2 Software Verification Techniques

Software verification techniques are applicable in every phase of software development life cycle. The purpose of such techniques is to review the outcomes of every phase of software development life cycle, i.e. from requirements and analysis phase to testing phase. Most of

the software professionals are convinced about the importance and usefulness of verification techniques. They also believe that a good quality software product can be produced only after the effective implementation of verification techniques. Many verification techniques are commonly used in practice and some of them are discussed in the following subsections.

9.2.1 Peer Reviews

This is the simplest, informal and oldest verification technique. It is applicable to every document produced at any stage of software development. Programs are also reviewed using this technique without executing them. We give documents/programs to some one else and ask them to review with an objective of finding faults. Some shortcomings or faults may be identified. This technique may give very good results if the reviewer has domain knowledge, technical expertise, programming skills and involvement in the reviewing work. A report may be prepared about the faults in the documents and programs. Sometimes, faults are also reported verbally during discussion. Reported faults are examined and appropriate corrections are made in the documents and programs. Every reviewing activity improves the quality of the documents and programs. Every reviewing activity improves the quality of the documents and programs without spending significant resources. This is very effective and easily applicable to small-size documents and programs. However, effectiveness reduces, as the size of the documents and programs increases.

9.2.2 Walkthroughs

Walkthrough is a group activity for reviewing the documents and programs. It is a more formal and systematic technique than peer reviews. A group of two to seven persons is constituted for the purpose of reviewing. The author of the document presents the document to the group during walkthroughs conducted in a conference room. The author may use any display mechanism (like whiteboard, projector) for presentation. Participants are not expected to prepare anything prior to such a meeting. Only the presenter who happens to be the author of the document prepares for the walkthrough. Documents and/or programs are distributed to every participant and the author presents the contents to all to make them understand. All participants are free to ask questions and write their observations on any display mechanism in such a way that everyone can see it. The observations are discussed thoroughly amongst the participants and changes are suggested accordingly. The author prepares a detailed report as per suggested changes. The report is studied, changes are approved and documents/programs are modified.

The limitations of such a technique are the non-preparation of participants prior to meeting and presentation of documents by their author(s). The author(s) may unnecessarily highlight some areas and hide a few critical areas. Participants may also not be able to ask critical and penetrating questions. Walkthroughs may create awareness amongst participants and may also find a few important faults. This technique is more expensive but systematic than peer reviews and applicable to any size of software project.

9.2.3 Inspections

This technique is more formal, systematic and effective. There are many names for this technique like formal reviews, formal technical reviews and inspections. The most popular

name is inspection and is different from peer reviews and walkthroughs. A group of three to six persons is constituted. The author of the document is not the presenter. An independent presenter is appointed for this specific purpose who prepares and understands the document. This preparation provides a second eye on the document(s) prior to meeting. The documents(s) is/are distributed to every participant in advance to give sufficient time for preparation. Rules of the meeting are prepared and circulated to all participants in advance to make them familiar with the framework of the meeting. The group is headed by an impartial moderator who should be a senior person to conduct the meetings smoothly. A recorder is also engaged to record the proceedings of the meetings.

The presenter presents the document(s) in the meeting and makes everyone comfortable with the document(s). Every participant is encouraged to participate in the deliberations as per modified rules. Everyone gets time to express his/her views, opinions about quality of document(s), potential faults and critical areas. The moderator chairs the meeting, enforces the discipline as per prescribed rules and respects views of every participant. Important observations are displayed in a way that is visible to every participant. The idea is to consider everyone's views and not to criticize them. Many times, a checklist is used to review the document. After the meeting, the moderator along with the recorder prepares a report and circulates it to all participants. Participants may discuss the report with the moderator and suggest changes. A final report is prepared after incorporating changes. The changes should be approved by the moderator. Most of the organizations use inspections as a verification technique and outcomes are very good. Many faults are generally identified in the early phases of software development. This is a popular, useful and systematic technique and is effective for every type of document and program. The comparison of all the three techniques is given in Table 9.1.

Table 9.1 Verification techniques comparison

	1	2	3
<i>Technique</i>	Peer reviews	Walkthrough	Inspections
<i>Presenter</i>	No one	Author	Someone other than the author
<i>Number of participants</i>	1 or 2	2 to 7 participants	3 to 6 participants
<i>Prior preparation</i>	Not required	Only the presenter is required to be prepared	All participants are required to be prepared
<i>Applicability</i>	Small-size software projects	Any size software projects	Any size software projects
<i>Report</i>	Optional	Compulsory	Compulsory
<i>Advantages</i>	Inexpensive and always finds some faults	Makes people aware about the project	Useful and finds many faults
<i>Disadvantages</i>	Dependent on the ability of reviewer	May find few faults	Skilled participants are needed and expensive

Verification is always more useful than validation due to its applicability in early phases of software development. It finds those faults which may not be detected by any validation technique. Verification techniques are becoming popular and more attention is given to such techniques from the beginning of the software development.

9.3 Checklist: A Popular Verification Tool

A checklist is normally used which consists of a list of important information contents that a deliverable must contain. A checklist may discipline the reviewing process and may identify duplicate information, missing information, and unclear and wrong information. Every document may have a different checklist which may be based on important, critical and difficult areas of the document. A checklist may be applicable in all verification techniques but it is more effective during inspections.

9.3.1 SRS Document Checklist

The SRS document is given in Chapter 3 (refer to Section 3.7) which is designed as per the IEEE standard 830–1998. Characteristics of good requirements are also given in Section 3.6 of Chapter 3 which may include the characteristics such as correct, unambiguous, complete, verifiable, modifiable, clear, feasible, necessary and understandable. Therefore, the SRS document checklist should address the above-mentioned characteristics. The SRS document is the source of many potential faults. It should be reviewed very seriously and thoroughly. The effective reviewing process improves the quality of the SRS document and minimizes the occurrence of many future problems. A properly designed checklist may help to achieve the objective of developing good quality software. A checklist is given in Table 9.2 which may be modified as per the requirement of the SRS document.

Table 9.2 Checklist for verifying SRS document

Part I

Reviewer name(s)	Organization	Review date	Project title

Part II

S. No.	Description	Yes/No/NA	Comments
Document overview			
1.	Are you satisfied with defined scope?		
2.	Are the objectives of the project clearly stated?		
3.	Has IEEE std-830-1998 been followed?		
4.	Is SRS document approved by the customer?		
5.	Is the layout of the screens well designed?		
6.	Are definitions, acronyms and abbreviations defined?		
7.	Do you suggest changes in the overall description?		
8.	Are user interfaces, hardware interfaces, software interfaces and communication interfaces clearly described?		
9.	Are major product functions stated?		
10.	Are you satisfied with the list of constraints?		
11.	Do you want to add any field in any form?		

(Contd.)

Table 9.2 Checklist for a good quality software (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
12.	Do you want to delete any field in any form?		
13.	Do you want to add/delete any validity check?		
14.	Is readability of the document satisfactory?		
15.	Are non-functional requirements specified?		
Actors, use cases and use case description			
16.	Are all the actors identified?		
17.	Are all the use cases determined?		
18.	Does the name of the use case represent the function it is required to perform?		
19.	Are all the actors included in the use case description of each use case?		
20.	Are you ready to accept identified use cases and their descriptions?		
21.	Does the use case description include precondition?		
22.	Does the use case description include postcondition?		
23.	Is the basic flow in the use case complete?		
24.	Are all the alternative flows of the use case stated?		
25.	Are related use cases stated?		
Characteristics of requirements			
26.	Are all requirements feasible?		
27.	Are requirements non-feasible due to technical problems?		
28.	Are requirements non-feasible due to inadequate resources?		
29.	Are a few requirements difficult to implement?		
30.	Are all requirements conveying only one meaning?		
31.	Are there conflicts amongst requirements?		
32.	Are all functional and non-functional requirement defined?		
33.	Are all use cases clearly understandable?		
34.	Are all forms available with adequate validity checks?		
35.	Are all requirements specified at a consistent level of detail?		
36.	Is consistency maintained throughout the document?		
37.	Are there any conflicts amongst requirements?		
38.	Are all defined requirements verifiable?		
39.	Are all stated requirements understandable?		
40.	Are redundant requirements identified?		
41.	Do you want to add a requirement?		
42.	Are all stated requirements written in a simple language?		

(Contd.)

Table 9.2 Checklist for a good quality software (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
43.	Are there any non-verifiable words in any requirement?		
44.	Are all assumptions and dependencies defined?		
45.	Are the requirements consistent with other documents of the project?		
General			
46.	Are you satisfied with the depth of the document?		
47.	Is logical database requirements specified?		
48.	Are design constraints described?		
49.	Is site adaptation requirement clearly defined?		
50.	Is product perspective written with adequate details?		

9.3.2 Object-Oriented Analysis Checklist

The checklist for OOA is given in Table 9.3 which may be used to verify the documents produced after OOA phase.

Table 9.3 Checklist for verifying OOA

Part I

Reviewer name(s)	Organization	Review date	Project title

Part II

S. No.	Description	Yes/No/NA	Comments
1.	Are all the classes—entity, interface and control—identified correctly?		
2.	Is any class missing per use case?		
3.	Is relationship between classes identified correctly?		
4.	Are entity classes persistent even after the end of use case?		
5.	Do control classes model flow of control in the system?		
6.	Does the name of the class convey the function it is intended to perform?		
7.	In case of association relationship, is multiplicity between classes identified correctly?		
8.	Are role names in case of association relationship between classes identified correctly?		
9.	Are all attributes identified correctly?		
10.	Does the name of the attribute convey the function it is intended to perform?		

(*Contd.*)

Table 9.3 Checklist for OOA (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
11.	Is UML syntax of each class correct?		
12.	Are the control classes necessary?		
13.	Is naming convention followed by the attributes?		
14.	Are classes identified for each use case?		
15.	Are all attributes defined clearly?		

9.3.3 Object-Oriented Design Checklist

The OOD checklist is given in Table 9.4 which may be used to verify the documents produced after the OOD phase.

Table 9.4 Checklist for verifying OOD

Part I

Reviewer name(s)	Organization	Review date	Project title

Part II

S. No.	Description	Yes/No/NA	Comments
1.	Are all the objects in the interaction diagram identified correctly?		
2.	Are all the classes in the interaction diagram identified correctly?		
3.	Are interaction diagrams created for each use case scenario?		
4.	Are all messages shown at right places in the interaction diagram?		
5.	Are all messages passing the parameters correctly?		
Sequence diagram			
6.	Do the sequence diagrams have an initiating actor?		
7.	Does an actor send message to the system in the sequence diagram?		
8.	Does the sequence diagram depict the order in which messages are sent?		
9.	Are the objects in the sequence diagram destroyed when not required?		
10.	Is the focus of control for each object identified correctly?		
11.	Are all the return messages identified correctly?		
12.	Are all the parameters of the messages identified correctly?		
13.	Are all the parameter types of the message parameters identified correctly?		

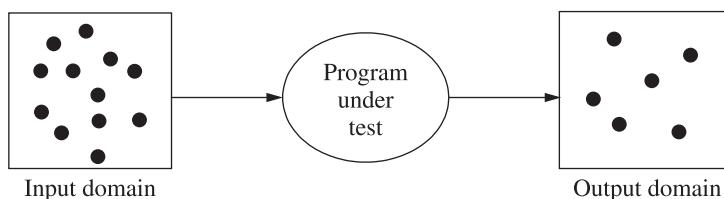
(*Contd.*)

Table 9.4 Checklist for OOD (*Contd.*)

S. No.	Description	Yes/No/NA	Comments
14.	Do the messages follow the naming convention?		
15.	Is the order of the messages identified correctly?		
16.	Are all swimlanes defined properly?		
17.	Are all states identified correctly in statechart diagrams?		
18.	Are all guard conditions used at proper places?		
19.	Are activity diagrams used to model the working of all processes?		
20.	Are all stated notations used in all activity diagrams and statechart diagrams?		

9.4 Functional Testing

Functional testing techniques are **validation techniques** because execution of the program is essential for the purpose of testing. Inputs are given to the program during execution and the **observed output is compared with the expected output**. If the observed output is different than the expected output, the situation is treated as a failure of the program. Functional testing techniques may design those test cases which have higher chances of making the program fail. These test cases are designed as per **functionality and internal structure** of the program is not at all considered. The program is treated as a black box and only functionality of the program is considered. Functional testing is also called black box testing because internal structure of the program is completely ignored and is shown in Figure 9.1.

**Figure 9.1** Functional (black box) testing.

Every dot of the input domain represents input(s) and every dot of the output domain represents output(s). Every dot of the input domain has a corresponding dot in the output domain. We consider valid and invalid inputs for the purpose of program execution and note the behaviour in terms of observed outputs. Functional testing techniques provide ways to design effective test cases to find errors in the program.

9.4.1 Boundary Value Analysis

The boundary value analysis technique focuses upon on or close to boundary values of input domain. We feel that the inputs on or close to boundary values have more chances to make the program fail after execution. Hence, test cases with input values on or close to boundary should

be designed. We consider a program ‘square root’ which takes a as an input value and prints the square root of a . The range of input value a is from 100 to 1000. We may execute the program for all input values from 100 to 1000 and observe the outputs of the program 900 times to see the output for every possible valid input. We may not like to execute the program for every possible valid input due to time and resource constraints. The boundary value analysis technique helps us to reduce the number of test cases by focusing only upon on or close to boundary values. On or close to boundary values cover the following:

- Minimum value
- Just above the minimum value
- Maximum value
- Just below the maximum value
- Nominal (average) value

These values are represented in Figure 9.2 for the program ‘square root’ with input value ranging from 100 to 1000.



Figure 9.2 Input to the program of ‘square root’.

The technique selects only five values instead of 900 values to test the program. These values of a are 100, 101, 550, 999, 1000 which cover the boundary portions of the input value except one nominal value (say 550). The nominal value represents those values which are neither on the boundary nor close to boundary. The number of test cases designed by this technique is $4n + 1$, where n is the number of input values. The test cases generated for the ‘square root’ program are shown in Table 9.5.

Table 9.5 Test cases of ‘square root’ program test cases

Test case	Input a	Expected output
1	100	10
2	101	10.05
3	550	23.45
4	999	31.61
5	1000	31.62

We consider a program ‘subtraction’ with two input values a and b and the output is the subtraction of a and b . The ranges of input values are shown as:

$$100 \leq a \leq 1000$$

$$200 \leq b \leq 1200$$

The selected values for a and b are given as:

$$a = (100, 101, 550, 999, 1000)$$

$$b = (200, 201, 700, 1199, 1200)$$

Both inputs (a and b) are required for the execution of the program. The input domain is shown in Figure 9.3 where any point within the rectangle is a legitimate input of the program.

The test cases are generated on the basis of ‘single fault’ assumption theory of reliability. This theory assumes that failures are rarely the result of simultaneous occurrence of two (or more) faults. Generally one fault is responsible for a failure. The implication of this theory is that we select one input for any of the defined states (minimum, just above minimum, just below maximum, maximum, nominal) and other input(s) as nominal values.

The test cases are $4n + 1$ (i.e. $8 + 1 = 9$) which are given in Table 9.6. The input values are also shown graphically in Figure 9.3.

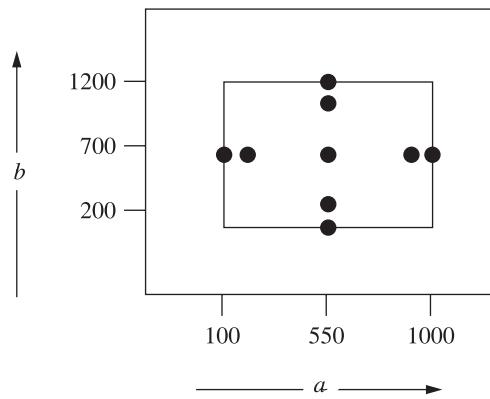


Figure 9.3 Graphical representation of inputs.

Table 9.6 Test cases for ‘subtraction’ program

Test case	a	b	Expected output
1	100	700	-600
2	101	700	-599
3	550	700	-150
4	999	700	299
5	1000	700	300
6	550	200	350
7	550	201	349
8	550	1199	-649
9	550	1200	-650

EXAMPLE 9.1 Consider a program to multiply and divide two numbers. The inputs may be two valid integers (say a and b) in the range of $[0, 100]$. Generate boundary value analysis test cases.

Solution Boundary value analysis test cases are given in Table 9.7.

Table 9.7 Boundary value analysis test cases for Example 9.1

Test case	a	b	Expected output	
1	0	50	0	0
2	1	50	50	0
3	50	50	2500	1
4	99	50	4950	1
5	100	50	5000	2
6	50	0	0	Divide by zero error
7	50	1	50	50
8	50	99	4950	0
9	50	100	5000	0

EXAMPLE 9.2 Consider a program which takes a date as an input and checks whether it is a valid date or not. Its input is a triple of day, month and year with the values in the following ranges:

$$1 \leq \text{month} \leq 12$$

$$1 \leq \text{day} \leq 31$$

$$2000 \leq \text{year} \leq 2070$$

Generate boundary value analysis test cases.

Solution Boundary value analysis test cases are given in Table 9.8.

Table 9.8 Boundary value test cases for program determining validity of date

Test case	Month	Day	Year	Expected output
1	1	15	2035	Valid date (1/15/2035)
2	2	15	2035	Valid date (2/15/2035)
3	6	15	2035	Valid date (6/15/2035)
4	11	15	2035	Valid date (11/15/2035)
5	12	15	2035	Valid date (12/15/2035)
6	6	1	2035	Valid date (6/1/2035)
7	6	2	2035	Valid date (6/2/2035)
8	6	30	2035	Valid date (6/30/2035)
9	6	31	2035	Invalid date
10	6	15	2000	Valid date (6/15/2000)
11	6	15	2001	Valid date (6/15/2001)
12	6	15	2069	Valid date (6/15/2069)
13	6	15	2070	Valid date (6/15/2070)

There are three extensions of boundary value analysis technique and are given below:

(i) Robustness testing

Two additional states ‘just below minimum’ and ‘just above maximum’ are added to see the behaviour of the program with invalid input values. Invalid inputs are equally important and may make the program fail when such inputs are given.

There are seven states in robustness testing and are given as:

1. Just below minimum (minimum^-)
2. Minimum
3. Just above minimum (minimum^+)
4. Nominal (average value)
5. Just below maximum (maximum^-)
6. Maximum
7. Just above maximum (maximum^+)

The total number of test cases generated for robustness testing is $6n + 1$, where n is the number of input values. Test cases for ‘subtraction’ program using robustness testing are 13 as given in Table 9.9.

Table 9.9 Robustness test cases for ‘subtraction’ program

Test case	a	b	Expected output
1	99	700	Invalid input
2	100	700	-600
3	101	700	-599
4	550	700	-150
5	999	700	299
6	1000	700	300
7	1001	700	Invalid input
8	550	199	Invalid input
9	550	200	350
10	550	201	349
11	550	1199	-649
12	550	1200	-650
13	550	1201	Invalid input

The input domain has four test cases which are outside; the other test cases lie in the legitimate input domain as shown in Figure 9.4.

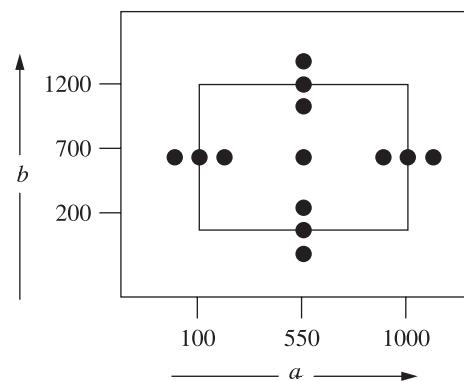


Figure 9.4 Input domain of ‘subtraction’ program for robustness test cases.

(ii) Worst case testing

This is another form of boundary value analysis where the single fault assumption theory of reliability is rejected. Hence, a failure may also be the result of the occurrence of more than one fault simultaneously. The result of this rejection is that one, two or all input values may have one of the following states:

- Minimum
- Minimum⁺
- Nominal (average value)
- Maximum⁻
- Maximum

In the worst case testing, the restriction of one input value at the above-mentioned states and other input values nominal is removed. The implication is that the number of test cases will increase from $4n + 1$ test cases to 5^n test cases, where n is the number of input values. The ‘subtraction’ program will have $5^2 = 25$ test cases which are shown in Table 9.10.

Table 9.10 Worst test cases for ‘subtraction’ program

Test case	a	b	Expected output
1	100	200	-100
2	100	201	-101
3	100	700	-600
4	100	1199	-1099
5	100	1200	-1100
6	101	200	-99
7	101	201	-100
8	101	700	-599
9	101	1199	-1098
10	101	1200	-1099
11	550	200	350
12	550	201	349
13	550	700	-150
14	550	1199	-649
15	550	1200	-650
16	999	200	799
17	999	201	798
18	999	700	299
19	999	1199	-200
20	999	1200	-201
21	1000	200	800
22	1000	201	799
23	1000	700	300
24	1000	1199	-199
25	1000	1200	-200

The inputs given in Table 9.10 are graphically represented in Figure 9.5.

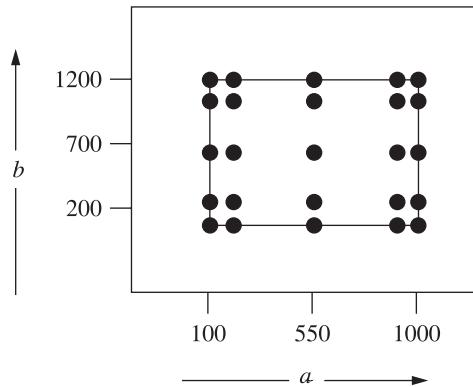


Figure 9.5 Graphical representation of inputs.

In the worst case testing, we select more number of test cases for completeness and thoroughness. This is a more effort and time-consuming technique.

(iii) Robust worst case testing

Two more states are added to check the behaviour of the program with invalid inputs. These two states are ‘just below minimum’ and ‘just above maximum’. The total number of test cases generated by this technique is 7^n . The subtraction program has $7^2 = 49$ test cases which are given in Table 9.11.

Table 9.11 Robust worst test cases for ‘subtraction’ program

Test case	a	b	Expected output
1	99	199	Invalid input
2	99	200	Invalid input
3	99	201	Invalid input
4	99	700	Invalid input
5	99	1199	Invalid input
6	99	1200	Invalid input
7	99	1201	Invalid input
8	100	199	Invalid input
9	100	200	-100
10	100	201	-101
11	100	700	-600
12	100	1199	-1099
13	100	1200	-1100
14	100	1201	Invalid input
15	101	199	Invalid input

(Contd.)

Table 9.11 Robust worst test cases for ‘subtraction’ program (*Contd.*)

Test case	<i>a</i>	<i>b</i>	Expected output
16	101	200	-99
17	101	201	-100
18	101	700	-599
19	101	1199	-1098
20	101	1200	-1099
21	101	1201	Invalid input
22	550	199	Invalid input
23	550	200	350
24	550	201	349
25	550	700	-150
26	550	1199	-649
27	550	1200	-650
28	550	1201	Invalid input
29	999	199	Invalid input
30	999	200	799
31	999	201	798
32	999	700	299
33	999	1199	-200
34	999	1200	-201
35	999	1201	Invalid input
36	1000	199	Invalid input
37	1000	200	800
38	1000	201	799
39	1000	700	300
40	1000	1199	-199
41	1000	1200	-200
42	1000	1201	Invalid input
43	1001	199	Invalid input
44	1001	200	Invalid input
45	1001	201	Invalid input
46	1001	700	Invalid input
47	1001	1199	Invalid input
48	1001	1200	Invalid input
49	1001	1201	Invalid input

The graphical representation of the input domain of ‘subtraction’ program is given in Figure 9.6.

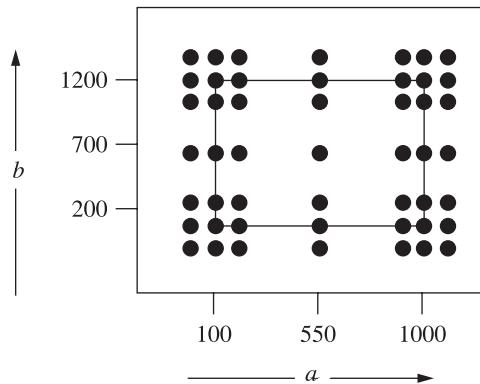


Figure 9.6 Graphical representation of inputs.

Boundary value analysis is a useful technique and may generate effective test cases. There is a restriction that input values should be independent. It is not applicable for Boolean variables because of only two available states—true and false. The technique is also easy to understand and implement for any size of program.

EXAMPLE 9.3 Consider a program to multiply and divide two numbers as explained in Example 9.1. Design the robust test cases and worst test cases for this program.

Solution Robust test cases are given in Table 9.12.

Table 9.12 Robust test cases for a program to multiply and divide two numbers

Test case	a	b	Expected output	
1	-1	50	Input values are out of range	
2	0	50	0	0
3	1	50	50	0
4	50	50	2500	1
5	99	50	4950	1
6	100	50	5000	2
7	101	50	Input values are out of range	
8	50	-1	Input values are out of range	
9	50	0	0	Divide by zero error
10	50	1	50	50
11	50	99	4950	0
12	50	100	5000	0
13	50	101	Input values are out of range	

Worst test cases are given in Table 9.13.

Table 9.13 Worst test cases for a program to multiply and divide two numbers

Test case	<i>a</i>	<i>b</i>	Expected output	
			Multiply	Divide
1	0	0	0	Undefined*
2	0	1	0	0
3	0	50	0	0
4	0	99	0	0
5	0	100	0	0
6	1	0	0	Divide by zero error
7	1	1	1	1
8	1	50	50	1
9	1	99	99	1
10	1	100	100	1
11	50	0	0	Divide by zero error
12	50	1	50	50
13	50	50	2500	1
14	50	99	4950	0
15	50	100	5000	0
16	99	0	0	Divide by zero error
17	99	1	99	99
18	99	50	4950	1
19	99	99	9801	1
20	99	100	9900	0
21	100	0	0	Divide by zero error
22	100	1	100	100
23	100	50	5000	2
24	100	99	9900	1
25	100	100	10,000	1

*0/0 is still undefined.

EXAMPLE 9.4 Consider the program for the determination of validity of the date as explained in Example 9.2. Design the robust and worst test cases for this program.

Solution Robust test cases and worst test cases are given in Tables 9.14 and 9.15, respectively.

Table 9.14 Robust test cases for program for determining the validity of the date

Test case	Month	Day	Year	Expected output
1	0	15	2035	Invalid date
2	1	15	2035	Valid date (1/15/2035)
3	2	15	2035	Valid date (2/15/2035)
4	6	15	2035	Valid date (6/15/2035)
5	11	15	2035	Valid date (11/15/2035)
6	12	15	2035	Valid date (12/15/2035)
7	13	15	2035	Invalid date
8	6	0	2035	Invalid date
9	6	1	2035	Valid date (6/1/2035)
10	6	2	2035	Valid date (6/2/2035)
11	6	30	2035	Valid date (6/30/2035)
12	6	31	2035	Invalid date
13	6	32	2035	Invalid date
14	6	15	1999	Invalid date (out of range)
15	6	15	2000	Valid date (6/15/2000)
16	6	15	2001	Valid date (6/15/2001)
17	6	15	2069	Valid date (6/15/2069)
18	6	15	2070	Valid date (6/15/2070)
19	6	15	2071	Invalid date (out of range)

Table 9.15 Worst test cases for the program determining the validity of the date

Test case	Month	Day	Year	Expected output
1	1	1	2000	Valid date (1/1/2000)
2	1	1	2001	Valid date (1/1/2001)
3	1	1	2035	Valid date (1/1/2035)
4	1	1	2069	Valid date (1/1/2069)
5	1	1	2070	Valid date (1/1/2070)
6	1	2	2000	Valid date (1/2/2000)
7	1	2	2001	Valid date (1/2/2001)
8	1	2	2035	Valid date (1/2/2035)
9	1	2	2069	Valid date (1/2/2069)
10	1	2	2070	Valid date (1/2/2070)
11	1	15	2000	Valid date (1/15/2000)
12	1	15	2001	Valid date (1/15/2001)
13	1	15	2035	Valid date (1/15/2035)
14	1	15	2069	Valid date (1/15/2069)

(Contd.)

Table 9.15 Worst test cases for the program determining the validity of the date (*Contd.*)

Test case	Month	Day	Year	Expected output
15	1	15	2070	Valid date (1/15/2070)
16	1	30	2000	Valid date (1/30/2000)
17	1	30	2001	Valid date (1/30/2001)
18	1	30	2035	Valid date (1/30/2035)
19	1	30	2069	Valid date (1/30/2069)
20	1	30	2070	Valid date (1/30/2070)
21	1	31	2000	Valid date (1/31/2000)
22	1	31	2001	Valid date (1/31/2001)
23	1	31	2035	Valid date (1/31/2035)
24	1	31	2069	Valid date (1/31/2069)
25	1	31	2070	Valid date (1/31/2070)
26	2	1	2000	Valid date (2/1/2000)
27	2	1	2001	Valid date (2/1/2001)
28	2	1	2035	Valid date (2/1/2035)
29	2	1	2069	Valid date (2/1/2069)
30	2	1	2070	Valid date (2/1/2070)
31	2	2	2000	Valid date (2/2/2000)
32	2	2	2001	Valid date (2/2/2001)
33	2	2	2035	Valid date (2/2/2035)
34	2	2	2069	Valid date (2/2/2069)
35	2	2	2070	Valid date (2/2/2070)
36	2	15	2000	Valid date (2/15/2000)
37	2	15	2001	Valid date (2/15/2001)
38	2	15	2035	Valid date (2/15/2035)
39	2	15	2069	Valid date (2/15/2069)
40	2	15	2070	Valid date (2/15/2070)
41	2	30	2000	Invalid date
42	2	30	2001	Invalid date
43	2	30	2035	Invalid date
44	2	30	2069	Invalid date
45	2	30	2070	Invalid date
46	2	31	2000	Invalid date
47	2	31	2001	Invalid date
48	2	31	2035	Invalid date
49	2	31	2069	Invalid date
50	2	31	2070	Invalid date

(Contd.)

Table 9.15 Worst test cases for the program determining the validity of the date (*Contd.*)

Test case	Month	Day	Year	Expected output
51	6	1	2000	Valid date (6/1/2000)
52	6	1	2001	Valid date (6/1/2001)
53	6	1	2035	Valid date (6/1/2035)
54	6	1	2069	Valid date (6/1/2069)
55	6	1	2070	Valid date (6/1/2070)
56	6	2	2000	Valid date (6/2/2000)
57	6	2	2001	Valid date (6/2/2001)
58	6	2	2035	Valid date (6/2/2035)
59	6	2	2069	Valid date (6/2/2069)
60	6	2	2070	Valid date (6/2/2070)
61	6	15	2000	Valid date (6/15/2000)
62	6	15	2001	Valid date (6/15/2001)
63	6	15	2035	Valid date (6/15/2035)
64	6	15	2069	Valid date (6/15/2069)
65	6	15	2070	Valid date (6/15/2070)
66	6	30	2000	Valid date (6/30/2000)
67	6	30	2001	Valid date (6/30/2001)
68	6	30	2035	Valid date (6/30/2035)
69	6	30	2069	Valid date (6/30/2069)
70	6	30	2070	Valid date (6/30/2070)
71	6	31	2000	Invalid date
72	6	31	2001	Invalid date
73	6	31	2035	Invalid date
74	6	31	2069	Invalid date
75	6	31	2070	Invalid date
76	11	1	2000	Valid date (11/1/2000)
77	11	1	2001	Valid date (11/1/2001)
78	11	1	2035	Valid date (11/1/2035)
79	11	1	2069	Valid date (11/1/2069)
80	11	1	2070	Valid date (11/1/2070)
81	11	2	2000	Valid date (11/2/2000)
82	11	2	2001	Valid date (11/2/2001)
83	11	2	2035	Valid date (11/2/2035)
84	11	2	2069	Valid date (11/2/2069)
85	11	2	2070	Valid date (11/2/2070)
86	11	15	2000	Valid date (11/15/2000)
87	11	15	2001	Valid date (11/15/2001)
88	11	15	2035	Valid date (11/15/2035)

Table 9.15 Worst test cases for the program determining the validity of the date (*Contd.*)

Test case	Month	Day	Year	Expected output
89	11	15	2069	Valid date (11/15/2069)
90	11	15	2070	Valid date (11/15/2070)
91	11	30	2000	Valid date (11/30/2000)
92	11	30	2001	Valid date (11/30/2001)
93	11	30	2035	Valid date (11/30/2035)
94	11	30	2069	Valid date (11/30/2069)
95	11	30	2070	Valid date (11/30/2070)
96	11	31	2000	Invalid date
97	11	31	2001	Invalid date
98	11	31	2035	Invalid date
99	11	31	2069	Invalid date
100	11	31	2070	Invalid date
101	12	1	2000	Valid date (12/1/2000)
102	12	1	2001	Valid date (12/1/2001)
103	12	1	2035	Valid date (12/1/2035)
104	12	1	2069	Valid date (12/1/2069)
105	12	1	2070	Valid date (12/1/2070)
106	12	2	2000	Valid date (12/2/2000)
107	12	2	2001	Valid date (12/2/2001)
108	12	2	2035	Valid date (12/2/2035)
109	12	2	2069	Valid date (12/2/2069)
110	12	2	2070	Valid date (12/2/2070)
111	12	15	2000	Valid date (12/15/2000)
112	12	15	2001	Valid date (12/15/2001)
113	12	15	2035	Valid date (12/15/2035)
114	12	15	2069	Valid date (12/15/2069)
115	12	15	2070	Valid date (12/15/2070)
116	12	30	2000	Valid date (12/30/2000)
117	12	30	2001	Valid date (12/30/2001)
118	12	30	2035	Valid date (12/30/2035)
119	12	30	2069	Valid date (12/30/2069)
120	12	30	2070	Valid date (12/30/2070)
121	12	31	2000	Valid date (12/31/2000)
122	12	31	2001	Valid date (12/31/2001)
123	12	31	2035	Valid date (12/31/2035)
124	12	31	2069	Valid date (12/31/2069)
125	12	31	2070	Valid date (12/31/2070)

9.4.2 Equivalence Class Testing

A program may have a large number of test cases. It may not be desirable to execute all test cases due to time and resource constraints. Many test cases may execute the same lines of source code again and again, and therefore, there is hardly any value addition. We may partition input domain into various groups on the basis of some relationship. We expect that any test case from that group will become the representative test case and produce the same behaviour. If a test case makes the program fail, then other test cases of the same group will also make the program fail. Similarly, if the output of the program is correct for one test case, the same behaviour is expected from other test cases of the group. This assumption of similar behaviour of the program for all test cases of the same group allows us to select only one test case from each group. If groups are made properly, a few test cases (equal to the number of groups) may give reasonable confidence about the correctness of the program. In this technique, each group is called an equivalence class. We may also partition the output domain into equivalence classes and generate one test case from each class.

Design of Equivalence Classes

In general, without considering any relationship, an input domain can be divided into at least two equivalence classes, i.e. one class of all valid inputs and other class of all invalid inputs. We may design many classes on the basis of relationships and logic of the program. The input domain and output domain may generate a good number of classes and every class gives us a test case which will represent all test cases of that class. We consider the program ‘square root’ which takes a as an input in the range (100–1000) and prints the square root of a . We may generate the following equivalence classes for the input domain:

$$I_1 = \{100 \leq a \leq 1000\} \text{ (valid input range from 100 to 1000)}$$

$$I_2 = \{a < 100\} \text{ (any invalid input where } a \text{ is less than 100)}$$

$$I_3 = \{a > 1000\} \text{ (any invalid input where } a \text{ is more than 1000)}$$

The above three equivalence classes are designed without considering any relationship. For example, the first class I_1 represents a valid input class where all inputs are within the specified range $\{100 \leq a \leq 1000\}$. Three test cases are generated, one from every equivalence class, and are given in Table 9.16.

Table 9.16 Test cases of ‘square root’ program for input domain

Test case	Input a	Expected output
I_1	500	22.36
I_2	10	Invalid input
I_3	1100	Invalid input

The output domain is also partitioned to generate equivalence classes. We generate the following output domain equivalence classes:

$$O_1 = \{\text{Square root of input value } a\}$$

$$O_2 = \{\text{Invalid value}\}$$

The test cases for the output domain are given in Table 9.17. Some of the input and output domain test cases may be the same.

Table 9.17 Test cases of ‘square root’ program for output domain

Test case	Input a	Expected output
O ₁	500	22.36
O ₂	1100	Invalid input

We consider the ‘subtraction’ program which takes two inputs a (range 100–1000) and b (range 200–1200) and performs the subtraction of these two input values. On the basis of the input domain, we generate the following equivalence classes:

- (i) I₁ = {100 ≤ a ≤ 1000 and 200 ≤ b ≤ 1200} (Both a and b are valid values)
- (ii) I₂ = {100 ≤ a ≤ 1000 and $b < 200$ } (a is valid and b is invalid)
- (iii) I₃ = {100 ≤ a ≤ 1000 and $b > 1200$ } (a is valid and b is invalid)
- (iv) I₄ = { $a < 100$ and 200 ≤ b ≤ 1200} (a is invalid and b is valid)
- (v) I₅ = { $a > 1000$ and 200 ≤ b ≤ 1200} (a is invalid and b is valid)
- (vi) I₆ = { $a < 100$ and $b < 200$ } (Both inputs are invalid)
- (vii) I₇ = { $a < 100$ and $b > 1200$ } (Both inputs are invalid)
- (viii) I₈ = { $a > 1000$ and $b < 200$ } (Both inputs are invalid)
- (ix) I₉ = { $a > 1000$ and $b > 1200$ } (Both inputs are invalid)

The input domain test cases are given in Table 9.18.

Table 9.18 Test cases of ‘subtraction’ program for input domain

Test case	a	b	Expected output
I ₁	600	300	300
I ₂	600	100	Invalid input
I ₃	600	1300	Invalid input
I ₄	10	300	Invalid input
I ₅	1100	300	Invalid input
I ₆	10	100	Invalid input
I ₇	10	1300	Invalid input
I ₈	1100	100	Invalid input
I ₉	1100	1300	Invalid input

The output domain equivalence classes are given as follows:

$$O_1 = \{\text{subtraction of two values } a \text{ and } b\}$$

$$O_2 = \{\text{Invalid input}\}$$

The test cases for the output domain are given in Table 9.19.

Table 9.19 Test cases of ‘subtraction’ program for output domain

Test case	a	b	Expected output
O ₁	600	300	300
O ₂	10	300	Invalid input

We have designed only one equivalence class for the valid input domain in both of the above discussed examples. We could not partition the valid input domain due to simplicity of the problem. However, programs are more complex and logic oriented. We may design more valid input domain equivalence classes on the basis of logic and relationship. The design of equivalence classes is subjective and two designers may design different classes. The objective is to cover every functionality of the program which may further cover maximum portion of the source code during testing. This technique is applicable at unit, integration system and acceptance testing levels. This is a very effective technique if equivalence classes are designed correctly and also reduces the number of test cases significantly.

EXAMPLE 9.5 Consider a program to multiply and divide two numbers. The inputs may be two valid integers (say a and b) in the range of [0, 100]. Develop test cases using equivalence class testing.

Solution The test cases are generated for output domain given in Table 9.20:

Table 9.20 Test cases for output domain

Test case	a	b	Expected output	
			Multiply	Divide
O ₁	50	50	2500	1
O ₂	101	50	Input values are out of range	

The equivalence classes for input domain are shown below:

- (i) I₁ = {0 ≤ a ≤ 100 and 0 ≤ b ≤ 100} (Both a and b are valid values)
- (ii) I₂ = {0 ≤ a ≤ 100 and b < 0} (a is valid and b is invalid)
- (iii) I₃ = {0 ≤ a ≤ 100 and b > 100} (a is valid and b is invalid)
- (iv) I₄ = {a < 0 and 0 ≤ b ≤ 100} (a is invalid and b is valid)
- (v) I₅ = {a > 100 and 0 ≤ b ≤ 100} (a is invalid and b is valid)
- (vi) I₆ = {a < 0 and b < 0} (Both inputs are invalid)
- (vii) I₇ = {a < 0 and b > 100} (Both inputs are invalid)
- (viii) I₈ = {a > 100 and b < 0} (Both inputs are invalid)
- (ix) I₉ = {a > 100 and b > 100} (Both inputs are invalid)

The test cases for input domain are given in Table 9.21.

Table 9.21 Test cases for input domain

Test case	<i>a</i>	<i>b</i>	Expected output	
I ₁	50	50	2500	1
I ₂	50	-1	Input values are out of range	
I ₃	50	101	Input values are out of range	
I ₄	-1	50	Input values are out of range	
I ₅	101	50	Input values are out of range	
I ₆	-1	-1	Input values are out of range	
I ₇	-1	101	Input values are out of range	
I ₈	101	-1	Input values are out of range	
I ₉	101	101	Input values are out of range	

EXAMPLE 9.6 Consider the program for determining whether the date is valid or not. Identify the equivalence class test cases for output and input domains.

Solution Output domain equivalence classes are:

$$O_1 = \{< \text{Day}, \text{Month}, \text{Year} > : \text{Valid}\}$$

$$O_2 = \{< \text{Day}, \text{Month}, \text{Year} > : \text{Invalid date if any of the inputs is invalid}\}$$

$$O_3 = \{< \text{Day}, \text{Month}, \text{Year} > : \text{Input is out of range if any of the inputs is out of range}\}$$

The output domain test cases are given in Table 9.22.

Table 9.22 Output domain equivalence class test cases

Test case	Month	Day	Year	Expected output
O ₁	6	11	1979	Valid date
O ₂	6	31	1979	Invalid date
O ₃	6	32	1979	Inputs out of range

The input domain is partitioned as given below:

(i) Valid partitions

M₁: Month has 30 days

M₂: Month has 31 days

M₃: Month is February

D₁: Days of a month from 1 to 28

D₂: Day = 29

D₃: Day = 30

D₄: Day = 31

Y₁: 1900 ≤ year ≤ 2058 and is a common year

Y₂: 1900 ≤ year ≤ 2058 and is a leap year

(ii) Invalid partitions

M₄: Month < 1

M₅: Month > 12

D₅: Day < 1

D_6 : Day > 31
 Y_3 : Year < 1900
 Y_4 : Year > 2058

We may have the following set of test cases which are based on input domain:

(a) Only for valid input domain

$I_1 = \{M_1 \text{ and } D_1 \text{ and } Y_1\}$ (All inputs are valid)
 $I_2 = \{M_2 \text{ and } D_1 \text{ and } Y_1\}$ (All inputs are valid)
 $I_3 = \{M_3 \text{ and } D_1 \text{ and } Y_1\}$ (All inputs are valid)
 $I_4 = \{M_1 \text{ and } D_2 \text{ and } Y_1\}$ (All inputs are valid)
 $I_5 = \{M_2 \text{ and } D_2 \text{ and } Y_1\}$ (All inputs are valid)
 $I_6 = \{M_3 \text{ and } D_2 \text{ and } Y_1\}$ (All inputs are valid)
 $I_7 = \{M_1 \text{ and } D_3 \text{ and } Y_1\}$ (All inputs are valid)
 $I_8 = \{M_2 \text{ and } D_3 \text{ and } Y_1\}$ (All inputs are valid)
 $I_9 = \{M_3 \text{ and } D_3 \text{ and } Y_1\}$ (All inputs are valid)
 $I_{10} = \{M_1 \text{ and } D_4 \text{ and } Y_1\}$ (All inputs are valid)
 $I_{11} = \{M_2 \text{ and } D_4 \text{ and } Y_1\}$ (All inputs are valid)
 $I_{12} = \{M_3 \text{ and } D_4 \text{ and } Y_1\}$ (All inputs are valid)
 $I_{13} = \{M_1 \text{ and } D_1 \text{ and } Y_2\}$ (All Inputs are valid)
 $I_{14} = \{M_2 \text{ and } D_1 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{15} = \{M_3 \text{ and } D_1 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{16} = \{M_1 \text{ and } D_2 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{17} = \{M_2 \text{ and } D_2 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{18} = \{M_3 \text{ and } D_2 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{19} = \{M_1 \text{ and } D_3 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{20} = \{M_2 \text{ and } D_3 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{21} = \{M_3 \text{ and } D_3 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{22} = \{M_1 \text{ and } D_4 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{23} = \{M_2 \text{ and } D_4 \text{ and } Y_2\}$ (All inputs are valid)
 $I_{24} = \{M_3 \text{ and } D_4 \text{ and } Y_2\}$ (All inputs are valid)

(b) Only for invalid input domain

$I_{25} = \{M_4 \text{ and } D_1 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{26} = \{M_5 \text{ and } D_1 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{27} = \{M_4 \text{ and } D_2 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{28} = \{M_5 \text{ and } D_2 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{29} = \{M_4 \text{ and } D_3 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{30} = \{M_5 \text{ and } D_3 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{31} = \{M_4 \text{ and } D_4 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{32} = \{M_5 \text{ and } D_4 \text{ and } Y_1\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{33} = \{M_4 \text{ and } D_1 \text{ and } Y_2\}$ (Month is invalid, Day is valid and Year is valid)
 $I_{34} = \{M_5 \text{ and } D_1 \text{ and } Y_2\}$ (Month is invalid, Day is valid and Year is valid)

I₇₅ = {M₃ and D₄ and Y₃} (Month is valid, Day is valid and Year is invalid)
I₇₆ = {M₃ and D₄ and Y₄} (Month is valid, Day is valid and Year is invalid)
I₇₇ = {M₄ and D₅ and Y₁} (Month is invalid, Day is invalid and Year is valid)
I₇₈ = {M₄ and D₅ and Y₂} (Month is invalid, Day is invalid and Year is valid)
I₇₉ = {M₄ and D₆ and Y₁} (Month is invalid, Day is invalid and Year is valid)
I₈₀ = {M₄ and D₆ and Y₂} (Month is invalid, Day is invalid and Year is valid)
I₈₁ = {M₅ and D₅ and Y₁} (Month is invalid, Day is invalid and Year is valid)
I₈₂ = {M₅ and D₅ and Y₂} (Month is invalid, Day is invalid and Year is valid)
I₈₃ = {M₅ and D₆ and Y₁} (Month is invalid, Day is invalid and Year is valid)
I₈₄ = {M₅ and D₆ and Y₂} (Month is invalid, Day is invalid and Year is valid)
I₈₅ = {M₄ and D₁ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₈₆ = {M₄ and D₁ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₈₇ = {M₄ and D₂ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₈₈ = {M₄ and D₂ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₈₉ = {M₄ and D₃ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₉₀ = {M₄ and D₃ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₉₁ = {M₄ and D₄ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₉₂ = {M₄ and D₄ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₉₃ = {M₅ and D₁ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₉₄ = {M₅ and D₁ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₉₅ = {M₅ and D₂ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₉₆ = {M₅ and D₂ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₉₇ = {M₅ and D₃ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₉₈ = {M₅ and D₃ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₉₉ = {M₅ and D₄ and Y₃} (Month is invalid, Day is valid and Year is invalid)
I₁₀₀ = {M₅ and D₄ and Y₄} (Month is invalid, Day is valid and Year is invalid)
I₁₀₁ = {M₁ and D₅ and Y₃} (Month is valid, Day is invalid and Year is invalid)
I₁₀₂ = {M₁ and D₅ and Y₄} (Month is valid, Day is invalid and Year is invalid)
I₁₀₃ = {M₂ and D₅ and Y₃} (Month is valid, Day is invalid and Year is invalid)
I₁₀₄ = {M₂ and D₅ and Y₄} (Month is valid, Day is invalid and Year is invalid)
I₁₀₅ = {M₃ and D₅ and Y₃} (Month is valid, Day is invalid and Year is invalid)
I₁₀₆ = {M₃ and D₅ and Y₄} (Month is valid, Day is invalid and Year is invalid)
I₁₀₇ = {M₁ and D₆ and Y₃} (Month is valid, Day is invalid and Year is invalid)
I₁₀₈ = {M₁ and D₆ and Y₄} (Month is valid, Day is invalid and Year is invalid)
I₁₀₉ = {M₂ and D₆ and Y₃} (Month is valid, Day is invalid and Year is invalid)
I₁₁₀ = {M₂ and D₆ and Y₄} (Month is valid, Day is invalid and Year is invalid)
I₁₁₁ = {M₃ and D₆ and Y₃} (Month is valid, Day is invalid and Year is invalid)
I₁₁₂ = {M₃ and D₆ and Y₄} (Month is valid, Day is invalid and Year is invalid)
I₁₁₃ = {M₄ and D₅ and Y₃} (All inputs are invalid)

$I_{114} = \{M_4 \text{ and } D_5 \text{ and } Y_4\}$ (All inputs are invalid)

$I_{115} = \{M_4 \text{ and } D_6 \text{ and } Y_3\}$ (All inputs are invalid)

$I_{116} = \{M_4 \text{ and } D_6 \text{ and } Y_4\}$ (All inputs are invalid)

$I_{117} = \{M_5 \text{ and } D_5 \text{ and } Y_3\}$ (All inputs are invalid)

$I_{118} = \{M_5 \text{ and } D_5 \text{ and } Y_4\}$ (All inputs are invalid)

$I_{119} = \{M_5 \text{ and } D_6 \text{ and } Y_3\}$ (All inputs are invalid)

$I_{120} = \{M_5 \text{ and } D_6 \text{ and } Y_4\}$ (All inputs are invalid)

The test cases generated on the basis of input domain are given in Table 9.23.

Table 9.23 Input domain equivalence class test cases

Test case	Month	Day	Year	Expected output
I_1	6	15	2035	Valid date
I_2	5	15	2035	Valid date
I_3	2	15	2035	Valid date
I_4	6	29	2035	Valid date
I_5	5	29	2035	Valid date
I_6	2	29	2035	Invalid date
I_7	6	30	2035	Valid date
I_8	5	30	2035	Valid date
I_9	2	30	2035	Invalid date
I_{10}	6	31	2035	Invalid date
I_{11}	5	31	2035	Valid date
I_{12}	2	31	2035	Invalid date
I_{13}	6	15	2000	Valid date
I_{14}	5	15	2000	Valid date
I_{15}	2	15	2000	Valid date
I_{16}	6	29	2000	Valid date
I_{17}	5	29	2000	Valid date
I_{18}	2	29	2000	Valid date
I_{19}	6	30	2000	Valid date
I_{20}	5	30	2000	Valid date
I_{21}	2	30	2000	Invalid date
I_{22}	6	31	2000	Invalid date
I_{23}	5	31	2000	Valid date
I_{24}	2	31	2000	Invalid date
I_{25}	0	15	2035	Input(s) out of range
I_{26}	13	15	2035	Input(s) out of range
I_{27}	0	29	2035	Inputs(s) out of range
I_{28}	13	29	2035	Input(s) out of range

(Contd.)

Table 9.23 Input domain equivalence class test cases (*Contd.*)

Test case	Month	Day	Year	Expected output
I ₂₉	0	30	2035	Input(s) out of range
I ₃₀	13	30	2035	Input(s) out of range
I ₃₁	0	31	2035	Input(s) out of range
I ₃₂	13	31	2035	Input(s) out of range
I ₃₃	0	15	2000	Input(s) out of range
I ₃₄	13	15	2000	Input(s) out of range
I ₃₅	0	29	2000	Input(s) out of range
I ₃₆	13	29	2000	Input(s) out of range
I ₃₇	0	30	2000	Input(s) out of range
I ₃₈	13	30	2000	Input(s) out of range
I ₃₉	0	31	2000	Input(s) out of range
I ₄₀	13	31	2000	Input(s) out of range
I ₄₁	6	0	2035	Input(s) out of range
I ₄₂	6	32	2035	Input(s) out of range
I ₄₃	5	0	2035	Input(s) out of range
I ₄₄	5	32	2035	Input(s) out of range
I ₄₅	2	0	2035	Input(s) out of range
I ₄₆	2	32	2035	Input(s) out of range
I ₄₇	6	0	2000	Input(s) out of range
I ₄₈	6	32	2000	Input(s) out of range
I ₄₉	5	0	2000	Input(s) out of range
I ₅₀	5	32	2000	Input(s) out of range
I ₅₁	2	0	2000	Input(s) out of range
I ₅₂	2	32	2000	Input(s) out of range
I ₅₃	6	15	1899	Input(s) out of range
I ₅₄	6	15	2059	Input(s) out of range
I ₅₅	5	15	1899	Input(s) out of range
I ₅₆	5	15	2059	Input(s) out of range
I ₅₇	2	15	1899	Input(s) out of range
I ₅₈	2	15	2059	Input(s) out of range
I ₅₉	6	29	1899	Input(s) out of range
I ₆₀	6	29	2059	Input(s) out of range
I ₆₁	5	29	1899	Input(s) out of range
I ₆₂	5	29	2059	Input(s) out of range
I ₆₃	2	29	1899	Input(s) out of range
I ₆₄	2	29	2059	Input(s) out of range
I ₆₅	6	30	1899	Input(s) out of range

(Contd.)

Table 9.23 Input domain equivalence class test cases (*Contd.*)

Test case	Month	Day	Year	Expected output
I ₆₆	6	30	2059	Input(s) out of range
I ₆₇	5	30	1899	Input(s) out of range
I ₆₈	5	30	2059	Input(s) out of range
I ₆₉	2	30	1899	Input(s) out of range
I ₇₀	2	30	2059	Input(s) out of range
I ₇₁	6	31	1899	Input(s) out of range
I ₇₂	6	31	2059	Input(s) out of range
I ₇₃	5	31	1899	Input(s) out of range
I ₇₄	5	31	2059	Input(s) out of range
I ₇₅	2	31	1899	Input(s) out of range
I ₇₆	2	31	2059	Input(s) out of range
I ₇₇	0	0	2035	Input(s) out of range
I ₇₈	0	0	2000	Input(s) out of range
I ₇₉	0	32	2035	Input(s) out of range
I ₈₀	0	32	2000	Input(s) out of range
I ₈₁	13	0	2035	Input(s) out of range
I ₈₂	13	0	2000	Input(s) out of range
I ₈₃	13	32	2035	Input(s) out of range
I ₈₄	13	32	2000	Input(s) out of range
I ₈₅	0	15	1899	Input(s) out of range
I ₈₆	0	15	2059	Input(s) out of range
I ₈₇	0	20	1899	Input(s) out of range
I ₈₈	0	29	2059	Input(s) out of range
I ₈₉	0	30	1899	Input(s) out of range
I ₉₀	0	30	2059	Input(s) out of range
I ₉₁	0	31	1899	Input(s) out of range
I ₉₂	0	31	2059	Input(s) out of range
I ₉₃	13	15	1899	Input(s) out of range
I ₉₄	13	15	2059	Input(s) out of range
I ₉₅	13	29	1899	Input(s) out of range
I ₉₆	13	29	2059	Input(s) out of range
I ₉₇	13	30	1899	Input(s) out of range
I ₉₈	13	30	2059	Input(s) out of range
I ₉₉	13	31	1899	Input(s) out of range
I ₁₀₀	13	31	2059	Input(s) out of range
I ₁₀₁	5	0	1899	Input(s) out of range
I ₁₀₂	5	0	2059	Input(s) out of range

(Contd.)

Table 9.23 Input domain equivalence class test cases (*Contd.*)

Test case	Month	Day	Year	Expected output
I ₁₀₃	6	0	1899	Input(s) out of range
I ₁₀₄	6	0	2059	Input(s) out of range
I ₁₀₅	2	0	1899	Input(s) out of range
I ₁₀₆	2	0	2059	Input(s) out of range
I ₁₀₇	5	32	1899	Input(s) out of range
I ₁₀₈	5	32	2059	Input(s) out of range
I ₁₀₉	6	32	1899	Input(s) out of range
I ₁₁₀	6	32	2059	Input(s) out of range
I ₁₁₁	2	32	1899	Input(s) out of range
I ₁₁₂	2	32	2059	Input(s) out of range
I ₁₁₃	0	0	1899	Input(s) out of range
I ₁₁₄	0	0	2059	Input(s) out of range
I ₁₁₅	0	32	1899	Input(s) out of range
I ₁₁₆	0	32	2059	Input(s) out of range
I ₁₁₇	13	0	1899	Input(s) out of range
I ₁₁₈	13	0	2059	Input(s) out of range
I ₁₁₉	13	32	1899	Input(s) out of range
I ₁₂₀	13	32	2059	Input(s) out of range

9.4.3 Decision Table-Based Testing

Decision tables are commonly used in engineering disciplines to represent complex logical relationships. They are effective to model situations where an output is dependent on many input conditions. Software testers have also found their applications in testing and a technique is developed which is known as decision table-based testing. There are four portions of a decision table, namely, condition stubs, condition entries, action stubs and action entries. A typical decision table is shown in Table 9.24.

Table 9.24 Portion of the decision table

	Stubs	Entries
Condition	All conditions are shown.	Inputs are shown on the basis of conditions. There are columns and each column represents a rule.
Action	All actions are shown.	Represent the outputs on the basis of various input conditions.

There are two types of decision tables. The first type is called the limited entry decision table where input values represent only the true and false condition as shown in Table 9.25.

Table 9.25 Limited entry decision table

Condition stub	Condition entry				
	T	F	F	F	F
c ₁	T	F	F	F	F
c ₂	-	T	F	F	F
c ₃	-	-	T	F	F
	-	-	-	T	F
a ₁	X		X		
a ₂		X			X
a ₃	X			X	

Every column of the decision table represents a rule and generates a test case. A ‘-’ in the condition entry represents a ‘do not care’ condition. An ‘X’ in the action entry represents the action mentioned in the corresponding action stub. In Table 9.25, if condition c₁ is true, then c₂, c₃ and c₄ become ‘do not care’ conditions and actions a₁ and a₃ are to be performed.

The second type of decision table is called extended entry decision table. In such a table, multiple conditions are used instead of true and false conditions. A condition may have many options instead of only true and false and such options are represented in the extended entry decision table.

We consider a program which takes a date as an input and checks whether it is a valid date or not. We prepare the following classes (options) and represent them in the extended entry decision table as shown in Table 9.26.

- I₁ = {M₁ : Month has 30 days}
- I₂ = {M₂ : Month has 31 days}
- I₃ = {M₃ : Month is February}
- I₄ = {M₄ : Month < 1}
- I₅ = {M₅ : Month > 12}
- I₆ = {D₁ : 1 ≤ Day ≤ 28}
- I₇ = {D₂ : Day = 29}
- I₈ = {D₃ : Day = 30}
- I₉ = {D₄ : Day = 31}
- I₁₀ = {D₅ : Day < 1}
- I₁₁ = {D₆ : Day > 31}
- I₁₂ = {Y₁ : 1900 ≤ Year ≤ 2058 and is a common year}
- I₁₃ = {Y₂ : 1900 ≤ Year ≤ 2058 and is a leap year}
- I₁₄ = {Y₃ : Year < 1900}
- I₁₅ = {Y₄ : Year > 2058}

In the decision table, various combinations of conditions are considered and that may sometimes result into an impossible action. In such a situation, we may incorporate an additional action ‘impossible condition’ in the action stub. We may change the design of equivalence classes to reduce the impossible conditions.

Table 9.26 Decision table of date program

Decision tables are used in situations where an output is dependent on many input conditions. Complex relationships can also be easily represented in such tables. Every column may generate a test case. The test cases of decision Table 9.26 are given in Table 9.27.

Table 9.27 Test cases of the program of date problem

Test case	Month	Day	Year	Expected output
1	6	15	1979	Valid date
2	6	15	2000	Valid date
3	6	15	1899	Input out of range
4	6	15	2059	Input out of range
5	6	29	1979	Valid date
6	6	29	2000	Valid date
7	6	29	1899	Input out of range
8	6	29	2059	Input out of range
9	6	30	1979	Valid date
10	6	30	2000	Valid date
11	6	30	1899	Input out of range
12	6	30	2059	Input out of range
13	6	31	1979	Invalid date
14	6	31	2000	Invalid date
15	6	31	1899	Input out of range
16	6	31	2059	Input out of range
17	6	0	1979	Input out of range
18	6	32	1979	Input out of range
19	5	15	1979	Valid date
20	5	15	2000	Valid date
21	5	15	1899	Input out of range
22	5	15	2059	Input out of range
23	5	29	1979	Valid date
24	5	29	2000	Valid date
25	5	29	1899	Input out of range
26	5	29	2059	Input out of range
27	5	30	1979	Valid date
28	5	30	2000	Valid date
29	5	30	1899	Input out of range
30	5	30	2059	Input out of range
31	5	31	1979	Valid date
32	5	31	2000	Valid date
33	5	31	1899	Input out of range

(Contd.)

Table 9.27 Test cases of the program of date problem (*Contd.*)

Test case	Month	Day	Year	Expected output
34	5	31	2059	Input out of range
35	5	0	1979	Input out of range
36	5	32	1979	Input out of range
37	2	15	1979	Valid date
38	2	15	2000	Valid date
39	2	15	1899	Input out of range
40	2	15	2059	Input out of range
41	2	29	1979	Invalid date
42	2	29	2000	Valid date
43	2	29	1899	Input out of range
44	2	29	2059	Input out of range
45	2	30	1979	Invalid date
46	2	30	2000	Invalid date
47	2	30	1899	Input out of range
48	2	30	2059	Input out of range
49	2	31	1979	Invalid date
50	2	31	2000	Invalid date
51	2	31	1899	Input out of range
52	2	31	2059	Input out of range
53	2	0	1979	Input out of range
54	2	32	1979	Input out of range
55	0	0	1899	Input out of range
56	13	32	1899	Input out of range

Decision tables are effectively applicable to small-size programs. As the size increases, handling becomes difficult and time consuming. We can easily apply the decision table at the unit level. System testing and integration testing do not find its applicability in any reasonable-size program.

EXAMPLE 9.7 Consider a program to multiply and divide two numbers. The inputs may be two valid integers (say a and b) in the range of $[0, 100]$. Develop the decision table and generate test cases.

Solution The decision table is given in Table 9.28 and the test cases are given in Table 9.29.

Table 9.28 Decision table for a program to multiply and divide two numbers

Input in valid range?	F	T	T	T	T
$a = 0?$	-	T	T	F	F
$b = 0?$	-	T	F	T	F
Input values out of range	X				
Valid output			X		X
Output undefined		X			
Divide by zero error				X	

Table 9.29 shows the test cases generated from the above decision table.

Table 9.29 Test cases for decision table shown in Table 9.28

Test case	a	b	Expected output	
1	50	-1	Input values are out of range	
2	0	0	Output undefined	
3	0	50	0	0
4	30	0	Divide by zero error	
5	50	50	2500	1

9.5 Structural Testing

Structural testing is complementary to functional testing where the source code is considered for the design of test cases rather than specifications. We focus on the internal structure of the source code and ignore the functionality of the program. Structural testing is also called white box testing and attempts to examine the source code rigorously and thoroughly to understand it correctly. The clear and correct understanding of the source code may find complex and weak areas and test cases are generated accordingly.

9.5.1 Path Testing

It is a popular structural testing technique. The source code of the program is converted into a program graph which represents the flow of control in terms of directed graphs. The program is further converted into the decision to decision (DD) path graph. Both graphs are commonly used in structural testing techniques for the generation of test cases.

A program graph is a graphical representation of the source code where statements of the program are represented by nodes and flow of control by edges. Jorgenson (2007) has defined program graph as:

A program graph is a directed graph in which nodes are either statements or fragments of a statement and edges represent flow of control.

The program graph provides the graphical view of the program and may become the foundation of many testing techniques. The fundamental constructs of the program graph are given in Figure 9.7.

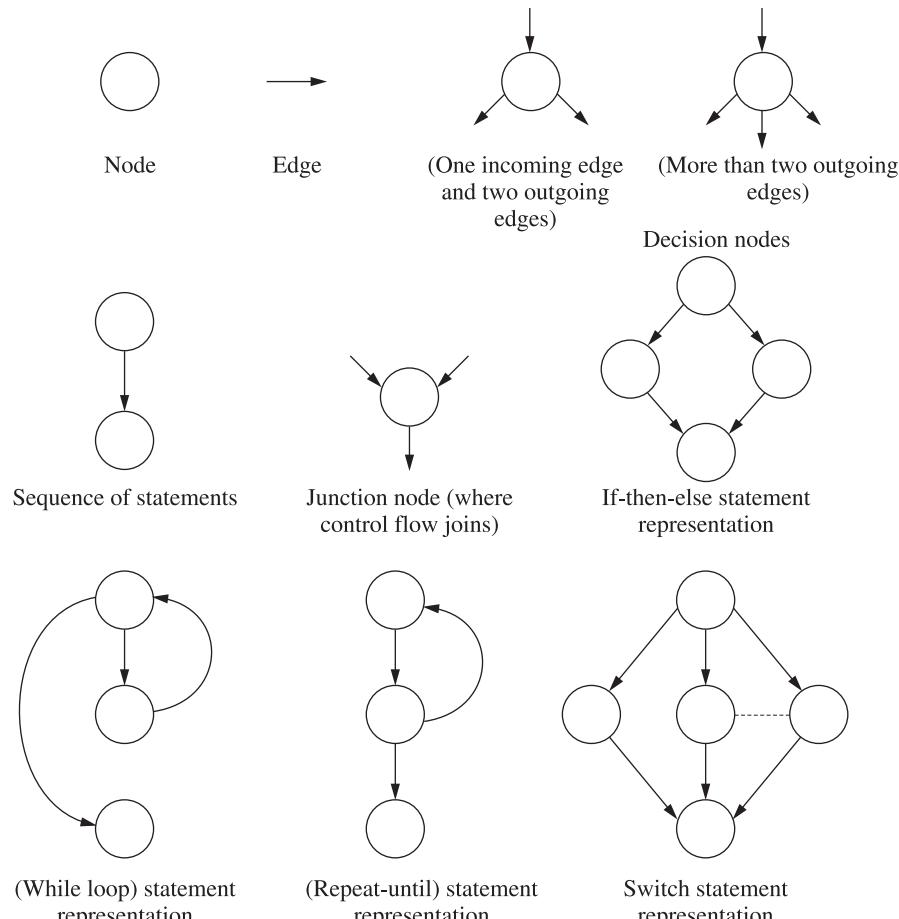


Figure 9.7 Basic constructs of a program graph.

A program can be converted into a program graph using fundamental constructs. We consider a program to determine whether a number is even or odd. The program is given in Figure 9.8 and its program graph is given in Figure 9.9. There are 16 statements in the program and hence 16 nodes in the program graph corresponding to every statement.

```

#include<stdio.h>
#include<conio.h>
1 void main()
2 {
3     int num;
4     clrscr() ;
5     cout<<"Enter number" ;
6     cin>>num ;

```

Figure 9.8 (Contd.)

```

7     if(num%2==0)
8     {
9       cout<<"Number is even";
10    }
11   else
12   {
13     cout<<"Number is odd";
14   }
15   getch();
16 }
```

Figure 9.8 Program to determine whether a number is even or odd.

A path in a graph is a sequence of adjacent nodes where nodes in sequence share a common edge or sequence of adjacent pair of edges where edges in sequence share a common node. It is clear from the program graph that there are two paths in the graph.

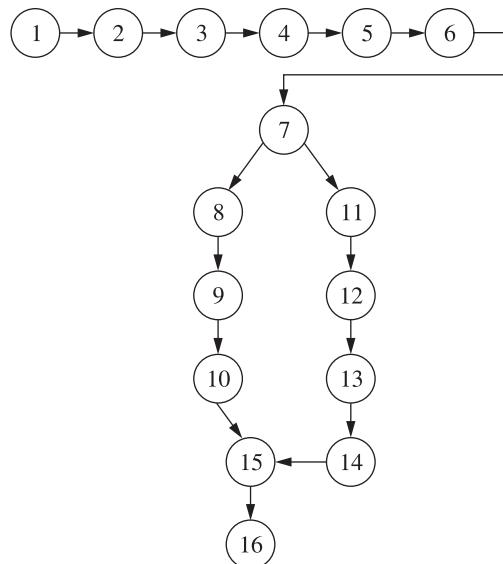


Figure 9.9 Program graph of source code given in Figure 9.8.

These paths are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16 and 1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16. Every program graph has one source node and one destination node. In the program graph given in Figure 9.9, nodes 1 to 6 are in sequence, node 7 has two outgoing edges (predicate node) and node 15 is a junction node.

A program graph can be converted into a DD program graph. There are many nodes in the program graph which are in a sequence like nodes 1 to 6 of the program graph given in Figure 9.9. When we enter into the sequence through the first node, we can exit only from the last node of the sequence. In the DD path graph, all nodes which are in a sequence are combined and represented by a single node. The DD path graph is a directed graph in which nodes are the

sequence of statements and edges are the control flow amongst the nodes. All DD path graphs have a source node and a destination node similar to the program graph. A mapping table is prepared to match the nodes of the DD path graph to the corresponding nodes of the program graph. All sequential nodes are combined into a single node which reduces the size of the DD path graph.

Mapping table of the program graph given in Figure 9.9 is presented in Table 9.30.

Table 9.30 DD path graph

S. No.	Program graph nodes	DD path graph corresponding nodes	Comments
1	1	S	Source node
2	2–6	N ₁	Sequential nodes
3	7	N ₂	Decision node
4	8–10	N ₃	Sequential nodes
5	11–14	N ₄	Sequential nodes
6	15	N ₅	Junction node
7	16	D	Destination node

The corresponding DD path graph is given in Figure 9.10. There are two paths, namely, S, N₁, N₂, N₃, N₅, D and S, N₁, N₂, N₄, N₅, D. Paths in the program are identified from the DD path graph easily. We may like to see the behaviour of the program when every identified path is executed. Hence, we may design test cases in such a way that at least every identified path is executed during testing.

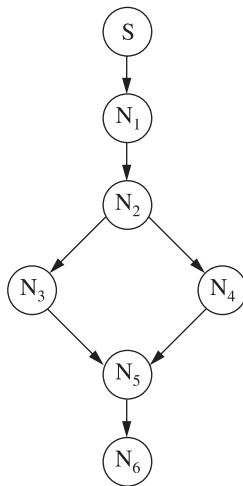


Figure 9.10 DD path graph.

Role of Independent Paths

The DD path graph is used to find independent paths. An independent path is a path through the DD path graph that introduces at least one new node or edge in its sequence from initial node to its final node. As we know, there are many paths in any program. If there are loops in the program (which is very common), the number of paths may increase and the same set of nodes

and edges may be traversed again and again. It may not be desirable to execute every path of any reasonable size program due to the repetition of the same set of statements and consumption of significant amount of time and resources. We may at least wish to execute every independent path of the DD path graph to ensure the minimum level of coverage of the source code and ascertain some confidence about the correctness of the program.

The number of independent paths can be found using a concept of graph theory which is called *cyclomatic number*. The same concept was redefined by McCabe (1976) as cyclomatic complexity for the identification of independent paths. There are three ways to calculate the cyclomatic complexity which are given below:

$$(i) \quad V(G) = e - n + 2P$$

where

$V(G)$ = cyclomatic complexity

G = graph

n = number of nodes

e = number of edges

P = number of connected components

The graph (G) is a directed graph with a single entry node and a single exit node. If this graph is a connected graph, the value of P will be 1. If there are parts of the graph, the value of P will be the number of parts of the graph.

$$(ii) \quad V(G) = \text{number of regions of the program graph}$$

$$(iii) \quad V(G) = \text{number of predicate nodes} (\pi) + 1$$

This is applicable only when the predicate node has two outgoing edges ("true" or "false"). If there are more than two outgoing edges, then this method is not applicable.

Properties of cyclomatic complexity:

- $V(G) \geq 1$.
- $V(G)$ is the maximum number of independent paths in the graph G .
- Addition or deletion of functional statements to graph G does not affect $V(G)$.
- G has only one path if $V(G) = 1$.
- $V(G)$ depends only on the decision structure of G .

The cyclomatic complexity of the DD path graph given in Figure 9.10 can be calculated as:

$$\begin{aligned} (i) \quad V(G) &= e - n + 2P \\ &= 7 - 7 + 2 \\ &= 2 \end{aligned}$$

$$(ii) \quad V(G) = \text{No. of regions of the graph}$$

Hence, $V(G) = 2$

One is the inner region constituted by nodes N_2 , N_3 , N_4 and N_5 and the second is the outer region of the graph.

$$\begin{aligned} (iii) \quad V(G) &= \pi + 1 \\ &= 1 + 1 = 2 \end{aligned}$$

The cyclomatic complexity is 2 which is calculated by all the three methods. Hence, there are two independent paths of this graph, namely, S, N_1 , N_2 , N_3 , N_5 , D and S, N_1 , N_2 , N_4 , N_5 , D.

Cyclomatic complexity also provides some insight into the complexity of a module. McCabe (1976) proposed an upper limit of cyclomatic complexity to 10 with significant supporting evidence. We may go up to 15 in today's scenario of new programming languages, availability of CASE tools, effective validation and verification techniques, and implementation of software engineering principles and practices. If the limit further exceeds, it is advisable to redesign the module and maintain the cyclomatic complexity within prescribed limits.

Issues in Path Testing

In practice, we should test every path of the program. However, this number may be too large in most of the programs due to the presence of loops and feedback connections. We may have to set a lower objective which may be set to execute at least all independent paths. We consider the program to determine “whether a number is even or odd” (given in Figure 9.8) along with its program graph (given in Figure 9.9). We find that there are two independent paths as given below:

Path 1: S, N₁, N₂, N₃, N₅, D

Path 2: S, N₁, N₂, N₄, N₅, D

We may design test cases in such a way that both paths are executed. The test cases are given in Table 9.31.

Table 9.31 Test cases for program given in Figure 9.8

S. No.	Path ID	Paths	Inputs	Expected output
1	Path 1	S, N ₁ , N ₂ , N ₃ , N ₅ , D	6	Number is even
2	Path 2	S, N ₁ , N ₂ , N ₄ , N ₅ , D	7	Number is odd

The program graph may generate a few paths which are impossible in practice. When we give inputs, some paths may not be possible to traverse due to logic of the program. Hence, some paths are impossible to create and traverse and cannot be implemented. Path testing ensures 100% statement and branch coverage and guarantees a reasonable level of confidence about the correctness of the program.

Generation of Paths Using Activity Diagram

We may also generate paths from the activity diagram. The details of an activity diagram are available in Chapter 7 (refer to Section 7.1). Activity diagram represents the flow of activities and is similar to the program graph. It may be generated from the use cases or from the classes. We may convert a source code into its activity diagram and may find the number of independent paths. The concept of cyclomatic complexity is still applicable with slight modifications. Nodes of the program graph are represented as branches/activities/initial state/end state of an activity diagram. The edges of the program graph are represented as transitions of the activity diagram. Cyclomatic complexity is modified as

$$\text{Cyclomatic complexity} = \text{Transitions} - \text{Activities/branches} + 2P$$

The cyclomatic complexity of the activity diagram given in Figure 7.2 is calculated as:

$$\text{Cyclomatic complexity} = 8 - 8 + 2 = 2$$

There are two independent paths which is also evident from the activity diagram.

After the identification of independent paths, we design test cases to execute these paths. This ensures that every transition and every activity/branch of an activity diagram are traversed, at least once, during execution. An activity diagram provides the pictorial view of the class and helps us to identify independent paths. This technique is effectively applicable to a class of any reasonable size.

EXAMPLE 9.8 Write a program for finding the roots of a quadratic equation. Draw the activity diagram and calculate the cyclomatic complexity. Generate the test cases on the basis of cyclomatic complexity.

Solution The program for finding the roots of a quadratic equation is given below:

```

#include<stdio.h>
#include<conio.h>

1 void main()
2 {
3     int a,b,c,validInput=0,d;
4     clrscr();
5     cout<<"Enter values of a, b & c:\n";
6     cin>>a>>b>>c;
7     if((a>=0)&&(a<=100)&&(b>=0)&&(b<=100)&&(c>=0)&&(c<=100)){
8         validInput=1;
9         if(a==0){
10             validInput=-1;
11         }
12     }
13     if(validInput==1){
14         d=b*b-4*a*c;
15         if(d==0){
16             cout<<"Equal roots";
17         }
18         else if(d>0){
19             cout<<"Real roots";
20         }
21         else{
22             cout<<"Imaginary roots";
23         }
24     }
25     else if(validInput==-1){
26         cout<<"Not quadratic";
27     }
28     else {
29         cout<<"The inputs are out of range";
30     }
31     getch();
32 }
```

The activity diagram for finding the roots of a quadratic equation is given in Figure 9.11.

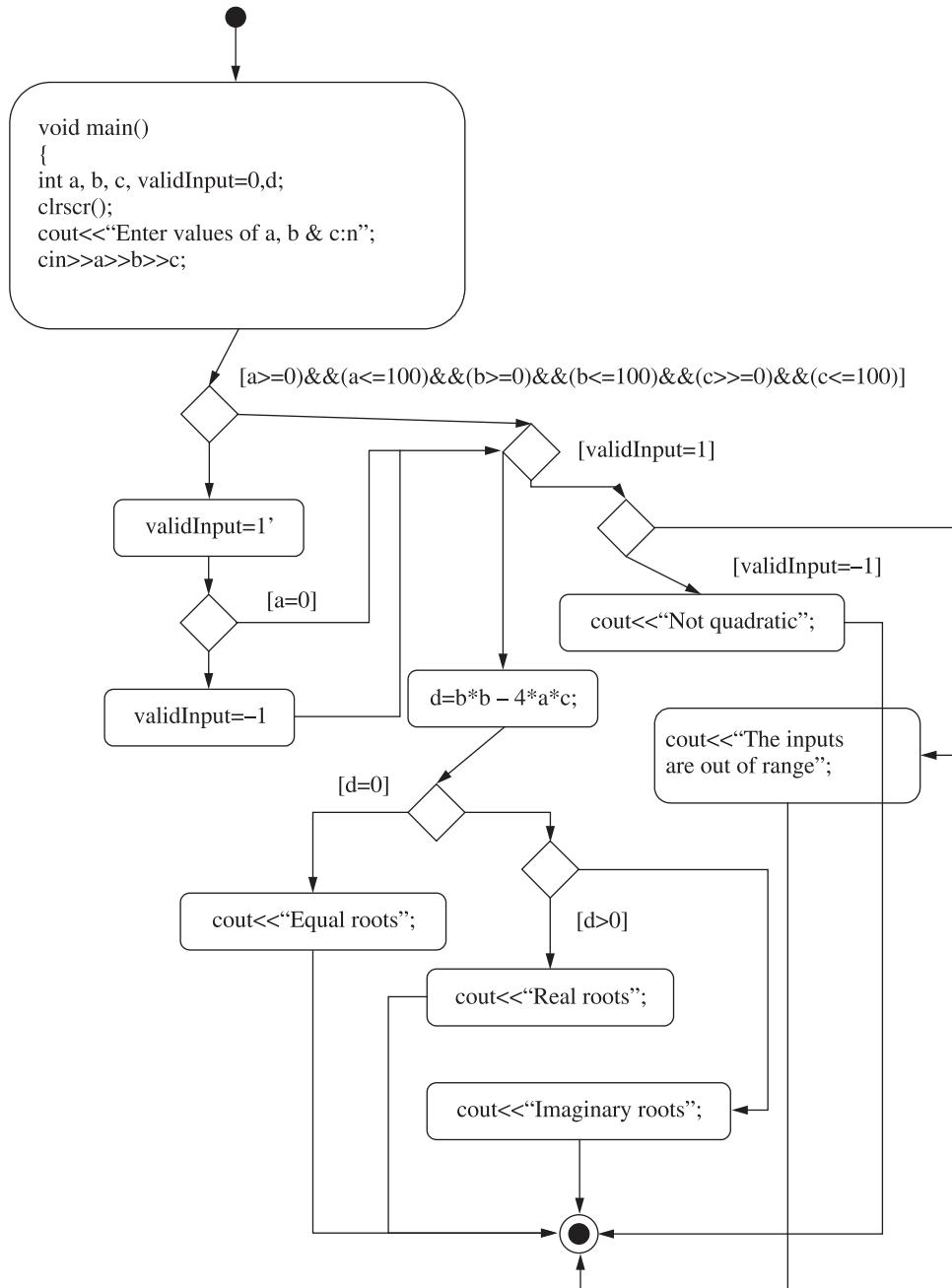


Figure 9.11 Activity diagram for finding the roots of a quadratic equation.

Cyclomatic complexity = $22 - 17 + 2 = 7$

The test cases for independent paths are given in Table 9.32.

Table 9.32 Test cases for independent paths

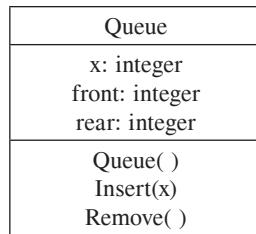
S. No.	a	b	c	Expected output
1	101	50	50	Input values not in range
2	-	-	-	-
3	-	-	-	-
4	0	50	50	Not quadratic
5	99	0	0	Equal roots
6	50	50	1	Real roots
7	50	50	50	Imaginary roots

9.6 Class Testing

A class is a fundamental entity in object-oriented software development activities. Developers define the attributes and operations of a class with utmost care. Operations are also known as methods of a class. The class is treated as a unit which is similar to the module/function of conventional programming. During testing, we may like to generate test cases on the basis of structure of a class. Therefore, implementation of a class is tested with respect to its specifications. We create instances of a class for the purpose of testing and test the behaviour of such instances. The classes cannot be tested in isolation. We may have to write additional source code which is too expensive and consumes huge amount of effort. In such situations, only inspections of classes are recommended which may find faults. Classes are usually tested by their developers due to their familiarity and understandability of the source code. Developers write additional source code (stubs and drivers) and make the classes executable. We test classes with respect to their specifications. If some unspecified behaviour is implemented, we may not be able to test it. Therefore, unspecified behaviour should never be implemented. If it is necessary to implement, changes should first be made in the SRS document and subsequently in analysis and design documents. The systematic approach may help us to know the deviations and incorporate them at proper places without any confusion. Hence, a disciplined and systematic procedure may generate test cases for additional specifications and further test them accordingly.

The most popular way to design test cases is from pre- and postconditions as specified in the use cases. Every operation of a class may have pre- and postconditions. An operation (method) will be initiated only when a precondition is true. After the successful execution of the operation, one or more postconditions may be generated. In class testing, we identify pre- and postconditions for every operation of the class and establish logical relationships amongst them. We should establish logical relationships when a precondition is true and also when a precondition is false. Every logical relationship (true and false) will generate a test case and finally we may be able to test the behaviour of a class for every operation and all in also possible conditions.

We consider a class ‘queue’ as given in Figure 9.12 with three attributes (x, front, rear) and three operations (queue(), insert(x), remove()).

**Figure 9.12 Structure of the class ‘queue’.**

We identify all pre- and postconditions for every operation of class ‘queue’. There are three operations and their pre- and postconditions are given as:

- (i) Operation Queue: The purpose of this operation is to create a queue
Queue::queue()
 (a) Pre = true
 (b) Post: front = 0, rear = 0
- (ii) Operation Insert(x): This operation inserts an item x into the queue.
Queue::insert(x)
 (a) Pre: rear < MAX
 (b) Post: rear = rear + 1
- (iii) Operation Remove(): This operation removes an item from the front side of the queue.
Queue::remove()
 (a) Pre: front > 0
 (b) Post: front = front + 1

We establish logical relationships between identified pre- and postconditions for insert(x) and remove() operations.

Insert(x):

1. (precondition: rear < MAX; postcondition: rear = rear + 1)
2. (precondition: not (rear < MAX); postcondition: exception)

Similarly for remove() operation, the following logical relationships are established:

3. (precondition: front > 0; postcondition: front = front + 1)
4. (precondition: not (front > 0); postcondition: exception)

Test cases are generated for every established logical relationship of pre- and postconditions. Test cases for insert(x) and remove() operations are given in Table 9.33.

Table 9.33 Test cases for two operations of class queue

S. No.	Operation	Test input	Condition	Expected output
1	Insert(x)	23	Rear < MAX	Element ‘23’ inserted successfully
2	Insert(x)	34	Rear = MAX	Stack overflow
3	Remove()	-	Front > 0	23
4	Remove()	-	Front = rear	Stack underflow

9.7 State-Based Testing

Statechart diagrams (given in Chapter 7) are used in state-based testing. As explained earlier, they model the behaviour of an object from its creation to the end of its life cycle. The flow of control from one state to another state is also represented graphically. Binder (1994) has recommended two additional states α (alpha) and ω (omega) for representing constructor and destructor of a class. He has emphasized his concept as:

The α state is a null state representing the declaration of an object before its construction. It may accept only a constructor, new, or a similar initialization message. The ω state is reached after an object has been destructed or deleted, or has gone out of scope. It allows for explicit modeling and systematic testing of destructors, garbage collection, and other termination actions.

We should not confuse with the start state and end state of the statechart diagram. α and ω are additional states and are represented explicitly in the statechart diagram. We consider an example of a class ‘queue’ with two operations—insert and remove. The queue is based on the principle of FIFO (first in first out) and there are three states: empty, holding and full. In addition to these three states, two additional states α and ω are also used. There are four events, namely, new, insert, remove and destroy with the following objectives:

- *New:* Creates an empty queue.
- *Insert:* Inserts an element in the queue, if space is available.
- *Remove:* Removes an element from the queue, if it is available.
- *Destroy:* Destroys the queue after the completion of its requirement, i.e. object of the queue class is destroyed.

We may draw the statechart diagram of the queue class as given in Figure 9.13.

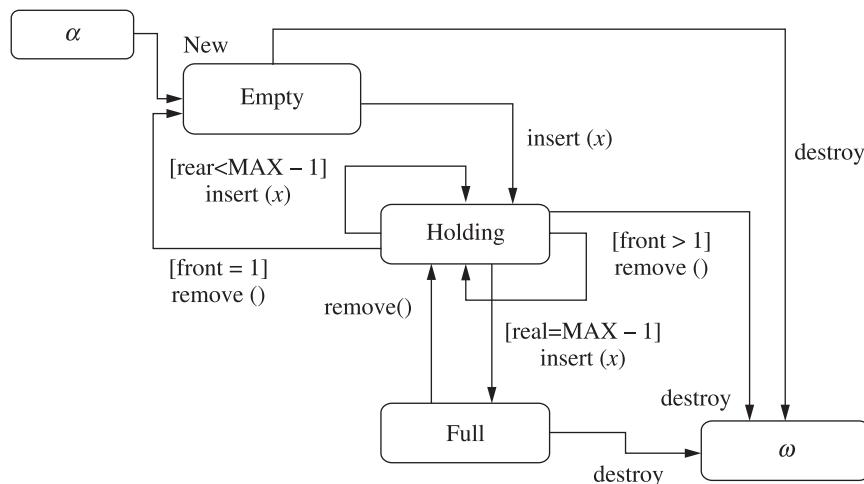


Figure 9.13 Statechart diagram of queue class.

The statechart diagram gives the graphical view of the behaviour of the system. All transitions are shown along with self-loop type transitions. In the above statechart diagram, the holding state has two self-loops for the following purposes:

- (i) When $[rear < MAX - 1]$, we insert an element x and the same state is maintained.
- (ii) When $[front > 1]$, we remove an element, the same state is maintained.

If the number of states increases, it becomes difficult to draw the statechart diagram and keep track of various transitions. In practice, the number of states is very large (more than 100 also) and it may not be possible to draw a statechart diagram. Another option is to represent the statechart diagram's information in tabular form and this table is known as *state transition table*. The row of the table represents the present state and the column represents the resultant state. The state transition table of 'queue' class is given in Table 9.34. The state transition table incorporates every information of the statechart diagram such as transitions, events and actions, and may become the basis to design the test cases.

Table 9.34 State transition table for queue class

State	Event/method	Resultant state				
		α	Empty	Holding	Full	ω
α	new		✓			
	insert(x)					
	remove()					
	destroy					
Empty	new					
	insert(x)					
	remove()					
	destroy					✓
Holding	new					
	insert(x)			✓	✓	
	remove()		✓	✓		
	destroy					✓
Full	new					
	insert(x)					
	remove()			✓		
	destroy					✓

9.7.1 Design of Test Cases

The state transition diagrams are used to identify paths and test cases may be generated for the execution of every path. There may be a large number of paths and it may not be advisable to execute all transitions which will further cover all events, all states and all actions. Statechart diagrams and statechart tables help us to achieve this objective and generate a good number of

test cases. The test cases for ‘queue’ class are given in Table 9.35 which are based on the state transition table as given in Table 9.34.

Table 9.35 Test cases

Test case ID	Test case input		Expected result	
	Event (method)	Test condition	Action	State
1.1	New			Empty
1.2	Insert(x)			Holding
1.3	Remove()	Front = 1	Return x	Empty
1.4	Destroy			∅
2.1	New			Empty
2.2	Insert(x)			Holding
2.3	Remove()	Front > 1	Return x	Holding
2.4	Destroy			∅
3.1	New			Empty
3.2	Insert(x)			Holding
3.3	Insert(x)	Rear < MAX – 1		Holding
3.4	Destroy			∅
4.1	New			Empty
4.2	Insert(x)			Holding
4.3	Insert(x)	Rear = MAX – 1		Full
4.4	Remove()			Holding
4.5	Destroy			∅
5.1	New			Empty
5.2	Insert(x)			Holding
5.3	Insert(x)	Rear = MAX – 1		Full
5.4	Destroy			∅
6.1	New			Empty
6.2	Destroy			∅

These test cases are systematic and cover every functionality of the ‘queue’ class. The state-based testing is simple, effective and systematically generates a good number of test cases.

9.8 Mutation Testing

Mutation testing is a useful testing technique to determine the effectiveness of an existing test suite of any program. Generally, it is applicable at the unit level, but we can also apply it at the system level without any difficulty. Why do we need such a technique? Why do we want to understand the adequacy of our test cases? As we all know, there may be a large number of test cases for any program. We normally do not execute large-size test suite completely due to

time and resource constraints. We attempt to reduce the size on the basis of some techniques. However, if we execute all test cases and do not find any fault in the program, there are the following possibilities:

- (i) Test suite is effective, but there are no faults in the program.
- (ii) Test suite is not effective, although there are faults in the program.

In both possibilities, we could not make our program fail, although reasons of non-failure of the program are different. In the first case, the quality of the program is good and there are no faults in the program. However, in the second case, there are faults in the program but test cases are not able to find them. Hence, mutation testing helps us to know the effectiveness of an existing test suite with reasonable level of accuracy.

9.8.1 Mutation Testing and Mutants

We prepare many copies of the program and make a change in every copy. The process of making change in the program by one or more changes is called *mutation* and the changed program is called a *mutant*. Each mutant is different than the original program by one or more changes. This change should not make the program grammatically incorrect. The mutant must be compiled and executed in the same way as the original program. A change should be a logical change in the program. We may change an arithmetic operator (+, -, *, \) or may change a Boolean relation with another one (replace > with >= or == with <=). There are many ways to make a change in the program till it is syntactically correct. Each mutant will have a unique change, which will make it different from other mutants. Consider a program to find the smallest amongst three numbers as given in Figure 9.14. The two mutants of the same program are also given in Figures 9.15 and 9.16.

```
#include<iostream.h>
#include<conio.h>
1 class smallest
2 {
3     private:
4         int a,b,c;
5     public:
6         void getdata()
7     {
8         cout<<"Enter first number: ";
9         cin>>a;
10        cout<<"\nEnter second number: ";
11        cin>>b;
12        cout<<"\nEnter third number: ";
13        cin>>c;
14    }
```

Figure 9.14 (Contd.)

```
15     void compute();
16 };
17 void smallest::compute()
18 {
19     if(a<b)
20     {
21         if(a<c)
22         {
23             cout<<"\nThe smallest number is:"<<a;
24         }
25     Else
26     {
27         cout<<"\nThe smallest number is:"<<c;
28     }
29 }
30 Else
31 {
32     if(c<b)
33     {
34         cout<<"\nThe smallest number is:"<<c;
35 }
36 Else
37 {
38     cout<<"\nThe smallest number is:"<<b;
39 }
40 }
41 }
42 void main()
43 {
44     clrscr();
45     smallest s;
46     s.getdata();
47     s.compute();
48     getch();
49 }
```

Figure 9.14 Program to find the smallest amongst three numbers.

The mutant M₁ is created by replacing the operator ‘<’ of line number 19 by the operator ‘=’. The mutant M₂ is created by replacing the operator ‘<’ of line number 32 by the operator ‘>’. These

changes are simple and both mutants are different than the original program. Both mutants are also different than each other but are syntactically correct. The mutants are known as first-order mutants. We may also get second-order mutants by making two changes in the program and third-order mutants by making three changes and so on. The second-order and above mutants are called higher-order mutants. However, in practice, we generally use only the first-order mutants in order to simplify the process of mutation. The second-order mutants can be created by making two changes in the program as shown in Figure 9.17.

```
19 if(a<b) ← if(a=b) //mutated statement where '<' is replaced by '='
```

Figure 9.15 Mutant (M_1) of program to find the smallest amongst three numbers.

```
32 if(c<b) ← if(c>b) //mutated statement where operator '<' is  
replaced by operator '>'
```

Figure 9.16 Mutant (M_2) of program to find the smallest amongst three numbers.

```
19 if(a<b) ← if(a>b) //mutated statement where '<' is replaced  
by '>'
```

```
32 if(c<b) ← if(c=b) //mutated statement where operator '<' is  
replaced by operator '='
```

**Figure 9.17 Second-order mutant of program to find the
smallest amongst three numbers.**

9.8.2 Mutation Operators

We use mutation operators to create mutants. Operators are used to change a grammatical expression to another expression without making the changed expression incorrect as per syntax of the implementation language. There are many ways to apply mutation operators and a large number of mutants may be created. If an expression ‘ $a+6$ ’ is changed to ‘ $a+10$ ’, it is considered as a lesser change as compared to an expression ‘ $b*6$ ’, where both operator and operands are changed. The first order mutants are more popular in practice as compared to the higher-order mutants because they are easy to understand, implement, manage and control. Some of the examples of mutation operators are given as follows:

- (i) access modifier change like private to public
- (ii) static modifier change
- (iii) Change of arithmetic operator with another one like ‘ \wedge ’ with ‘ $*$ ’ or ‘ $!$ ’ with ‘ $-$ ’
- (iv) Change of Boolean relation with another one like ‘ $<$ ’ with ‘ $>$ ’ or ‘ $=$ ’ with ‘ $==$ ’
- (v) Delete a statement
- (vi) Change of argument order
- (vii) Change of any operand by a numeric value
- (viii) Type case operator insertion
- (ix) Type case operator deletion
- (x) Change of super keyword

9.8.3 Mutation Score

We execute mutants with existing test suite and observe the behaviour. If the existing test suite is able to make a mutant fail (any observed output is different than the actual output), the mutant is treated as a killed mutant. If the test suite is not able to kill the mutant, the mutant is treated as equivalent to the original program and such mutants are called *equivalent mutants*. Equivalent mutants are also called *live mutants*. The mutation score of the test suite is calculated as:

$$\text{Mutation score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

The total number of mutants is equal to the number of killed mutants plus the number of live mutants. A mutation score range is from 0 to 1. The higher value of a mutation score shows the effectiveness of the test suite. If the mutation score is 0, it indicates that the test suite is not able to detect any introduced fault in mutants and is useless for the program. The live mutants are important and should be studied thoroughly. Special test cases should be designed to kill live mutants. The new test cases which are able to kill live mutants should be added to the original test suite to enhance its capability. The mutation testing not only assesses the capability of the test suite but also enhances the capability by adding new test cases. Some of the popular tools are insure++, Jester for Java, nester for C++, MuJava tool, Mothra, etc.

We consider the program given in Figure 9.14 to find the smallest of three numbers. Table 9.36 shows the test suite available to test the program.

Table 9.36 Test suite to test the program given in Figure 9.14

S. No.	A	B	C	Expected output
1	3	1	5	1
2	6	2	8	2
3	4	5	7	4
4	8	6	3	3
5	9	10	6	6

Six mutants are created as per details given in Table 9.37.

Table 9.37 Mutants generated

Mutant No.	Line No.	Original line	Modified line
M_1	19	if(a<b)	if(a=b)
M_2	32	if(c<b)	if(c>b)
M_3	21	if(a<c)	if(a!=c)
M_4	32	if(c<b)	if(c=b)
M_5	19	if(a<b)	if(a>b)
M_6	32	if(c<b)	if(c<(a+b))

The actual output of all mutants with existing test suite is given in Table 9.38.

Table 9.38 Actual output of all mutants

S. No.	A	B	C	Expected output	Actual output of M ₁	Actual output of M ₂	Actual output of M ₃	Actual output of M ₄	Actual output of M ₅	Actual output of M ₆
1	3	1	5	1	1	5	1	1	3	1
2	6	2	8	2	2	8	2	2	6	2
3	4	5	7	4	5	4	4	4	5	4
4	8	6	3	3	3	6	3	6	3	3
5	9	10	6	6	6	6	9	6	6	6

The existing test suite kills five mutants (M₁ to M₅), but fails to kill mutant M₆. Hence, a mutation score is calculated as given below:

$$\begin{aligned} \text{Mutation score} &= \frac{\text{Number of mutants killed}}{\text{Total number of mutants}} \\ &= \frac{5}{6} \\ &= 0.83 \end{aligned}$$

Effectiveness of a test suite is directly related to the mutation score. Mutant M₆ is live here and an additional test case is to be written to kill this mutant. The additional test case is created to kill this mutant M₆ and is given in Table 9.39.

Table 9.39 Additional test case to kill mutant M₆

S. No.	A	B	C	Expected output
6	6	4	5	4

When we execute this test case, we get the observed behaviour of the program given in Table 9.40.

Table 9.40 Observed behaviour of the program

S. No.	A	B	C	Expected output	Actual output
6	6	4	5	4	5

This additional test case is very important and must be added to the given test suite. Therefore, the revised test suite is given as in Table 9.41.

Table 9.41 Revised test suite

S. No.	A	B	C	Expected output
1	3	1	5	1
2	6	2	8	2
3	4	5	7	4
4	8	6	3	3
5	9	10	6	6
6	6	4	5	4

9.9 Levels of Testing

There are four levels in testing, namely, unit testing, integration testing, system testing and acceptance testing. Software testers are responsible for the first three levels of testing and customers are responsible for the last level (acceptance testing) of testing. Testing at each level is important and has unique advantages and challenges. At the unit level, individual units are tested using functional and/or structural testing technique. At the integration level, two or more units are combined to test the issues related to integration of units. At the system level, the complete system is tested using primarily functional testing techniques. Non-functional requirements such as reliability, testability, performance, etc. can be tested at this level only. At the acceptance level, customers test the system as per their expectations. Their testing strategy may range from ad hoc testing to well-planned systematic testing.

9.9.1 Unit Testing

There are two ways to define a unit in object-oriented testing. We may consider either each class as a unit or each operation of the class as a unit for the purpose of unit testing. How can we test a class at the unit testing level which has a parent class? The operations and attributes of the parent class will not be available which will prohibit class testing. The solution is to merge the parent class and the class under test which will make all operations and attributes available for testing. This type of merging is called *flattening of classes*. We may test such classes after flattening. After completion of testing, we should redo flattening because the final product should not have flattened classes. The issues of inheritance are to be handled carefully. If we decide to select a method as a unit, these issues will be more challenging and difficult to implement. Hence, in practice, classes are generally used as a unit at the level of unit testing.

We create instance of class (object) and pass the desired parameters to the constructor. We may also call each operation of the object with appropriate parameters and note the actual outputs. The encapsulation is important because attributes and operations are combined in a class. We focus on the encapsulated class; however, operations within the classes are the smallest available units for testing. Operations as a unit are difficult to test due to inheritance and polymorphism. In unit testing, generally, classes are treated as unit, and functional and structural testing techniques are equally applicable. Verification techniques such as peer reviews,

inspections and walkthroughs are easily applicable and may find a good number of faults. State-based testing, path testing, class testing, boundary value analysis, equivalence class and decision table-based testing techniques are also applicable.

9.9.2 Integration Testing

The objective of integration testing is to test the various combinations of different units and to check that they are working together properly. We do not have hierarchical control structure in object-oriented systems. Hence, conventional integration testing techniques such as top down, bottom up and sandwich integration may not be applicable in their true sense. The meaning of integration testing is basically interclass testing. There are three ways to carry out interclass testing. The most popular interclass testing is thread-based testing. In the thread-based testing, we integrate classes that are required to respond to an input given to the system. When the input is given to the software, one or more classes are needed for execution, and such classes make a thread. There may be many such threads depending on the inputs. The expected output of every thread is calculated and is compared with the actual output. This technique is simple and easy to implement.

The second technique is the use case-based testing. We test every basic and alternative paths of a use case. A path may require one or more classes for execution. Every use case scenario (path) is tested and due to involvement of many classes, interclass issues are automatically tested. The third technique is the cluster testing where classes are combined to show one collaboration. In all approaches, classes are combined on the basis of logic and then executed to know the outcome. The most popular and simple technique is the thread-based testing.

9.9.3 System Testing

The system testing is performed after the unit testing and integration testing. The complete software within its expected environment is tested. We define a system as a combination of software, hardware and other associated parts which work together to provide the desired functionality. All functional testing techniques are effectively applicable. Structural testing techniques may also be used technically but they are not very common due to the large size of the software. Verification techniques are normally used for reviewing the source code and documents. We test the functional requirements of the software under stated conditions. This is the only level where non-functional requirements such as stress, load, reliability, usability and performance are tested. A good number of testing tools are available to test the functional and non-functional requirements.

We may like to ensure a reasonable level of correctness of the software before delivering it to the customer. Whenever the source code is modified to remove an error, an impact analysis of this modification is performed. If any error is not possible to remove due to lack of time or is technically not possible in the present design, the best way is to document the error as a limitation of the system. We would like to test every stated functionality of the software, keeping in mind, the customer's expectations. After completion of the system testing, the software is ready for customers.

9.9.4 Acceptance Testing

The acceptance testing is carried out by the customer(s) or their authorized persons for the purpose of accepting the software. The place of testing may be the developer's site or the customer's site depending on the mutual agreement. In practice, generally, it is carried out at the customer's site. Customers may like to test the software as per their expectations. They may do it in an ad hoc way or in a well-planned systematic way in order to establish the confidence about the correctness of the software.

When a software product is developed for anonymous customers (in case of operating systems, compilers, CASE tools, etc.), potential customers are identified to use the software as per their expectations. If they use the software at the developer's site under the supervision of the developers, it is known as *alpha testing*. Another approach is to distribute the software to potential customers and ask them to use at their site in a free and independent environment and this is known as *beta testing*. The purpose of the acceptance testing is to test the software with an intention to accept it after getting reasonable confidence about its usage and correctness.

9.10 Software Testing Tools

Software testing tools are available for various applications. They help us to design and execute test cases, analyse the program complexity, identify the non-coverage area of the source code and ascertain the performance of the software. There are numerous similar applications during testing which may be improved using a testing tool. The whole process of testing a software may be automated and carried out without human involvement. Software tools are also very effective for repeated testing where similar data set is to be given again and again. Many non-functional requirements such as performance under load, efficiency, reliability and extreme stress conditions are also tested using software testing tools. Broadly, these tools may be partitioned into three categories—static, dynamic and process management. Most of the tools may fall into any one of the categories and have specified scope with predefined applications.

9.10.1 Static Testing Tools

Static testing tools analyse the program without executing it. They may calculate program complexity and also identify those portions of the program which are hard to test and maintain. These tools may find a good number of faults prior to the execution of the program. The identified faults may range from logical faults to syntax faults and may include non-declaration of a variable, double declaration of a variable, divide by zero issue, unspecified inputs, etc. Some tools may also examine the implementation of good programming guidelines and practices and highlight the violations, if any, to improve the quality of the program. Many tools which calculate the metrics are static analysis tools. Some of the popular tools are CMTJava (Complexity Measures Tool for Java), Jenssoft's Code Companion, Sun Microsystems' JavaPureCheck, ParaSoft's Jtest, Rational Purify, CMT++ (Complexity Measures Tool for C++), ParaSoft CodeWizard, ObjectSoftware's ObjectDetail, Software Research's STATIC (Syntac and Semantic Analysis Tool), Eastern System's TestBed, McCabe QA, etc.

9.10.2 Dynamic Testing Tools

Dynamic testing tools execute the programs for specified inputs. The observed output is compared with the expected output and if they are different, the program is considered in a failure condition. These tools also analyse the programs and the reasons of such failure may also be found. Dynamic testing tools are also very effective to test the non-functional requirements such as performance, reliability, efficiency and portability.

Performance Testing Tools

These tools are used to test the performance of the software under stress and load. The performance testing is also called stress and load testing. Popular tools are Mercury Interactive's Load Runner, Apache JMeter, Rational's Performance Tester, Compuware's QALOAD, Auto Tester's Autocontroller, Quest Software's Benchmark Factory, Sun Microsystems' Java Load, Minq Software's PureLoad, Rational Software's Test Studio, etc. These tools generate heavy load on the system to test under extreme conditions.

Functional/Regression Testing Tools

These tools are used to test the functionality of the software. They may generate test cases and execute them without human involvement. In regression testing, the software is retested after modifications. Most of the tools are common for both groups. Some of the popular tools are Junit, Test Manager, Rational's Robot, Mercury Interactive's Win Runner, Compuware's QA Centre, Segue Software's Silktest, AutoTester for Windows, Qronus Interactive's TestRunner, Automated QA's AQtest, Rational's Visual Test, etc.

Coverage Analysis Tools

These tools are used to provide an idea about the level of coverage of the program. They also indicate the effectiveness of the test cases. They may also highlight the untested portion of the program which may help us to design special test cases for the coverage of that portion of the program. Some popular source code coverage analysis tools are Rational's Pure Coverage, Quality Checked Software's Cantata++, CentreLine Software's QC/coverage, Vision Soft's Vision Soft, Plum Hall's SQS, etc.

Some popular test coverage analysis tools are Software Research's TCAT for Java, IBM's Visual Test Coverage, Bulseye Testing Technology's C-Cover, Testwell's CTC++, Testing Foundation's GCT, Parasoft's TCA, Software Research's TCAT C/C++, and McCabe's Visual Testing Tool Set.

9.10.3 Process Management Tools

The focus of process management tools is to improve the testing processes. They may help us to allocate resources, prepare test plan and keep track of the status of testing. Some of the popular tools are IBM Rational Test Manager, Mercury Interactive's Test Director, Segue Software's Silk Plan Pro, Compuware's QA Director, etc. Some configuration management tools are IBM Rational Software clear DDTs Bugzilla, Samba's Jitterbug. A few test management tools are

Vector Software's Vector CAST, Auto Tester's Auto Advisor, Silver Mark's Test Mentor, Test Master's TMS and TOOTSIE.

A software reliability measurement tool is used to estimate the reliability of the software. It may also calculate the time needed to achieve an objective failure intensity. A popular tool is SoftRel's WhenToStop. Software testing tools not only reduce the testing effort but also make testing a pleasant discipline. Moreover, some non-functional requirements (such as performance and efficiency) cannot be tested without using a software testing tool.

Review Questions

1. What is software testing? Discuss issues, limitations, practices and future of software testing.
2. “Testing is not related to only one phase of software development life cycle”. Comment on the correctness of this statement.
3. Differentiate between verification and validation. Which one is more important and why?
4. Which is the most popular verification technique? Explain with suitable examples.
5. Write short notes on the following verification techniques:
 - (a) Peer reviews
 - (b) Walkthroughs
 - (c) Inspections
6. Design an SRS document verification checklist. Highlight some important issues which such a checklist must address.
7. What is a checklist? Discuss its significance and role in software testing.
8. Differentiate between walkthroughs and inspections. List the advantages and disadvantages of both techniques.
9. Design checklists for OOA document and OOD document. Both checklists should cover important issues of both documents.
10. Establish the relationship between verification, validation and testing along with suitable examples.
11. What is functional testing? Discuss any one technique with the help of an example.
12. What is boundary value analysis? What are its extensions? List the advantages and limitations of each extension.
13. Consider a program that determines the previous date and the next date. Its inputs are a triple of day, month and year with its values in the following range:

$$1 \leq \text{month} \leq 12$$

$$1 \leq \text{day} \leq 31$$

$$1947 \leq \text{year} \leq 2011$$

The possible outputs are “next date”, “previous date” or “invalid input”. Design boundary value analysis test cases, robust test cases, worst test cases and robust worst test cases.

14. Consider a program that calculates median of three numbers. Its input is a triple of positive integers (say a , b and c) and values are from the interval [1, 1000]. Generate boundary value and robust test cases.
15. Describe the equivalence class testing technique. How is it different from boundary value analysis?
16. What is decision table-based testing? What is the role of a rule count? Discuss the concept with an example.
17. Consider a program to sort a list in ascending order. Write test cases using equivalence class testing and decision table-based testing.
18. What are the limitations of boundary value analysis? Highlight the situations in which it is not effective.
19. Consider a program that counts the number of digits in a number. The range of input is given as [1, 1000]. Design the boundary value analysis and equivalence class test cases.
20. What is structural testing? How is it different from functional testing?
21. What is path testing? How can we make it more effective and useful?
22. Show that a very high level of statement coverage does not guarantee a defect-free program. Give an example to explain the concept.
23. What is mutation testing? What is the significance of mutation score? Why are higher-order mutants not preferred?
24. Why is mutation testing becoming popular? List some significant advantages and limitations.
25. What is a program graph? How can it be used in path testing?
26. Define the following:
 - (a) Program graph
 - (b) DD path graph
 - (c) Mapping table
 - (d) Independent paths
27. What is class testing? Why should unspecified behaviour not be implemented?
28. What is state-based testing? What are alpha and omega states? List the advantages and limitations of the state-based testing.
29. What are levels of testing? Which level is most popular and why?
30. Explain the concept of flattening of classes. How can it help in unit testing?
31. Discuss the importance of software testing tools. List some static, dynamic and process management tools.
32. Differentiate between static and dynamic testing tools. Which one is more effective and why?
33. “Some non-functional requirements cannot be tested without using a software testing tools”. Comment on this statement and justify with an example.

34. Differentiate between black box testing and white box testing. Consider a program to find the smallest number amongst three numbers. Generate test cases using one of the testing techniques.

35. Write short notes on:

 - (a) Software testing tools
 - (b) State-based testing
 - (c) Class testing

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. What is the purpose of testing?
 - (a) To show the correctness of a program
 - (b) To calculate the reliability of a program
 - (c) To find faults in a program
 - (d) To know the limitations of a program
 2. Software testing primarily focuses on:
 - (a) Verification activities only
 - (b) Validation activities only
 - (c) Verification and validation activities
 - (d) None of the above
 3. All validation activities are related to:

(a) Static testing	(b) Dynamic testing
(c) Reviewing	(d) Inspecting
 4. All verification activities are related to:

(a) Static testing	(b) Dynamic testing
(c) Functional testing	(d) Structural testing
 5. Which is **not** a verification activity?

(a) Peer review	(b) Walkthrough
(c) Inspection	(d) Path testing
 6. Which is **not** a term for inspection?

(a) Peer review	(b) Review
(c) Technical review	(d) Formal technical review
 7. Who is the presenter of documents in walkthroughs?

(a) Customer	(b) User
(c) Author	(d) Developer
 8. Who is the presenter of documents in inspections?

(a) Customer	(b) Developer
(c) Third party moderator	(d) Author

33. Which is **not** a process management tool?
- (a) Mercury Interactive's Test Director (b) Compuware's QA Director
(c) Auto Tester's Auto Advisor (d) Rational's Robot
34. Identify the functional testing tool:
- (a) Rational's Robot (b) Test Master's TMS
(c) Apache JMeter (d) Mercury Interactive's Load Runner
35. Which is **not** a functional testing tool?
- (a) Rational's Robot (b) Mercury Interactive's Win Runner
(c) Seque Software's Silktest (d) Samba's Jitterbug

Further Reading

Fagan shows that by using inspection, the cost of errors may be reduced significantly in initial phases of software development:

Fagan, M.E., Design and code inspections to reduce errors in program development. *IBM Systems Journal*, **15**(3): 182–211, 1976.

Strauss et al. provide a comprehensive guide to software inspections method that may reduce program defects in the early phases of software design and development:

Strauss, S.H., Susan, H., and Ebeneau, Robert G., *Software Inspection Process*. New York: McGraw-Hill, 1994.

Yourdon's book may provide a useful guidance to practitioners and programmers on group walkthroughs:

Yourdon, E., *Structured Walkthroughs*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

The below book is resource for pre-1981 literature and contains a huge bibliography up to and including 1981:

Miller, E.F. and Howden, W.E., *Tutorial: Software Testing and Validation Techniques*. New York: IEEE Computer Society, 1981.

A hands-on guide to the black box testing technique:

Beizer, B., *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. New York: John Wiley & Sons, 1995.

An introductory book on software testing with a special focus on functions testing is:

Jorgensen, P.C., *Software Testing: A Craftsman Approach*, 3rd ed. New York: Auerbach Publications, 2007.

A useful guide for designing test cases for object-oriented applications. This book provides comprehensive and detailed coverage of techniques to develop testable models from unified modelling language and state machines:

Binder, R.V., *Testing Object Oriented Systems: Models, Patterns and Tools*. Reading, MA: Addison-Wesley, 1999.

Binder has significantly contributed in the area on object-oriented testing:

Binder, R., State-based testing. *Object Magazine*. 5(4): 75–78, July–Aug, 1995.

Binder, R., State-based testing: sneak paths and conditional transitions. *Object Magazine*, 5(6), 87–89, Nov–Dec, 1995.

Binder, R., *Testing Object-Oriented Systems: A Status Report*, Released online by RBSC Corporation, 1995. Available at <http://stsc.hill.af.mil/crosstalk/1995/April/testinoo.asp>

McGregor and Sykes provide good differences of testing traditional and object-oriented software. They also describe methods for testing of classes:

McGregor, J.D. and Sykes, David A., *A Practical Guide to Testing Object Oriented Software*. Canada: Addison–Wesley, 2001.

A complete bibliographical list on object-oriented testing can be obtained from:

<http://oo-testing.com/bib/>

A study on mutation analysis on object-oriented programs can be found in:

Ma, Y.S. and Kwon, Y.R., A study on method and tool of mutation analysis for object-oriented programs. *Software Engineering Review*, 15(2): 41–52, 2002.

10

Software Maintenance

When a software product is delivered to the customer and is made operational, that initiates the beginning of the maintenance phase of software development life cycle. The time spent and effort required to keep the software operational is always very significant which makes the maintenance phase expensive. We may not appreciate the cost of maintenance till we understand the maintenance activities clearly. The software maintenance phase consumes 40% to 70% of the total cost spent during the entire life of the software. We may take 2 to 3 years for development of software, which may have to be maintained for 10 to 15 years or even more, depending on the sustainability of the software product. The most challenging, important and expensive phase is generally treated as a headache and nobody wants to do the maintenance tasks happily. Moreover, issues and concerns in software increase for a object oriented software.

In this chapter, we describe maintenance-related activities, problems and models. We also address the difficulties involved in maintaining object-oriented software.

10.1 What is Software Maintenance?

The term maintenance has slightly different meaning than used in other fields. As we all know, software does not break or wear out like hardware and also does not have any particular physical shape. However, software may fail due to errors and may be modified for doing new things. Robert Glass has explained rightly as:

Software errors are not due to material fatigue, but rather to errors made when the software was being built or errors made as the software is being changed. So, software maintenance is about fixing those errors as they are discovered and making those changes as they become necessary.

The modifications for the purpose of error correction and enhancements for additional functionality consume significant effort which may account to more than 70% of the total maintenance effort.

The remaining effort (around 30% or less) may be consumed for making changes in the software due to changes in business rules, external condition, restructuring, source code optimization, etc. The maintenance activities are primarily focused on adding new functionalities to the old software. In general, any modification to software (source code, documentation and manuals) after its release to the customer is considered as a maintenance activity. The maintenance phase should be treated as a problem-solving, creative and positive phase of SDLC. Effective maintenance phase increases the life of the software product and makes the customers happy and ever smiling. As per IEEE (1990), software maintenance is defined as:

It is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes or adapt to a changed environment.

This definition clearly states that any changes in the software for any purpose after delivery come under software maintenance. It also emphasizes that the maintenance is not just about fixing faults in the software but also involves improvements in the source code and modifications due to changes in the environment.

10.2 Categories of Software Maintenance

There are four categories of software maintenance which are constituted on the basis of changes in the software. Every change has an objective and that objective decides the category of software maintenance. These categories are discussed in the following subsections.

10.2.1 Corrective Maintenance

It is related to removing faults from the software after the **failure is reported by the customer**. The purpose is to make the software operational as early as possible due to serious time constraints. Corrective maintenance may incorporate all types of changes made to software in order to remove faults. IEEE (1998) defines corrective maintenance as:

*Corrective maintenance is **reactive modification** of a software product performed after delivery to correct discovered faults.*

Many people feel that maintenance is related to corrective maintenance only. As reported by Lientz and Swanson (1980), corrective maintenance is only 21% of the total maintenance activities. However, it is a very important activity for the sustainability, reliability and reputation of the software product. The distribution of maintenance activities is given in Figure 10.1 which is based on the study carried out by Lientz and Swanson (1980).

10.2.2 Adaptive Maintenance

It is related to modifications made in the software in order to match changes in the environment

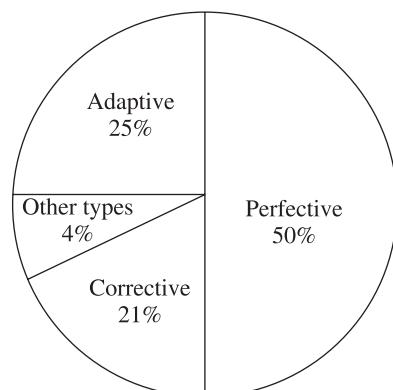


Figure 10.1 Distribution of maintenance activities.

in which the software operates. The environmental changes may include hardware, operating system, network or changes in the business rules, government policies, etc. A change to the whole or part of the environment requires a corresponding modification to the software (BR0087). As per IEEE (1998), adaptive maintenance is defined as:

*Adaptive maintenance is related to modification of a software product performed after delivery to keep a computer program **usable** in a changed or changing environment.*

We may have to make changes in the software due to introduction of a new operating system or new hardware or modifications in the existing operating system/hardware. We may also modify the software due to some changes in rules, policies or guidelines. When European Union had decided to go for a single European Currency ‘Euro’, all financial software products were modified to incorporate the new currency. As per Lientz and Swanson (1980), adaptive maintenance is 25% of the total maintenance activities.

10.2.3 Perfective Maintenance

It is related to the changes made on the requests of users of the software. The users may request for additional functionality or modifications in documents or improvement in performance or increase in ease of use, etc. Hence, requests of users for additional functionality, performance improvement, usability enhancement, documents and reports modifications, and table updations come under the category of perfective maintenance. IEEE (1998) defines perfective maintenance as:

*It is related to modification of a software product performed after delivery to **improve** performance or maintainability.*

The perfective maintenance activities are 50% of the total maintenance activities (Lientz and Swanson, 1980) and consume the significant amount of effort. In general, perfective maintenance includes activities to make better product, improve performance, provide more functions, update reports and documents, enhance usability, and improve the computational efficiency. Most of the times, the user’s requests are confined to additional functionality and modifications in the tables and reports.

10.2.4 Other Types of Maintenance Activities

There is an adverse effect on the structure of the software due to continuous changes in the maintenance phase. The complexity of the software may increase and understandability may decrease which make the maintenance more difficult and challenging. Any modifications for the improvement of maintainability which will enhance the sustainability of the software come under “other types of maintenance” category. Some authors have named it as “preventive maintenance” which is carried out for the purpose of reducing complexity, improving maintainability and enhancing the understandability of the source code and documents. Canfora and Cimitile (2000) have expressed their concerns as:

Ideally, maintenance operations should not degrade the reliability and the structure of the software, neither they should degrade its maintainability, otherwise future changes will be progressively more difficult and costly to implement.

The maintainability is defined as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes or adapt to a changed environment (IEEE, 1998).

Hence, any modifications related to improve the maintainability come under this category of maintenance. These activities are initiated by the maintenance group and are 4% of the total maintenance activities (Lientz and Swanson, 1980).

The software maintenance is the most effort-consuming phase (more than 50% of the total effort). The effective maintenance activities ensure the sustainability of the software product along with happy users and customers.

10.3 Challenges of Software Maintenance

Software maintenance is a very long and most costly phase of software development life cycle. The duration and cost make this phase challenging with many difficulties, myths and perceptions. Some of the challenges are discussed in the following subsections.

10.3.1 High Staff Turnover

Software industry has high staff turnover. Average stay of a software developer in a company is less than five years. This situation results into non-availability of developers of the software for maintaining the same software. As a result, many software products are maintained by persons who are not the original developers. The new persons may not have proper understanding and adequate knowledge about the software. They may make modifications without understanding their impacts on the other parts of the software. The impact of any change on other parts of the software is called *ripple effect*. The problem may further be complicated due to non-availability of documentation. Even if some documents are available, they may not be proper and adequate to help in the maintenance process of the software. Furthermore, if changes are made without proper understanding of the source code, program logic and implementation strategies, the quality of the software may reduce and chances of failure may increase. Hence, the original developers are real assets for any software product and its company, but retaining them is a real challenge and perhaps the most difficult task.

10.3.2 Flexible Nature of Software

Many persons feel that software is flexible enough to accommodate any change and may incorporate additional functionalities easily even after its delivery to the customer. This is not at all true. Every change is difficult to incorporate. The additional functionalities may do value addition but the quality of the software always becomes suspicious. This “so-called flexibility” makes the life difficult for the software developers and leads to a critical, complex and challenging maintenance phase.

10.3.3 Poor Documentation and Manuals

Software maintenance activities can be performed effectively with the help of proper documentation and operating procedure manuals. They help in understanding the software and also help to identify the reasons of software failures. The reasons may be present in SRS document, SDD document or in the source code itself. After the identification of reason(s), a corrective action is required to make the software operational. In the case of addition of new functionality, documents provide foundations for design of interfaces and ways to proper sharing of data amongst new and old classes.

10.3.4 Inadequate Budgetary Provisions

Maintenance activities are performed under time and resource constraints. Generally, adequate budgetary provisions are not made for such activities. Software costing models are not giving due importance to maintenance activities. Maintenance effort cannot be imagined and estimated at the time of software costing. We may ask additional cost for new functionality but many times customers are not willing to pay requested cost happily. The conflict between the customer and the company may make the maintenance phase more challenging. The problems may be minimized if adequate budgetary provisions are made at the time of costing of the software product.

10.3.5 Emergency Fixing of Bugs

There is always an emergency to restore the operations of the software after its failure. The bugs are identified and corrective actions are performed. We may like to retest the software in order to ensure that corrections are rightly implemented and have not affected other parts of the software. This retesting (which is also called regression testing) is generally performed in emergency due to serious shortage of time. In reality, modified portion of the software is given with inadequate testing. This emergency fixing of bugs is also known as *patching*. Patching deteriorates the quality and structure of the software and may make the software more vulnerable. Regression testing should be performed effectively to maintain the expected standards of the software. If we do not do so, we are playing with the quality of the software. This practice of fixing bugs under emergency conditions resulting in inadequate testing is very common, which makes the maintenance phase much more challenging and difficult.

10.4 Maintenance of Object-Oriented Software

One of the reasons of the popularity of object-oriented paradigm is that such software products are easy to maintain and help to reduce maintenance effort. The fundamental entity is an object which is an independent unit of the software. Implementation details of an object are not visible to the outside world due to information hiding. An operation of an object is carried out after receiving a message which is the only allowed way of communication. When we make a change in an object, it will not have any impact outside that object due to information hiding. With this concept, we are able to minimize the scope of a change, which further makes it easy to maintain such a software product. A change will not affect the other portions of the source code

which are outside the modified object. Physical and conceptual independence of objects may help to identify that portions of the source code which are to be modified to achieve a specific maintenance purpose.

Inheritance and dynamic binding are two important characteristics of object-oriented software development and are commonly used in practice. With all their great advantages (like software reuse), these characteristics complicate the operations and also reduce the understandability of the program. Poor understanding may further make the maintenance tasks difficult and challenging.

Inheritance promotes the distributed class descriptions. A complete description of a class can be obtained by viewing not only the class but also its superclasses. Classes are generally described at different places in the program and there is no single place where a developer can turn to get a complete description of a class. This dimension reduces the understandability and a developer has to spend a substantial amount of time to search through various class descriptions to find the desired information. It is also required to study all the parts in sufficient detail to understand the program. In some cases, maintenance effort may not reduce significantly as compared to procedural software development. For example, consider the class hierarchy given in Figure 10.2. The student class inherits the person class, the examination class inherits the student class and the result class inherits the examination class. In order to understand the result class, a developer in the maintenance phase will have to study all the classes of the inheritance tree.

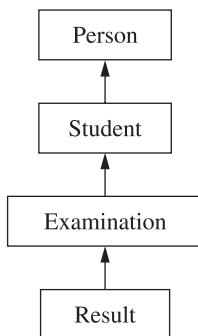


Figure 10.2 Inheritance hierarchy.

Dynamic binding provides many facilities (like flexibility) in object-oriented software development, but it may make it difficult to trace the dependencies. The difficulty in tracing has an adverse effect on the understandability of the program and on the calculation for the severity of impact of any change in the program. Dynamic binding can be achieved through run-time polymorphism. In run-time polymorphism, the objects of the polymorphic class change and respond in a different manner at run-time for the same message. Run-time polymorphism can be a problem during the maintenance phase. Consider the example shown in Figure 10.3. The base class is the shapes class and has four subclasses—line, square, circle and rhombus.

In the base class shapes, draw is a virtual function. Thus, all the four subclasses must implement their own draw function. With the absence of virtual functions, all the outputs, no matter the object of the shapes class contains reference to any of the subclasses, would print

```

.....
class shapes
{:
public:
    virtual void draw()
{
    cout<<"Point";
}
};

class Line: public shapes
{
void draw()
{
    cout<< "\nLine";
}
};

class Square: public shapes
{
void draw()
{
    cout<< "\nSquare";
}
};

class Circle: public shapes
{
void draw()
{
    cout<< "\nCircle";
}
};

class Rhombus: public shapes
{
virtual void draw()
{
    cout<< "\nRhombus";
}
};
.....

```

Figure 10.3 Example to demonstrate run-time polymorphism.

‘points’ because all the calls to draw will refer to the function draw in the base class. However, with the use of virtual functions, the object of the base class shape will call the corresponding object which it refers to. The consequence of the run-time polymorphism will be shown in the maintenance phase. The object of the shapes class is declared as follows:

Shapes *obj;

obj → draw();

If the testing person has to test obj → draw(), then he/she has to understand what would be the output if the object obj consists of reference to any of the four subclasses—line, square, circle

and rhombus. The trace through the source code is the only solution to understand the run-time polymorphism either by debugging or manually.

In the inheritance hierarchy, if there is a change in any of the base classes, all the classes dependent on the base class will have to be analysed and retested. Hence, although inheritance and dynamic binding have many advantages during software development, they also make the life of the developer difficult in the software maintenance phase due to poor understanding of the program and complicated dependencies. Error handling mechanism helps in constructing a robust system and reduces the cost to failure. In object-oriented languages such as C++ and Java, this error handling mechanism is called *exception handling*. Exception handling is the method of building a system to detect and recover from exceptional conditions. This mechanism may be helpful in detecting the exceptions in the maintenance phase.

10.5 Software Rejuvenation

It is not easy to maintain a software product for a long period of time due to continuous changes in the source code and associated documents. All changes should be done in a systematic way and at all desired places of the software. Associated documents should also be modified in order to make them relevant and meaningful after the changes. Software rejuvenation addresses all those activities which are carried out to enhance the quality of the existing system. This may include modifications in the documents, upgradations of software interfaces, restructuring of source code to improve efficiency, performance, usefulness, etc. Some of the activities of software rejuvenation are given as:

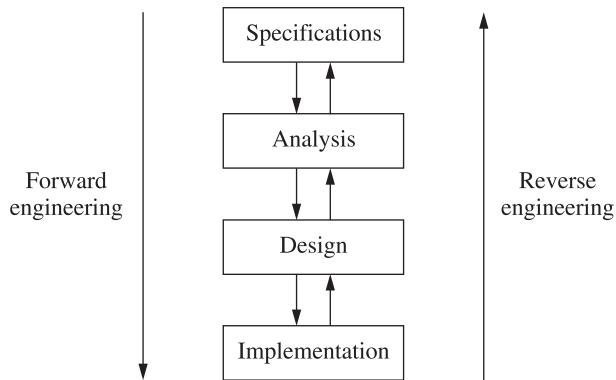
- (i) Reverse engineering
- (ii) Software re-engineering
- (iii) Redocumentation and restructuring

The purpose of such activities is to improve the quality of the software system and preserve its value. The output may be a more sustainable and reliable software system and in some cases, a new software system will be produced on the basis of the existing requirements.

10.5.1 Reverse Engineering

There are situations in practice when the only documentation available for maintenance is the source code. These situations are really very difficult from the perspective of maintaining such software product. One of the solutions is to understand the source code thoroughly and prepare a design document and also a specification document. This process of creating the design document from the source code and the specification document from the design document is called *reverse engineering*. Hence, the journey of creating documents from source code to specifications (in parts or complete) is called reverse engineering and is shown in Figure 10.4.

In software development life cycle, we start from the specification phase and go through the analysis phase, design phase and implementation phase. This journey of software development life cycle is called *forward engineering*.

**Figure 10.4 Process of reverse engineering.**

Reverse engineering helps to find complex, difficult, unknown and hidden information about a software system. It also helps to understand the behaviour of the software and improves the process of maintenance. Reverse engineering works from the lower level of abstraction to the higher level of abstraction like source code to design and design to specifications. Tools are available to support the process of reverse engineering. There are tools that produce a class diagram from the source code. Some available tools are Rational Rose of IBM, Paradigm Plus of Computer Associates International, Graphical Designer of Advanced Software Technologies, etc. Some of other popular tools are given as follows:

1. *Rigi (University of Victoria, Canada)*: This is an open and public domain reverse engineering tool primarily developed as a research prototype. Rigi can be used for C, C++ and COBOL programming languages.
2. *Refine (Reasoning Systems Inc.)*: This is an open and programmable tool and primarily designed for refinery environment. Refine can be used on Ada, C and COBOL for the purpose of analysis and generating documents.
3. *SNIFF+ (Takefive Software)*: This tool provides a software development environment along with reverse engineering capabilities.

The role of reverse engineering is important in maintenance, and effective tools may make the task less expensive and less troublesome. Reverse engineering is applicable at any phase of software development life cycle and also at any level of abstraction. It improves the understanding level and also highlights the critical and difficult area of software system.

10.5.2 Software Re-engineering

We may keep on modifying the software to perform various maintenance activities. As the duration of the maintenance phase increases, such software systems are often called *legacy systems*. These legacy systems are based on earlier technologies and may be the best in terms of those technologies. However, after many years, new technologies are available which are efficient, simple and easy to use and offer a good number of facilities. Hence, legacy systems may not be able to fulfil the aspirations and expectations of the today's users and customers. Moreover, structures of the legacy systems are deteriorated due to various modifications, and

their documentations, if available, have also become irrelevant over the period of time. The developers of such legacy systems may also not be available in the software organizations.

The software re-engineering is the process of taking legacy systems and re-implement them to make more efficient, useful, maintainable, reliable and effective software systems. Arnold (1993) has defined re-engineering as:

Software re-engineering is any activity that: (i) improves one's understanding of software or (ii) prepares or improves the software itself, usually for increased maintainability, reusability, or evolvability.

The re-engineering may require some form of reverse engineering for the creation of abstract view of the system. The abstract view may be converted into a new software system using forward engineering activities. The whole process is depicted in Figure 10.5.

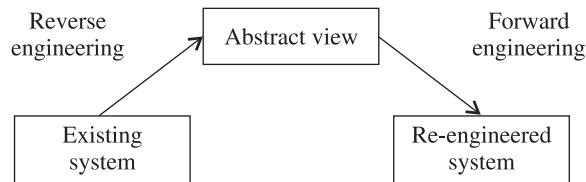


Figure 10.5 Process of re-engineering.

The abstract view may be a design, specification or analysis view of the system. The SRS document, SDD document and analysis documents may be redesigned/modified for the development of a new re-engineered system using new technologies or a mix of new and old technologies. The objective is to develop a new software system from the existing system which fulfills the needs of customers and users. Some of the applications of re-engineering are migrating a software system from one platform to another, designing a system using new programming language and/or database, developing a system to fulfil the customer's expectations, etc. The re-engineered systems are expected to have improved quality, better efficiency and performance, higher maintainability and enhanced usability.

The software re-engineering process is complex in nature and may not be completely automated. Software tools may only support some activities. The process is primarily carried out by human intervention and thus, the success of the final product is dependent on the capabilities of the humans involved in the process of re-engineering.

10.5.3 Redocumentation and Restructuring

The redocumentation activities are carried out at the same level of abstraction. They are performed at the source code level, design level and specification level. The purpose is to improve the quality of the document under consideration. It is the process of recreating semantically equivalent document with better understandability, improved quality and enhanced usability. This process improves the quality of the existing document and can be applied at the same level of abstraction. The resulting document is considered as an alternate view intended for its users. Redocumentation is an ongoing activity and helps to improve the maintainability of the software. The documents are modified to preserve their relevance and value.

Restructuring is the process of transforming a system from one form to another without changing the logic or functionality. We generally restructure a software product to make it more understandable and easy to change with reduced complexity. Software restructuring is language dependent. A tool of one language (say C) will not work for another language (say visual basic). There are many technical restructuring problems. For example, there is no perfect available solution for handling C-preprocessor directives. There will never be a complete solution for analysing pointers. Restructuring is like renovating a building to maintain and improve its value. The same analogy is true for a software product to maintain and improve its value. Restructuring work comes under the category of perfective maintenance and modifies the structure of the source code along with associated documents.

10.6 Estimation of Maintenance Effort

How can we estimate the maintenance effort? As we all know, it is very significant and important for the sustainability of the software system. If effective verification activities are performed in the early phases of software development, then maintenance effort may be reduced. The changes made in the software during the maintenance phase may degrade the coupling and cohesion of components and inheritance structure of an object-oriented system. A few models are available in the literature to estimate the maintenance effort and two are discussed in the subsequent subsections.

10.6.1 Belady and Lehman Model

Belady and Lehman (1976) were among the first researchers who worked to find the maintenance effort. They considered the deterioration that occurs to a large system over time. The proposed model indicates that the effort can increase exponentially if a poor software development approach is used and developers are not available for maintaining the system.

The fundamental equation of the model is given as:

$$M = P + Ke^{c-d}$$

where

M = Total maintenance effort

P = Productive effort which includes analysis, design, coding and testing

K = An empirically determined constant

c = Complexity caused by lack of structured design and documentation

d = Degree to which the maintenance team is familiar with the software

The complexity (c) will increase if the software engineering practices are not followed. The value of c will be high for large-size projects as compared to small-size projects. If any changes are made without proper understanding, then the value of degree will be low. The issues of software development effort estimation are also applicable to maintenance effort estimation. If the values of c and d are found realistically, we may get an indicative figure of the software maintenance effort. The estimated effort may further be refined on the basis of past experience on similar projects.

EXAMPLE 10.1 The development effort of a software project is 1000 person-months (PMs). The value of K for the project is 0.5. The complexity of the source code is 10. Find the maintenance effort if

- (i) the maintenance team has very good understanding of the project ($d = 0.8$).
- (ii) the maintenance team has poor understanding of the project ($d = 0.2$).

Solution The given values are

$$P = 1000 \text{ PMs}$$

$$c = 10$$

$$K = 0.5$$

- (i) Case 1 when $d = 0.8$

$$\begin{aligned} M &= P + Ke^{c-d} \\ &= 1000 + 0.5e^{10-0.8} = 5948.56 \end{aligned}$$

- (ii) Case 2 when $d = 0.2$.

$$\begin{aligned} M &= P + Ke^{c-d} \\ &= 1000 + 0.5e^{10-0.2} = 10016.87 \end{aligned}$$

The effort increases exponentially if poor software engineering practices are used and project understanding is low.

10.6.2 Boehm Model

Boehm (1981) proposed an equation for the calculation of maintenance effort along with its COCOMO model. Boehm used the term ‘annual change traffic’ (ACT) and defined it as:

The fraction of a software product’s source instructions which undergo change during a maintenance year either through addition, deletion or modification.

ACT is related to the number of change requests and is calculated as:

$$\text{ACT} = \frac{\text{KLOC}_{\text{added}} + \text{KLOC}_{\text{deleted}} + \text{KLOC}_{\text{modified}}}{\text{KLOC}_{\text{total}}}$$

Annual maintenance effort (AME) is calculated as:

$$\text{AME} = \text{ACT} \times \text{SDE}$$

where SDE is the software development effort in person-months.

Suppose, for a project, the development effort is 500 PM. It is also estimated that 25% of the source code is modified in a year. The AME will be $0.25 \times 500 = 125$ PMs. Boehm has further suggested that the estimated effort may further be defined using appropriate cost multipliers, and the effort adjustment factor (EAF) is calculated on the guidelines of COCOMO model. The modified equation is given as:

$$\text{AME} = \text{ACT} \times \text{SDE} \times \text{EAF}$$

The AME may be indicative but verifiable after the completion of a few years of maintenance phase.

EXAMPLE 10.2 The development effort for a software project is 1000 PMs. The ACT is 25% per year. Compute the AME. If the life of the project is 10 years, calculate the total effort.

Solution

$$\text{SDE} = 1000 \text{ PMs}$$

$$\text{ACT} = 25\% \text{ per year}$$

Duration of the project = 10 years

$$\text{AME} = \text{ACT} \times \text{SDE} = 1000 \times 0.25 = 250 \text{ PMs}$$

Maintenance effort for 10 years = $250 \times 10 = 2500 \text{ PMs}$

EXAMPLE 10.3 A software project has development effort of 1000 PMs. It is assumed that 20% source code will be modified per year. Some of the cost multipliers are given as:

- (i) Required software reliability (RELY): high (1.15)
- (ii) Analyst capability (ACAP): low (1.18)
- (iii) Application experience (AEXP): high (0.91)
- (iv) Modern programming practices (MODP): high (0.8)
- (v) Data base size (DATA): high (1.08)

These values are taken from the COCOMO model's table of cost drivers (Boehm, 1981). Other multipliers are nominal (value = 1). Calculate AME.

Solution

$$\text{ACT} = 20\% \text{ per year}$$

$$\text{SDE} = 1000 \text{ PMs}$$

$$\begin{aligned}\text{EAF} &= 1.15 \times 1.19 \times 0.91 \times 0.91 \times 1.08 \\ &= 1.224\end{aligned}$$

$$\begin{aligned}\text{AME} &= \text{ACT} \times \text{SDE} \times \text{EAF} \\ &= 0.2 \times 1000 \times 1.224 \\ &= 244.78 \text{ PMs}\end{aligned}$$

10.7 Configuration Management

All the items produced during software development are controlled and managed by configuration management. The changes made in any software item during the software development life cycle are also controlled by the configuration management. The configuration management consists of three activities: configuration identification, configuration control and configuration accounting. Configuration identification activity ensures that each individual software item is uniquely numbered and named along with the date and time. The configuration control activity ensures that the changes are well notified, approved and documented, and the configuration accounting activity keeps account of the status of each software item.

10.7.1 Configuration Identification

Each individual item developed in the software development life cycle phase should be uniquely named. The individual items include individual source code modules, documents (such as SRS, SDD, test plans, test procedures, test cases and reports), and operating manuals. The first issue of these items is known as *release*. After the release, if any significant changes are made, they are known as *versions*. Similarly, after the versions, if any minor revisions are made, these are known as *editions* (see Figure 10.6).

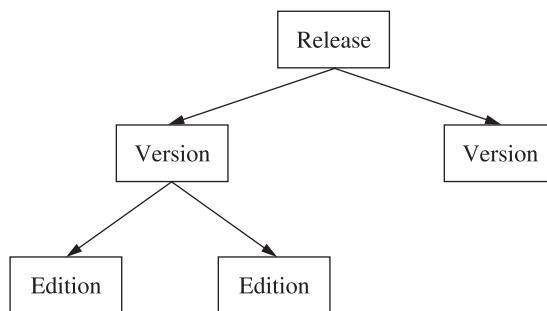


Figure 10.6 Levels of products.

Hence, all the modules of a source code or any document should be uniquely identified by the name, date and time so that the latest software item is always known and available for use.

10.7.2 Configuration Control

Configuration control is a part of configuration management activities and is responsible for approving, controlling and implementing changes. The typical change request process is shown in Figure 10.7.

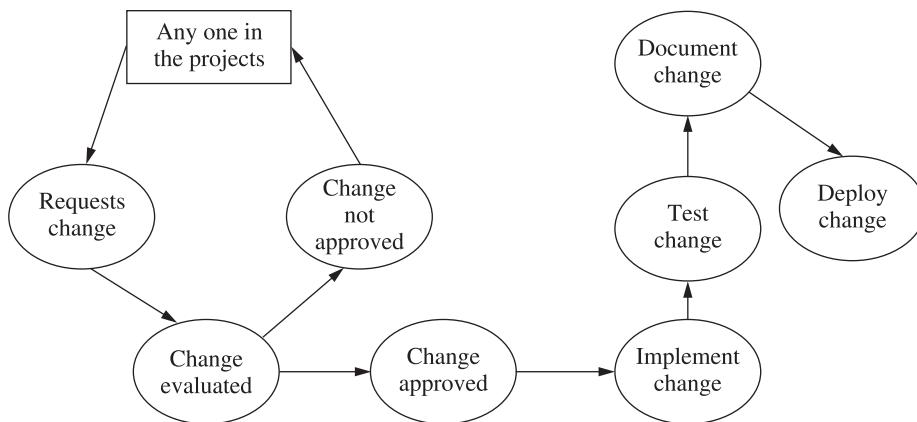


Figure 10.7 Typical change request process.

The change control board (CCB) is responsible for approving and monitoring the changes. All the changes must be carefully reviewed by this board before approval. After the changes have been made, these changes must be notified so that they are included in the software library. The software library is a repository where the entire products produced during the software development life cycle are kept. The librarian is responsible for keeping all the current issues of each software component. Once the changes have been made, implemented and deployed, then these changes are officially notified using software change notice. The software change notice informs the librarian of a change. The format of the software change notice is given in Table 10.1.

Table 10.1 Format of software change notice

Module in which change is made:	
Identifier of item in which change is made#:	
Change approved by:	
Date and time of the change approval:	
Change implemented by:	
Date and time of the change implementation:	
Signature:	

The changes may include defect corrections, quality improvement or any enhancement in the software product.

10.7.3 Configuration Accounting

The responsibility of configuration accounting is to keep track of all the activities including changes and any action taken that affects the configuration of the product. Configuration accounting is responsible for keeping all the data corresponding to the change. The parts affected by the change must also be tracked and monitored so that all the identifiers of the affected components can be updated after the changes have been made. The multiple instances of any software item may be present, and configuration accounting should ensure that the correct instance is called each time.

10.8 Regression Testing

When the software is developed, development testing is carried out in order to increase the confidence in the correctness and reliability of the software. The development testing involves construction of test plans, development of test design and procedure, and design and execution of test cases that produce test results. When the software is modified in the maintenance phase, it requires retesting to assure its continued operation. This process of retesting is known as *regression testing*. Regression testing is defined as the process of testing the modified parts of the software and ensures that no new faults have occurred in the previously developed source code due to these modifications. Thus, regression testing not only tests the modified part of the

software, but also checks the parts affected by the change. Hence, regression testing is done to ensure that:

1. There are no faults in the other parts of the software due to modifications.
2. Correctness of the software.
3. Quality and reliability of the software.
4. Confidence in the modified parts.

The existing test suite may be used in regression testing. However, if any new portions are added in the source code, then an additional regression test case must be developed and an updated test suite should be prepared which can be used in any future modification. The difference found from the expected results must be addressed and the required corrective action must be made.

Regression testing is not only a part of maintenance activity, it may also be performed during the later stages of software development cycle. This later phase begins after the completion of the construction of test plans and test suites and start of initial testing of the software. During this phase, we test the source code with the aim to correct faults and hence is similar to the maintenance activity. The differences between development and regression testing are given in Table 10.2.

Table 10.2 Development testing versus regression testing

S. No.	Development testing	Regression testing
1	Test plans are prepared and test suites are constructed.	The test suites including the test cases are already available. Some additional test cases may be added.
2	The complete software is tested.	Only the modified parts of the software and the other portion of the software affected by these modifications are tested.
3	It is carried out once in the entire lifetime of the software.	It may occur many times during the lifetime of the software.
4	Budget and time are allocated separately.	Generally, no separate budget and time are allocated.
5	It is performed under the release date pressure	It is performed under greater pressure and time constraints.

The process of regression testing is very costly and involves the following steps:

1. Analysis of failure report
2. Debugging of the source code
3. Identification of fault(s) in the source code
4. Source code modification (new and old programs will be different)
5. Selection of test cases from existing test suite to ensure the correctness of modification(s)
6. Addition of new test cases, if required
7. Performing retesting to ensure correctness using selected test cases and new test cases, if any

Regression testing may be performed effectively within specified time, if existing test cases are prioritized according to some criteria. Many techniques and algorithms are available in literature

with proclaimed effectiveness and efficiency for prioritizing the test cases. However, applications of such techniques are limited to specific projects under stated conditions. Some software tools are also available which may assist and help during the process of regression testing.

Review Questions

1. What is software maintenance? Explain the different categories of maintenance.
2. What are the challenges involved in maintaining an object-oriented system? Explain with the help of an example.
3. Discuss the various problems during maintenance of object-oriented software? Describe some solutions to these problems.
4. Explain the significance of maintenance. Which category of maintenance consumes maximum effort and why?
5. What are the challenges of software maintenance?
6. Inheritance and polymorphism are two major constructs of object-oriented software. How do these characteristics increase complications in the maintenance phase of software development? How can these complications be reduced?
7. What is software rejuvenation? What are the activities involved in software rejuvenation?
8. What is regression testing? How is it different than development testing?
9. What is reverse engineering? List the tools for reverse engineering.
10. What is re-engineering? Explain the process of re-engineering.
11. Annual charge traffic in a software system is 30% per year. The initial development cost was ₹ 25 lacs. Total lifetime for the software is 12 years. What is the total cost of the software system?
12. How can maintenance effort be computed using Belady–Lehman's model?
13. The development effort of a software project is 2000 person-months. The value of K for the project is 0.8. The complexity of the source code is 10. Find the maintenance effort if
 - (i) the maintenance team has very good understanding of the project ($d = 0.6$).
 - (ii) the maintenance team has poor understanding of the project ($d = 0.4$).
14. Describe Boehm's model of software maintenance. What is the role of effort adjustment factor?
15. What is configuration management? Explain the various activities of configuration management.
16. What is the role of change control board in configuration management?
17. Explain the typical change management procedure with the help of a block diagram.
18. (a) What are the various activities followed in configuration management? Explain them in detail.
(b) What is the use of software libraries?

19. Draw the performa of software change notice (SCN). What is the importance of issuing an SCN?
20. What is the importance of configuration identification? Differentiate between version, edition and release.

Multiple Choice Questions

Note: Select the most appropriate answer of the following questions:

1. Which of the following is **not** a maintenance activity?

(a) Perfective maintenance	(b) Quality maintenance
(c) Corrective maintenance	(d) Adaptive maintenance
2. Maintenance related to modification in the software due to changing environment is called:

(a) Perfective maintenance	(b) Preventive maintenance
(c) Corrective maintenance	(d) Adaptive maintenance
3. Corrective maintenance is related to:

(a) Modifications due to change in environment	(b) Modifications due to demand of addition of new functionalities by the customer
(c) Modifications due to correction of defects in the software	(d) Modifications due to increase in complexity
4. Perfective maintenance refers to:

(a) Modifications in order to improve the maintainability	(b) Modifications due to demand of addition of new functionalities by the customer
(c) Modifications due to correction of the defects in the software	(d) Modifications due to increase in complexity
5. How much percent effort is consumed by maintenance phase?

(a) >50%	(b) ≤30%
(c) 10%	(d) 80%
6. Which maintenance activity consumes maximum share of the effort?

(a) Perfective maintenance	(b) Preventive maintenance
(c) Corrective maintenance	(d) Adaptive maintenance
7. Software rejuvenation refers to:

(a) Activities related to elimination of defects from the software	(b) Activities related to improvement in the quality of the software
(c) Activities related to modifications due to changes in the changing environment	(d) None of the above
8. The purpose of reverse engineering is:

(a) Restructuring of source code	(b) Upgradation of software interfaces
(c) Modifications in the documents	(d) Creation of design and specification documents from the source code

9. The purpose of re-engineering is:
 - (a) Reimplementation of legacy systems
 - (b) Upgradation of software interfaces
 - (c) Modifications in the documents
 - (d) Creation of design and specification documents from the source code
10. Which one of the following is a maintenance model?
 - (a) COCOMO model
 - (b) Putnam resource allocation model
 - (c) Belady and Lehman model
 - (d) Prototyping model
11. Which model considers the deterioration that occurs to a large system over time?
 - (a) Boehm model
 - (b) Putnam resource allocation model
 - (c) Belady and Lehman model
 - (d) Quick and fix model
12. ACT stands for:
 - (a) Annual change traffic
 - (b) Annual charge traffic
 - (c) Actual change traffic
 - (d) Annual charge team
13. In Boehm's model, annual maintenance effort (AME) is calculated as:
 - (a) $AME = ESTIMATED\ EFFORT \times SDE$
 - (b) $AME = CAP \times SDE$
 - (c) $AME = ACC \times SDE$
 - (d) $AME = ACT \times SDE$
14. Configuration identification ensures:
 - (a) Every item is defect free
 - (b) Every item is uniquely identified
 - (c) Status of each software item is maintained
 - (d) Changes are approved and notified
15. The responsibility of CCB is to:
 - (a) Approve and monitor changes
 - (b) Detect and correct defects
 - (c) Improve the quality of the source code
 - (d) All of the above
16. CCB stands for:
 - (a) Change corporation board
 - (b) Control change board
 - (c) Change control board
 - (d) Change control boundary
17. The purpose of regression testing is to ensure:
 - (a) Correctness of the software
 - (b) Quality of the software
 - (c) Confidence in modified parts of the software
 - (d) All of the above
18. Regression testing is related to:
 - (a) Object-oriented analysis phase
 - (b) Object-oriented design phase
 - (c) Maintenance phase
 - (d) Implementation phase

19. Maintenance of object-oriented software is difficult due to the use of:

 - (a) Classes
 - (b) Inheritance and polymorphism
 - (c) Exception handling
 - (d) None of the above

20. Software maintenance is difficult due to:

 - (a) Non-availability of original developers
 - (b) Flexible nature of software
 - (c) Ever-increasing demands of customers
 - (d) All of the above

Further Reading

The classic paper written by Canfora and Cimitile explains the categories of software maintenance, maintenance management and re-engineering with a special focus on legacy systems:

Canfora, G., and Cimitile, A., Software maintenance, *Proc. 7th Int. Conf. Software Engineering and Knowledge*, 1995.

The following research paper describes the basic object-oriented testing and maintenance concepts and introduces object oriented-testing and maintenance environment:

This paper identifies difficulties that can be expected in maintenance of object-oriented programs and also provides tool support for the maintenance process:

Wilde, N. and Huitt, R., Maintenance support for object oriented programs. *Proceedings Conference on Software Maintenance*, 162-170, Oct. 1991.

The following research paper presents an elaborate case study on software maintenance:

Domsch, M., and Schach, S.R., A case study in object-oriented maintenance. *15th IEEE International Conference on Software Maintenance (ICSM'99)*, 1999.

Vellankny describes the issues, tools, techniques and trends for maintaining software in his book:

Vellanki, P., *Software Maintenance—A Management Perspective (Issues, Tools, Techniques, and Trends)*. Universal Publishers, India, 2007.

An extensive review on maintainability of object-oriented software is given in:

Hall, John M., The maintainability of object-oriented software. Unpublished paper.

The following paper provides an empirical study on software maintenance focusing on error occurrence and fault detection:

Torchiano, M., Ricca, F., and Lucia, A., Empirical studies in software maintenance and evolution.
IEEE International Conference on Software Maintenance, 2–5, Oct. 2007.

Hoglund and McGraw provide a relationship between reverse engineering and program understanding in their paper:

Hoglund, G., and McGraw, G., *Reverse Engineering and Program Understanding*. Boston: Addison-Wesley, 2004.

Bennett and Rajlich suggested a research framework and summarized a long-term view of software evolution in:

Bennett, K., and Rajlich, V., Software maintenance and evolution: A roadmap, *Proceedings of the Conference on the Future of Software Engineering*. ACM New York, USA, 2000.

The following paper presents a technique for detecting classes that may be prone to deteriorate:

Subramaniam, G.V., and Byrne, E.J., Reengineering the class—An object oriented maintenance activity, *Proceedings of the 22nd International Computer Software and Applications Conference*. IEEE Computer Society Washington, DC, USA, 1998.

An excellent chapter on configuration management is present in:

Horch, John W., *Practical Guide to Software Quality Management*. Norwood, MA: Artech House, 2003.

A practical-oriented book on configuration management is given below:

Buckley, Fletcher J., *Implementing Configuration Management: Hardware, Software and Firmware*, 2nd ed., New York: IEEE Press, 1995.

The below article provides a full account on design and maintenance of behavioural regression test suites that may help to change code with confidence:

daVeiga, Nada, Change code without fear: Utilize a regression safety net, *DDJ*, February 2008.

Useful recommendation by Microsoft's on regression testing can be obtained from:

Microsoft regression testing recommendations, [http://msdn.microsoft.com/en-us/library/aa292167\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa292167(VS.71).aspx)

A useful introduction on regression testing performed in real-life environment is given by Onomo et al.:

Onomo, A.K., Tsai, Wei-Tek,, Poonawala, M., and Suganuma, H., Regression testing in an industrial environment. *Communications of the ACM*, **45**(5): 81–86, 1998.

APPENDIX

Software Requirements Specification Document for Library Management System

Problem Statement

A software is to be developed for automating the functioning of a university library. There are three types of members in the library—students, faculty and employees. There is an upper limit in terms of maximum number of books issued to a member and duration for which a book is issued. This limit may vary from one type of members to other types of members. The library performs the following functions:

1. Issue of Books

- (a) Book are issued to students, faculty and employees as per their specified limits of duration and the number of books.
- (b) Current date is used as issue date for a book.
- (c) The due date for return of book is calculated as per specified limit of duration.
- (d) The due date is stamped on the book.

2. Return of Books

- (a) Any person can return the issued books.
- (b) If a book is returned after the due date, a fine is charged for each day delayed from students. The charges may vary from time to time. However, faculty and employees are not charged any fine due to late return of a book.

3. Maintenance of Information

The library maintains the following information:

- (a) Details of all members with their name, address, designation (wherever applicable), phone number and unique identification code.
- (b) Details of all books with their author name(s), publisher, price, number of copies and ISBN number.
- (c) Details of books issued to every member.

The automation of library activities should provide the following:

- (a) A system which can run on the university's library LAN.
- (b) A barcode reader should be used to facilitate the process of issue and return of books.
- (c) Generates login ID and password to the system operator.
- (d) Calculates the fine, if applicable and generates a fine receipt.
- (e) Maintains the details of students, faculty and employees.
- (f) Maintains the details of all books in the library.
- (g) Maintains the details of books issued to every member.
- (h) Availability of a particular book.
- (i) Availability of the number of copies of a particular book.
- (j) Reserves a book for a student for 24 hours, if that book is not currently available.

The system should also be able to generate reports like:

- (a) Details of all books in the library.
 - Author-wise
 - Publisher-wise
 - Title-wise
 - Subject-wise
- (b) Details of all members.
- (c) Details of books issued to members.
- (d) Status of fine, member-wise, wherever applicable.

Contents

- 1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions, Acronyms and Abbreviations
 - 1.4 References
 - 1.5 Overview
- 2. Overall Description
 - 2.1 Product Perspective
 - 2.1.1 System Interfaces
 - 2.1.2 User Interfaces
 - 2.1.3 Hardware Interfaces
 - 2.1.4 Software Interfaces
 - 2.1.5 Communication Interfaces
 - 2.1.6 Memory Constraints

-
- 2.1.7 Operations
 - 2.1.8 Site Adaptation Requirements
 - 2.2 Product Functions
 - 2.3 User Characteristics
 - 2.4 Constraints
 - 2.5 Assumptions and Dependencies
 - 2.6 Apportioning of Requirements
 - 3. Specific Requirements
 - 3.1 External Interface Requirements
 - 3.1.1 User Interfaces
 - 3.1.2 Hardware Interfaces
 - 3.1.3 Software Interfaces
 - 3.1.4 Communication Interfaces
 - 3.2 Functional Requirements
 - 3.2.1 Login
 - 3.2.2 Maintain Book Details
 - 3.2.3 Maintain Student Details
 - 3.2.4 Issue Book
 - 3.2.5 Return Book
 - 3.2.6 Fine Calculation
 - 3.2.7 Reserve Book
 - 3.2.8 Query Book
 - 3.2.9 Search Book
 - 3.2.10 Report Generation
 - 3.3 Performance Requirements
 - 3.4 Design Constraints
 - 3.5 Software System Attributes
 - 3.6 Logical Database Requirements
 - 3.7 Other Requirements

1. Introduction

A university is organized in different teaching schools and each school conducts a variety of programmes. A university has a central library that issues/returns books manually to the students.

A software is to be developed for automating the manual library system. The system should be stand-alone in nature. The system operator must obtain a login ID and password from the “System administrator”. After admission, every student is able to receive books from the library management system and return them. The faculty employed in various schools and university employees can also receive books from the library management system and return them. The software calculates fine if the book is being returned after the specified due date. A student can reserve a book if the book is not currently available in the library. The software generates details of the books available in the library at any given time.

1.1 Purpose

The library management system (LMS) maintains the information about various books available in the library. The LMS also maintains records of all the students, faculty and employees in the university. These records are maintained in order to provide membership to them.

1.2 Scope

The proposed LMS must be able to perform the following functions:

1. Issue login ID and password to the system operators.
2. Maintain details of books available in the library.
3. Maintain details of the students in the university to provide student membership.
4. Maintain details of the faculty and employees in the university to provide faculty membership.
5. Issue a book.
6. Return a book, and calculate fine if the book is being returned after the specified due date.
7. Reserve a book.
8. Provide search facility to check the availability of a book in the library.
9. Generate the following reports:
 - (a) List of books issued by a member.
 - (b) Details of books issued and returned on daily basis.
 - (c) Receipt of fine.
 - (d) Total available books in the library.
 - (e) List of library members along with issued books.
 - (f) List of available books in the library:
 - Author-wise
 - Book title-wise
 - Publisher-wise
 - Subject-wise

1.3 Definitions, Acronyms and Abbreviations

SRS: Software Requirement Specification

LMS: Library Management System

System Operator: System administrator, library staff, data entry operator

RAM: Random Access Memory

Accession number: It is a unique sequence number allocated to each book in the catalogue.

Student: Any candidate admitted in a programme offered by a school.

System Administrator/Administrator: User having all the privileges to operate the LMS.

Data Entry Operator (DEO): User having privileges to maintain book, student and faculty/employee details.

Library Staff: User having privileges to issue, return, reserve and query books.

Faculty: Teaching staff of the university—Professor, Associate Professor and Assistant Professor.

School: Academic Unit that offers various programmes

1.4 References

- (a) Software Engineering by K.K. Aggarwal & Yogesh Singh, New Age Publishing House, 3rd Edition, 2008.

- (b) IEEE Recommended Practice for Software Requirements Specifications—IEEE Std. 830-1998.
- (c) IEEE Standard for Software Test Documentation—IEEE Std. 829-1998.

1.5 Overview

The rest of the SRS document describes various system requirements, interfaces, features and functionalities.

2. Overall Description

The LMS maintains records of books in the library and membership details of students, faculty and employees in the university. It is assumed that approved books have already been purchased by the acquisition section. It is also assumed that the students have already been admitted in the university for a specific programme. The system administrator will receive lists of the admitted students (school-wise and programme-wise) from the academic section. The establishment section will provide the list of the faculty and employees appointed in the university.

The LMS issues and returns book to students, faculty and employees. The LMS generates fine if the book is returned after the due date. It also allows students, faculty and employees to reserve a book. The student/faculty/employee can access the LMS on the university's library LAN in order to search the availability of a book from the library.

The administrator/DEO will have to maintain the following information:

- Book details
- Student membership details
- Faculty and employee membership details

The administrator/library staff will perform the following functions:

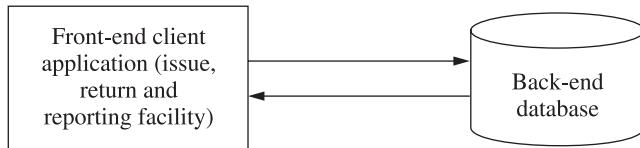
- Issue a book
- Return a book
- Reserve a book
- Query a book
- Generate reports
 - ◆ List of books issued by a member
 - ◆ Details of books issued and returned on daily basis
 - ◆ Receipt of fine
 - ◆ Total available books in the library
 - ◆ List of library members along with issued books

The administrator/student/faculty/employee requires the following information from the system:

- List of available books in the library:
 - ◆ Author-wise
 - ◆ Book title-wise
 - ◆ Publisher-wise
 - ◆ Subject-wise

2.1 Product Perspective

The LMS shall be developed using client/server architecture and be compatible with Microsoft Windows Operating System. The front-end of the system will be developed using Visual Basic 6.0 and the back-end will be developed using MS SQL Server 2005.



2.1.1 System Interfaces

None

2.1.2 User Interfaces

The LMS will have the following user-friendly and menu-driven interfaces:

- (a) **Login:** to allow the entry of only authorized users through valid login ID and password.
- (b) **Book Details:** to maintain book details.
- (c) **Student Membership Details:** to maintain student membership details.
- (d) **Faculty/Employee Membership Details:** to maintain faculty/employee membership details.
- (e) **Issue Book:** to allow the library staff to issue a book from the library.
- (f) **Return Book:** to allow the library staff to return a book to the library.
- (g) **Reserve Book:** to allow the library staff reserve a book.
- (h) **Query Book:** to check the availability of a book in the library.

The software should generate the following information:

- (a) Details of books issued and returned on daily basis.
- (b) Details of books available in the library.
- (c) Details of book issued to a student.
- (d) List of library members with issued books.
- (e) Receipt of fine.

2.1.3 Hardware Interfaces

- (a) Screen resolution of at least 640×480 or above.
- (b) Support for printer (dot matrix, deskjet, laserjet).
- (c) Computer systems will be in the networked environment as it is a multi-user system.

2.1.4 Software Interfaces

- (a) MS-Windows Operating System
- (b) Microsoft Visual Basic 6.0 for designing front-end
- (c) MS SQL Server 2005 for back-end

2.1.5 Communication Interfaces

Communication is via local area network (LAN).

2.1.6 Memory Constraints

At least 512 MB RAM and 500 MB space of hard disk will be required to run the software.

2.1.7 Operations

None

2.1.8 Site Adaptation Requirements

The terminal at the client site will have to support the hardware and software interfaces specified in sections 2.1.3 and 2.1.4, respectively.

2.2 Product Functions

The LMS will allow access only to authorized users with specific roles (system administrator, library staff, DEO and student). Depending upon the user's role, he/she will be able to access only specific modules of the system.

A summary of major functions that the LMS will perform is given as follows:

- A login facility for enabling only authorized access to the system.
- The system administrator/DEO will be able to add, modify, delete or view book, student, faculty/employee and login information.
- The system administrator/library staff will be able to issue, return and reserve a book.
- The system administrator/library staff will be able to query a book in order to check the availability of the book in the library.
- The system administrator/students/faculty/employee will be able to search a book from the library catalogue (author-wise, title-wise and publisher-wise).
- The system administrator/library staff will be able to generate various reports from the LMS.

2.3 User Characteristics

- Qualification: At least matriculation and comfortable with English.
- Experience: Should be well versed/informed about the registration process of the university.
- Technical Experience: Elementary knowledge of computers.

2.4 Constraints

- The software does not maintain records of periodicals.
- There will be only one administrator.
- The delete operation is available to the administrator and DEO. To reduce the complexity of the system, there is no check on delete operation. Hence, the administrator/DEO should be very careful before deletion of any record and he/she will be responsible for data consistency.
- User will not be allowed to update the primary key.

2.5 Assumptions and Dependencies

- The acquisition section of the library will provide the lists of the books purchased by the library.
- The academic section will provide the lists of the admitted students (school-wise and programme-wise).
- The establishment section will provide the list of the faculty members in various schools of the university. It will also provide the list of employees working in various branches such as examination, academics, establishment and schools of the university.
- The login ID and password must be created by the system administrator and communicated to the concerned user confidentially to avoid unauthorized access to the system.

2.6 Apportioning of Requirements

Not required

3. Specific Requirements

This section contains the software requirements in detail along with the various forms to be developed.

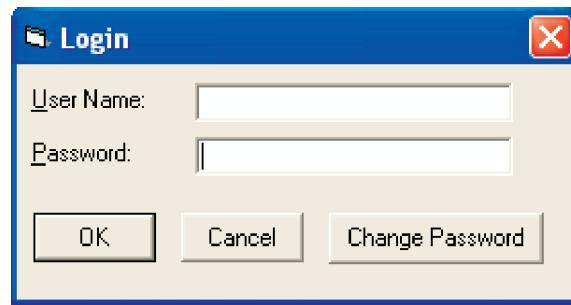
3.1 External Interface Requirements

3.1.1 User Interfaces

The following user interfaces (or forms) will be provided by the system.

(i) Login Form

This will be the first form, which will be displayed. It will allow the user to access the different forms based on his/her role.



Various fields available on this form will be:

- *Login ID*: Alphanumeric of length in the range of 4 to 15 characters. Special characters and blank spaces are not allowed.
- *Password*: Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are not allowed. However, special characters are allowed.

(ii) Change Password

The change password form facilitates the user to change the password. Various fields available on this form will be:

- *Login ID*: Alphanumeric of length in the range of 4 to 15 characters. Special characters and blank spaces are not allowed.
- *Old Password*: Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are not allowed. However, special characters are allowed.
- *New Password*: Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are not allowed. However, special characters are allowed.
- *Confirm Password*: Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are not allowed. However, special characters are allowed. The content of this field must match with the contents of the new password field.



(iii) Maintain Book Details

This form will be accessible only to the system administrator and DEO. It will allow him/her to add/update/delete/view information about existing book(s) for the library.

Various fields available on this form will be:

- **Accession number:** Numeric and will have value from 10 to 99999.
- **Subject descriptor:** Will display all the subject descriptors.
- **ISBN:** Numeric
- **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
- **Language:** Will display all the languages available.

- **Author:**
 - ◆ **First name:** Alphanumeric of length 3 to 50 characters. Special characters are not allowed. Numeric data will not be allowed.
 - ◆ **Last name:** Alphanumeric of length 3 to 50 characters. Special characters are not allowed. Numeric data will not be allowed.
- **Publisher:** Alphanumeric of length 3 to 300 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.

(iv) Student Membership Details

This form will be accessible only to the system administrator and DEO. It will allow him/her to add/update/delete/view information about existing student(s) in the university.

The screenshot shows a Windows-style application window titled "Student Membership". The interface is organized into several sections:

- Membership No.:** An input field for entering a numeric membership number.
- Roll No.:** An input field for entering a roll number.
- Name:** An input field for entering the member's name.
- Load Photograph:** A button to upload a photograph.
- School:** A dropdown menu listing schools.
- Programme:** A dropdown menu listing programmes.
- Father's Name:** An input field for entering the father's name.
- Date of Birth:** An input field for entering the date of birth.
- Address:** An input field for entering the address.
- Telephone:** An input field for entering the telephone number.
- Email:** An input field for entering the email address.
- Membership Date:** An input field for entering the membership date.
- Valid Upto:** An input field for entering the validity date.
- Password:** An input field for entering the password.
- Action Buttons:** On the right side, there are five buttons: "Add", "Delete", "Edit", "View", and "Close".

Various fields available on this form will be:

- **Membership number:** Numeric and will have value from 100 to 5999.
- **Photograph:** Will allow user to upload image of the member.
- **Roll No.:** Alphanumeric of length 11 characters and only digits from 0 to 9 are allowed. Alphabets, special characters and blank spaces are not allowed.
- **Name:** Alphanumeric, with length 3 to 50 characters. Blank spaces are allowed. Special characters are not allowed.
- **School:** Will display the name of all the schools.

- **Programme:** Will display all the programmes available in the selected school.
- **Father's name:** Alphanumeric and can have length 3 to 50 characters. Alphabetic letters and blank spaces are allowed. Special characters are not allowed (except '.' character).
- **Date of birth:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
- **Address:** Alphanumeric of length up to 10 to 200 characters. Blank spaces are allowed.
- **Telephone:** Numeric and can have length up to 11 digits.
- **Email:** Alphanumeric and can have length up to 50 characters. Email must have one '@' and '.' symbol.
- **Membership date:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
- **Valid up to:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
- **Password:** Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are not allowed. However, special characters are allowed. Initially contains 8 digits randomly generated number (auto-generated).

(v) Maintain Faculty/Employee Details

This form will be accessible only to the system administrator and DEO. It will allow him/her to add/update/delete/view information about existing faculty/employees in the university.

Various fields available on this form will be:

- **Membership number:** Numeric and will have value from 100 to 5999.
- **Photograph:** Will allow the user to upload image of the member.

- **Faculty/Employee ID:** Alphanumeric of length 11 characters and only digits from 0 to 9 are allowed. Alphabets, special characters and blank spaces are not allowed.
- **Name:** Alphanumeric, with length 3 to 50 characters. Blank spaces are allowed. Special characters are not allowed.
- **School:** Will display the name of all the schools.
- **Branch:** Will display all the branches available in the university.
- **Father's name:** Alphanumeric and can have length 3 to 50 characters. Alphabetic letters and blank spaces are allowed. Special characters are not allowed (except ‘.’ character).
- **Date of birth:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
- **Address:** Alphanumeric of length up to 10 to 200 characters. Blank spaces are allowed.
- **Telephone:** Numeric and can have length up to 11 digits.
- **Email:** Alphanumeric and can have length up to 50 characters. Email must have one ‘@’ and ‘.’ symbol.
- **Membership date:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
- **Valid up to:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
- **Password:** Alphanumeric of length in the range of 4 to 15 characters. Blank spaces are not allowed. However, special characters are allowed. Initially contains 8 digits randomly generated number (auto-generated).

(vi) Issue Book

This form will be accessible only to the system administrator and library staff. It will allow him/her to issue existing book(s) to the student(s).

Issue Status:		
Accession No.	Title	Expected

Reservation Status:			
Accession No.	Title	Reservation Date	Expected

Various fields available on this form will be:

- **Membership ID:** Numeric and will have value from 100 to 5999.
- **Accession number:** Numeric and will have value from 10 to 99999.
- **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
- Issue status (the following member information will be displayed):
 - ◆ **Accession number:** Numeric and will have value from 10 to 99999.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
 - ◆ **Expected:** Date of return. It will be of format mm/dd/yyyy and will have 10 alphanumeric characters.
- Reservation status (the following fields will be displayed)
 - ◆ **Accession number:** Numeric and will have value from 10 to 99999.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
 - ◆ **Reservation date:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
 - ◆ **Expected:** Date of return. It will be of format mm/dd/yyyy and will have 10 alphanumeric characters.

(vii) Return Book

This form will be accessible only to the system administrator and library staff. It will allow him/her to return existing book(s) from the student(s).

Various fields available on this form will be:

- **Membership ID:** Numeric and will have value from 100 to 5999.
- **Accession number:** Numeric and will have value from 10 to 99999.
- **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
- Issue status (the following member information will be displayed):
 - ◆ **Accession number:** Numeric and will have value from 10 to 99999.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
 - ◆ **Expected:** Date of return. It will be of format mm/dd/yyyy and will have 10 alphanumeric characters.
- Reservation status (the following fields will be displayed):
 - ◆ **Accession number:** Numeric and will have value from 10 to 99999.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
 - ◆ **Reservation date:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
 - ◆ **Expected:** Date of return. It will be of format mm/dd/yyyy and will have 10 alphanumeric characters.

(viii) Reserve Book

This form will be accessible only to the system administrator and library staff. It will allow him/her to reserve issued book(s) for the student(s).

Issue Status:			
Accession No.	Title	Expected	

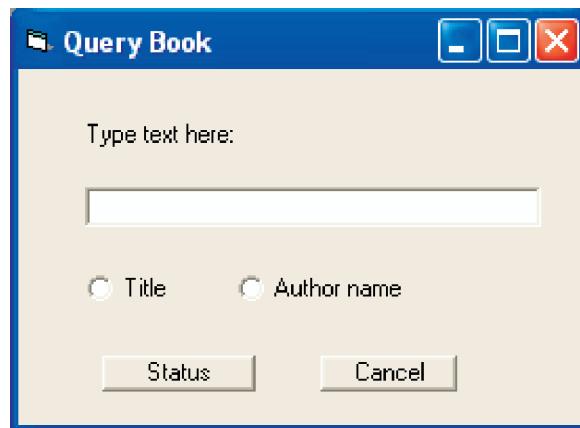
Reservation Status:			
Accession No.	Title	Reservation Date	Expected

Various fields available on this form will be:

- **Membership ID:** Numeric and will have value from 100 to 5999.
- **Accession number:** Numeric and will have value from 10 to 99999.
- **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
- Issue status (the following member information will be displayed):
 - ◆ **Accession number:** Numeric and will have value from 10 to 99999.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
 - ◆ **Expected:** Date of return. It will be of format mm/dd/yyyy and will have 10 alphanumeric characters.
- Reservation status (the following fields will be displayed):
 - ◆ **Accession number:** Numeric and will have value from 10 to 99999.
 - ◆ **Title:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.
 - ◆ **Reservation date:** Will be of format mm/dd/yyyy. It will have 10 alphanumeric characters.
 - ◆ **Expected:** Date of return. It will be of format mm/dd/yyyy and will have 10 alphanumeric characters.

(ix) Query Book

This form will be accessible only to the system administrator and library staff. It will allow him/her to search whether a book is available in the library or not.

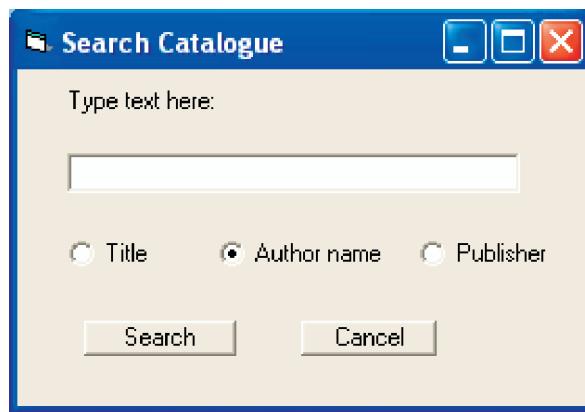


The field available on this form will be:

- **Type text here:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.

(x) Search Catalogue

This form will be accessible only to the system administrator/student/faculty/employee. It will allow him/her to search whether a book is available in the library.



The field available on this form will be:

- **Type text here:** Alphanumeric of length 3 to 100 characters. Special characters (except brackets) are not allowed. Numeric data will be allowed.

(xi) Generate Reports

The reports will be accessible to the system administrator and librarian. The system will generate different reports according to the specified criteria.

(i) Details of members

Name of the University	
Member ID _____	
Member details	

(ii) Details of books issued to members

Name of the University			
Member ID _____			
Books issued			
Book ID	Book name	Authors	Publisher

(iii) Status of fine

Name of the University		
Member ID _____		
Fine Total _____		
Book ID	Book name	Fine amount

3.1.2 Hardware Interfaces

As stated in section 2.1.3.

3.1.3 Software Interfaces

As stated in section 2.1.4.

3.1.4 Communication Interfaces

None

3.2 Functional Requirements

3.2.1 Login

A. Use Case Description

1. Introduction This use case documents the steps that must be followed in order to log into the LMS
2. Actors <ul style="list-style-type: none"> Administrator Data Entry Operator Library staff
3. Precondition The user must have valid login ID and password.
4. Postcondition If the use case is successful, the actor is logged into the system. If not, the system state remains unchanged.
5. Basic Flow Starts when actor wishes into login into the LMS. <ol style="list-style-type: none"> The system requests that the actor specify the function he/she would like to perform (either Login, Change Password). Once the actor provides the requested information, one of the following flows is executed: <ul style="list-style-type: none"> If the actor selects “Login”, the Login flow is executed. If the actor selects “Change Password”, the Change Password flow is executed.

Basic Flow 1: Login

- (i) The system requests that the actor enter his/her login ID and password information.
- (ii) The actor enters his/her login ID and password.
- (iii) The actor enters into the system.

Basic Flow 2: Change Password

- (i) The system requests that the actor enter login ID, old password, new password and confirm the new password information.
- (ii) The actor enters login ID, old password, new password and confirms the new password information.
- (iii) The system validates the entered new password and password change is confirmed.

6. Alternative Flows**Alternative Flow 1: Invalid Login ID/Password**

If in the Login flow, the actor enters an invalid login ID and/or password or leaves the login ID and/or password empty, the system displays an error message. The actor returns to the beginning of the basic flow.

Alternative Flow 2: Invalid Entry

If in the Change Password flow, the actor enters an invalid login ID, old password, new password or the new password does not match with the confirm password, the system displays an error message. The actor returns to the beginning of the basic flow.

Alternative Flow 3: User Exits

This allows the user to exit during the use case. The use case ends.

7. Special Requirement

None

8. Associated Use Cases

None

B. Validity Checks

- (i) Every user will have a unique login ID.
- (ii) Login ID cannot be blank.
- (iii) Login ID can only have 4 to 15 characters.
- (iv) Login ID will not accept special characters and blank spaces.
- (v) Password cannot be blank.
- (vi) Length of password can only be 4 to 15 digits.
- (vii) Alphabets, digits, hyphen and underscore characters are allowed in the password field.
- (viii) Password will not accept blank spaces.

C. Sequencing Information

None

D. Error Handling/Response to Abnormal Situations

If any of the validation flows does not hold true, appropriate error message will be prompted to the user for doing the needful.

3.2.2 Maintain Book Details

A. Use Case Description

Introduction
This use case documents the steps that the administrator/DEO must follow in order to maintain book details and add, update, delete and view book information.
Actors
Administrator Data Entry Operator
Precondition
The administrator/DEO must be logged into the system before this use case begins.
Postcondition
If the use case is successful, then book information is added, updated, deleted or viewed. Otherwise, the system state is unchanged.
Flow of Events
Basic Flow
This use case starts when the administrator/DEO wishes to add/update/delete/view book information.
<ol style="list-style-type: none"> 1. The system requests that the administrator/DEO specify the function he/she would like to perform (either Add a book, Update a book, Delete a book or View a book). 2. Once the administrator/DEO provides the requested information, one of the following subflows is executed: <ul style="list-style-type: none"> • If the administrator/DEO selects “Add a Book”, the Add a Book subflow is executed. • If the administrator/DEO selects “Update a Book”, the Update a Book subflow is executed. • If the administrator/DEO selects “Delete a Book”, the Delete a Book subflow is executed. • If the administrator/DEO selects “View a Book”, the View a Book subflow is executed.
Basic Flow 1: Add a Book
The system requests that the administrator/DEO enter the book information. This includes:
<ul style="list-style-type: none"> • Accession number • Subject descriptor • ISBN • Title • Language • Author • Publisher
Once the administrator/DEO provides the requested information, the book is added to the system.

Basic Flow 2: Update a Book

1. The system requests that the administrator/DEO enter the accession number.
2. The administrator/DEO enters the accession number.
3. The system retrieves and displays the book information.
4. The administrator/DEO makes the desired changes to the book information. This includes any of the information specified in the **Add a Book** subflow.
5. Once the administrator/DEO updates the necessary information, the system updates the book information with the updated information.

Basic Flow 3: Delete a Book

1. The system requests that the administrator/DEO specify the accession number.
2. The administrator/DEO enters the accession number. The system retrieves and displays the required information.
3. The system prompts the administrator/DEO to confirm the deletion of the book record.
4. The administrator/DEO verifies the deletion.
5. The system deletes the record.

Basic Flow 4: View a Book

1. The system requests that the administrator/DEO specify the accession number.
2. The system retrieves and displays the book information.

Alternative Flows**Alternative Flow 1: Invalid Entry**

If in the **Add a Book** or **Update a Book** flow, the actor enters invalid accession number/ISBN/title/author/publisher/language/subject descriptor or leaves the accession number/ISBN/title/author/publisher/language/subject descriptor empty, the system displays an appropriate error message. The actor returns to the basic flow and may reenter the invalid entry.

Alternative Flow 2: Book Already Exists

If in the Add a Book flow, a book with a specified accession number already exists, the system displays an error message. The administrator returns to the basic flow and may reenter the book.

Alternative Flow 3: Book not Found

If in the **Update a Book**, **Delete a Book** or **View a Book** flow, the book information with the specified accession number does not exist, the system displays an error message. The administrator returns to the basic flow and may reenter the accession number.

Alternative Flow 4: Update Cancelled

If in the **Update a Book** flow, the administrator/DEO decides not to update the book, the update is cancelled and the **Basic Flow** is re-started at the beginning.

Alternative Flow 5: Delete Cancelled

If in the **Delete a Book** flow, the administrator/DEO decides not to delete the book, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

Alternative Flow 6: Deletion not Allowed

If in the **Delete a Book** flow, issue/return/reserve details of the book selected exist, then the system displays an error message. The administrator returns to the basic flow and may reenter the student membership number.

Alternative Flow 7: User Exits

This allows the user to exit at any time during the use case. The use case ends.

Special Requirements
None
Extension Points
None

B. Validity Checks

- (i) Only the administrator/DEO will be authorized to access the Book Details module.
- (ii) Every book will have a unique accession number.
- (iii) Accession number cannot be blank.
- (iv) Accession number can only have value from 10 to 99999 digits.
- (v) Subject descriptor cannot be blank.
- (vi) ISBN number cannot be blank.
- (vii) Length of ISBN number for any user can only be equal to 11 digits.
- (viii) ISBN number will not accept alphabets, special characters and blank spaces.
- (ix) Title cannot be blank.
- (x) Length of title can be of 3 to 100 characters.
- (xi) Title will only accept alphabetic characters, brackets, numeric digits and blank spaces.
- (xii) Language cannot be blank.
- (xiii) Author (first name and last name) cannot be blank.
 - (a) Length of first name and last name can be of 3 to 100 characters.
 - (b) First name and last name will not accept special characters and numeric digits.
- (xiv) Publisher cannot be blank.
- (xv) Length of first name and last name can be of 3 to 300 characters.
- (xvi) Publisher will not accept special characters and numeric digits.

C. Sequencing Information

None

D. Error Handling/Response to Abnormal Situations

If any of the validation flows does not hold true, appropriate error message will be prompted to the user for doing the needful.

3.2.3 Maintain Student Details

A. Use Case Description

Introduction
This use case documents the steps that the administrator/DEO must follow in order to maintain student membership details. This includes adding, updating, deleting and viewing student information
Actors
Administrator DEO
Precondition
The administrator/DEO must be logged into the system before this use case begins.

Postconditions

If the use case is successful, then student information is added/updated/deleted/viewed from the system. Otherwise, the system state is unchanged.

Flow of Events**Basic Flow**

This use case starts when the administrator/DEO wishes to add, update, delete or view student information from the system.

1. The system requests that the administrator/DEO specify the function he/she would like to perform (either Add a student, Update a student record, Delete a student record or View a student record).
2. Once the Administrator/DEO provides the requested information, one of the following subflows is executed:
 - If the administrator/DEO selects “Add a Student”, the **Add a Student** subflow is executed.
 - If the administrator/DEO selects “Update a Student”, the **Update a Student** subflow is executed.
 - If the administrator/DEO selects “Delete a Student”, the **Delete a Student** subflow is executed.
 - If the Administrator/DEO selects “View a Student”, the **View a Student** subflow is executed.

Basic Flow 1: Add a Student

The system requests that the administrator/DEO enter the user information. This includes:

- Membership number
- Photograph
- Roll No.
- Name
- School
- Programme
- Father’s name
- Date of birth
- Address
- Telephone
- Email
- Membership date
- Valid up to
- Password

Once the administrator/DEO provides the requested information, the system checks that student Membership number is unique. The student is added to the system.

Basic Flow 2: Update a Student

1. The system requests that the administrator/DEO enter the student’s membership number.
2. The administrator/DEO enters the student’s membership number.

3. The system retrieves and displays the student's information.
4. The administrator/DEO makes the desired changes to the student information. This includes any of the information specified in the **Add a Student** subflow.
5. Once the administrator/DEO updates the necessary information, the system updates the student record with the updated information.

Basic Flow 3: Delete a Student

1. The system requests that the administrator/DEO specify the membership number of the student.
2. The administrator/DEO enters the membership number. The system retrieves and displays the student information.
3. The system prompts the Administrator/DEO to confirm the deletion of the student record.
4. The administrator/DEO verifies the deletion.
5. The system deletes the record.

Basic Flow 4: View a Student

1. The system requests that the administrator/DEO specify the membership number.
2. The system retrieves and displays the student information.

Alternative Flows

Alternative Flow 1: Invalid Entry

If in the **Add a Student** or **Update a Student** flow, the actor enters invalid photograph/Roll No./Name/School/Programme/Father's name/Date of Birth/Address/Telephone/Email/Membership date/Valid up to/Password or leaves the photograph/Roll No./Name/School/Programme/Father's name/Date of Birth/Address/Telephone/Email/Membership date/Valid up to/Password empty, the system displays an appropriate error message. The actor returns to the basic flow and may reenter the invalid entry.

Alternative Flow 2: Student Already Exists

If in the Add a Student flow, a student with a specified membership number already exists, the system displays an error message. The administrator returns to the basic flow and may reenter the student information.

Alternative Flow 3: Student not Found

If in the **Update a Student**, **Delete a Student** or **View a Student** flow, the student information with the specified membership number does not exist, the system displays an error message. The administrator returns to the basic flow and may reenter the membership number.

Alternative Flow 4: Update Cancelled

If in the **Update a Student** flow, the administrator decides not to update the student, the update is cancelled and the **Basic Flow** is re-started at the beginning.

Alternative Flow 5: Delete Cancelled

If in the **Delete a Student** flow, the administrator decides not to delete the student, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

Alternative Flow 6: Deletion not Allowed

If in the **Delete a Student** flow, issue/return/reserve details of the student selected exist, then the system displays an error message. The administrator returns to the basic flow and may reenter the membership number.

Alternative Flow 7: User Exits
This allows the user to exit at any time during the use case. The use case ends.
Special Requirements
None

Extension Points

None

B. Validity Checks

- (i) Only the administrator/DEO will be authorized to access the Student Membership Details module.
- (ii) Every student will have a unique membership number.
- (iii) Membership number can only have value from 100 to 5999 digits.
- (iv) Membership number will not accept alphabets, special characters and blank spaces.
- (v) Every student will have a unique membership number.
- (vi) Roll No. cannot be blank.
- (vii) Length of Roll No. for any user can only be equal to 11 digits.
- (viii) Roll No. cannot contain alphabets, special characters and blank spaces.
- (ix) Student name cannot be blank.
- (x) Length of student name can be of 3 to 50 characters.
- (xi) Student name will only accept alphabetic characters and blank spaces.
- (xii) School name cannot be blank.
- (xiii) Programme name cannot be blank.
- (xiv) Father's name cannot be blank.
- (xv) Father's name cannot include special characters and digits, but blank spaces are allowed.
- (xvi) Father's name can have length 3 to 50 characters.
- (xvii) Date of birth cannot be blank.
- (xviii) Address cannot be blank.
- (xix) Address can have length up to 10 to 200 characters.
- (xx) Phone cannot be blank.
- (xxi) Phone cannot include alphabets, special characters and blank spaces.
- (xxii) Phone can be up to 11 digits.
- (xxiii) Email cannot be blank.
- (xxiv) Email can have up to 50 characters.
- (xxv) Email should contain @ and . characters.
- (xxvi) Email cannot include blank spaces.
- (xxvii) Password cannot be blank (initially auto-generated of 8 digits).
- (xxviii) Password can have length from 4 to 15 characters.
- (xxix) Alphabets, digits, hyphen and underscore characters are allowed in the password field. However, blank spaces are not allowed.

C. Sequencing Information

None

D. Error Handling/Response to Abnormal Situations

If any of the validation/sequencing flows does not hold true, appropriate error message will be prompted to the administrator for doing the needful.

3.2.4 Issue Book

A. Use Case Description

Introduction
This use case documents the steps that must be followed in order to get a book issued.
Actors
Administrator Library staff
Precondition
The Administrator/Library staff must be logged into the system before the use case begins.
Postcondition
If the use case is successful, a book is issued to the student/faculty/employee and the database is updated, else the system state remains unchanged.
Event Flow
Basic Flow
<ol style="list-style-type: none"> 1. Student/faculty/employee membership number is read through the bar code reader. 2. The system displays information about the student/faculty/employee. 3. Book information is read through the bar code reader. 4. The system checks the reservation status of the book. 5. The system validates the membership and number of books issued in the account. 6. The system checks whether the member fine exceeds the maximum limit of fine. 7. The book is issued for the specified number of days and the return date of the book is calculated. 8. The book and student/faculty/employee information is saved into the database. 9. If the book is reserved by the same member, then it is marked as unreserved.
Alternate Flows
Alternative Flow 1: Unauthorized Member
If the system does not validate the member's (student/faculty/employee) membership number (due to membership expiry or any other reason), then an error message is flagged and the use case ends.
Alternative Flow 2: Account is Full
If the student/faculty/employee has requested a book in Account and it is full, i.e. he/she has already maximum number of books issued on his/her name, then the request for issue is denied and the use case ends.
Alternative Flow 3: Book is Already Reserved
If the book is already reserved by some other member of the library, then the request for issue is denied and the use case ends.

Alternative Flow 4: Fine Exceeds the Specified Limit If the fine of the student exceeds the specified limit, then the request for issue is denied and the use case ends.
Alternative Flow 5: Unable to Read Entry If the barcodeID or memberID is unreadable, then an error message is flagged and the use case returns to the beginning of the basic flow.
Alternative Flow 6: Invalid Entry If in the issue book flow, the actor enters an invalid barcodeID and/or memberID or leaves the barcodeID and/or memberID empty, the system displays an error message. The actor returns to the beginning of the basic flow.
Alternative Flow 7: User Exits This allows the user to exit at any time during the use case. The use case ends.
Special Requirements None
Associated Use Cases None

B. Validity Checks

- (i) Only the administrator/library staff will be authorized to access the Student Membership Details module.
- (ii) Every student will have a unique membership number.
- (iii) Membership number can only have value from 100 to 5999 digits.
- (iv) Membership number will not accept alphabets, special characters and blank spaces.
- (v) Membership list cannot be blank.
- (vi) Accession number cannot be blank.
- (vii) Accession number can only have value from 10 to 99999 digits.
- (viii) Date cannot be blank.
- (ix) Title cannot be blank.
- (x) Length of title can be of 3 to 100 characters.
- (xi) Title will only accept alphabetic characters, brackets, numeric digits and blank spaces.

C. Sequencing Information

Book and student membership details should be available in the system.

D. Error Handling/Response to Abnormal Situations

If any of the validation/sequencing flows does not hold true, appropriate error message will be prompted to the administrator for doing the needful.

3.2.5 Return Book

A. Return Book Use Case Description

Introduction
This use case documents the steps that must be followed in order to return a book.
Actors
Administrator Library staff
Precondition
The administrator/library staff must be logged into the system before the use case begins.
Postcondition
If the use case is successful, the book is returned to the library and if needed, the “fine calculation” use case is called; otherwise, the system state is unchanged.
Event Flow
Basic Flow
<ol style="list-style-type: none"> 1. The book information is read from the bar code of the book through the bar code reader. 2. The student/faculty/employee detail on whose name the books were issued is displayed on the system. The date of issue and return is also displayed. 3. The administrator/library staff checks the stamp on the book to check the duration of issue of book. 4. The database is updated and the book bar code is also updated. 5. The date stamp on the book is cancelled.
Alternate Flows
Alternative Flow 1: Late Return of Book
If the duration for which the book has been kept by the student is more than 15 days, then a fine calculation use case is called.
Alternative Flow 2: User Exits
This allows the user to exit at any time during the use case. The use case ends.
Special Requirements
None
Associated Use Cases
None

B. Validity Checks

- (i) Only the administrator/library staff will be authorized to access the Student Membership Details module.
- (ii) Every student will have a unique membership number.
- (iii) Membership number can only have value from 100 to 5999 digits.
- (iv) Membership number will not accept alphabets, special characters and blank spaces.
- (v) Membership list cannot be blank.

- (vi) Accession number cannot be blank.
- (vii) Accession number can only have value from 10 to 99999 digits.
- (viii) Date cannot be blank.
- (ix) Title cannot be blank.
- (x) Length of title can be of 3 to 100 characters.
- (xi) Title will only accept alphabetic characters, brackets, numeric digits and blank spaces.

C. Sequencing Information

Book, student membership and issue details should be available in the system.

D. Error Handling/Response to Abnormal Situations

If any of the validation/sequencing flows does not hold true, appropriate error message will be prompted to the administrator for doing the needful.

3.2.6 Fine Calculation

A. Use Case Description

Introduction
This use case documents the procedure of calculating the fine, if the book is returned after the specified due date.
Actors
None
Precondition
The book is returned after the specified due date.
Postcondition
If the use case is successful, the total fine amount is calculated and displayed to the operator.
Event Flow
Basic Flow
This use case starts when a “Return Book” use case calls this use case.
<ul style="list-style-type: none"> • The system fetches current date from the computer and compares it with the specified due date. • The fine amount is calculated, and the specified fine is charged for each day. • The use case ends.
Alternate Flow
None
Special Requirements
None
Associated Use Cases
Return Book

3.2.7 Reserve Book

A. Use Case Description

Introduction
This use case documents the steps that must be followed in order to reserve a book for 24 hours for a student.
Actors
Administrator Library staff
Precondition
The administrator/library staff must be logged into the system before this use case begins.
Postcondition
If the use case is successful, the requested book is reserved for 24 hours for the student, else the system state remains unchanged.
Event Flow
Basic Flow
<ol style="list-style-type: none"> 1. The administrator/library staff initiates the “query book” use case to determine whether a copy of book is available for issue. 2. If all copies of the book have been issued, then information about the book reserved is written on the student’s bar code. 3. The database is updated and the student’s enrollment number is saved along with the book information.
Alternate Flows
Alternative Flow 1: Book Does not Exist
If the book is not at all available in the library, then the request for reserving book is rejected.
Alternative Flow 2: User Exits
This allows the user to exit at any time during the use case. The use case ends.
Special Requirements
None
Associated Use Cases
None

B. Validity Checks

- (i) Only the administrator/library staff will be authorized to access the Student Membership Details module.
- (ii) Every student will have a unique membership number.
- (iii) Membership number can only have value from 100 to 5999 digits.
- (iv) Membership number will not accept alphabets, special characters and blank spaces.
- (v) Membership list cannot be blank.
- (vi) Accession number cannot be blank.
- (vii) Accession number can only have value from 10 to 99999 digits.
- (viii) Date cannot be blank.

- (ix) Title cannot be blank.
- (x) Length of title can be of 3 to 100 characters.
- (xi) Title will only accept alphabetic characters, brackets, numeric digits and blank spaces.

C. Sequencing Information

Book and student membership and issue details should be available in the system.

D. Error Handling/Response to Abnormal Situations

If any of the validation/sequencing flows does not hold true, appropriate error message will be prompted to the administrator for doing the needful.

3.2.8 Query Book

A. Use Case Description

Introduction
This use case documents the steps that must be followed in order to query a book by a student.
Actors
Administrator Library staff
Precondition
The operator must be logged into the system before this use case begins.
Postcondition
If the use case is successful, the system shows details of book queried on the screen, else the system state remains unchanged.
Event Flow
Basic Flow
<ol style="list-style-type: none"> 1. The operator enters any one of the following information: <ul style="list-style-type: none"> • Full or partial name of the book. • Full or partial name of the author. 2. The system conducts a search for the book. 3. The system displays the details of the book, with the number of copies available.
Alternate Flows
Alternative Flow 1: Book Does not Exist
If the book name or author name does not return any reference to the book, a message reporting the failure of search is displayed.
Alternative Flow 2: User Exits
This allows the user to exit at any time during the use case. The use case ends.
Special Requirements
None
Associated Use Cases
None

3.2.9 Search Book

A. Use Case Description

Introduction

This use case documents the procedure for searching a book based on the specified criteria, which are:

- Availability of a particular book of any title
- Availability of book of any particular author
- Availability of book of any particular publisher
- Availability of book of any particular subject

The search results can direct the user to get a book reserved, if he/she wishes to do so.

Actors

Student

Faculty

Employee

Precondition

The administrator/student/faculty/employee must be logged into the system before this use case.

Postcondition

If the use case is successful, the book details are displayed and if needed, the “reserve book” use case is called up; otherwise the system state is unchanged.

Event Flow

Basic Flow

This use case starts when a student wants to search for a particular book.

- The system displays the various search criteria to the user.
- The user selects the search criteria.
- If the criterion is “Availability of a particular book of any title”, then the user needs to enter the book title.
- Otherwise, if the criterion is “Availability of book of any particular author”, then the user needs to enter the author of the book.
- If the criterion is “Availability of book of any particular publisher”, then the user needs to enter the publisher of the book.
- If the criterion is “Availability of book of any particular subject”, then the user needs to enter the subject of the book.
- The result is displayed to the user.
- The use case ends.

Alternate Flow

Alternative Flow 1: User Exits

This allows the user to exit at any time during the use case. The use case ends.

Special Requirements

None

Associated Use Cases

None

3.2.10 Report Generation

A. Use Case Description

Introduction This use case documents the steps that must be followed in order to generate a report of all books available at current time in the library.
Actors Administrator Librarian
Precondition The operator must be logged onto the system before the use case begins.
Postcondition If this use case is successful, the system displays the details of all the books available in the library at the current time. No changes are made to the database. Else, no report is displayed.
Event Flow Basic Flow <ol style="list-style-type: none">1. The operator issues the command to generate the report.2. The system displays a report including details of all the books available in the library at the current time. Alternate Flow None
Special Requirements None
Associated Use Cases None

3.3 Performance Requirements

- (a) Should run on 500 MHz, 512 MB RAM machine.
- (b) Responses should be within 2 seconds.

3.4 Design Constraints

None

3.5 Software System Attributes

Usability

The application will be user-friendly and easy to operate and the functions will be easily understandable.

Reliability

The applications will be available to the students throughout the registration period and have a high degree of fault tolerance.

Security

The application will be password protected. Users will have to enter correct login ID and password to access the application.

Maintainability

The application will be designed in a maintainable manner. It will be easy to incorporate new requirements in the individual modules.

Portability

The application will be easily portable on any windows-based system that has SQL Server installed.

3.6 Logical Database Requirements

The following information will be placed in a database:

Table name	Description
Login	Records the login details of the user.
Book	Records the details of the books in the library.
Member	Stores the details of the members in the library.
Transaction	Stores the details of transactions in the library.
FineStatus	Records the fine status details of the book.
Student	Records student details.
Faculty	Records faculty details.
Employee	Records employee details.
Reserve	Records the reservation details of the book.

3.7 Other Requirements

None

References

- Abren, A., Moore, J., Bourque, P. and Dupis, R., *Guide to the Software Engineering body of Knowledge—2004 Version*. IEEE Computer Society, 2004.
- Aggarwal, K.K. and Singh, Y., *Software Engineering: Programs, Documentation, Operating Procedures*. New Delhi: New Age International Publishers, 2008.
- Aggarwal, K.K., Singh, Y., Kaur, A. and Malhotra, R., Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: A replicated case study. *Software Process Improvement and Practice*, **16**(1): 39–62, 2009.
- Aggarwal, K.K., Singh, Y., Kaur, A. and Malhotra, R., Empirical study of object-oriented metrics. *Journal of Object-Technology*, **5**(8): 149–173, 2006.
- Albrecht, A. and Gaffney, J., Software function, source lines of code and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, **9**(6): 639–648, 1983.
- ANSI, *Standard Glossary of Software Engineering Terminology, STD-729-1991*. ANSI/IEEE, 1991.
- Arnold, R.S., *A Roadmap to Software Re-engineering Technology: Software Re-engineering—A Tutorial*. Los Alamitos, CA: IEEE Computer Society Press, 3–22, 1993.
- Basili, V. and Reiter, R., Evaluating automable measures of software models. *IEEE Workshop on Quantitative Software Models*, 107–116, 1979.
- Basili, V. and Turner, A., Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, **1**(4): 390–396, 1975.
- Belady, L. and Lehman, W., A model of large program development. *IBM Systems Journal*, **15**(3): 225–252, 1976.
- Benlarbi, S. and Melo, W., Polymorphism measures for early risk prediction. In: *Proceeding of the 21st International Conference on Software Engineering*, 334–344, 1999.
- Bevan, N., Measuring usability as quality of use. *Software Quality Journal*, **4**: 115–150, 1995.

- Bieman, J. and Kang, B., Cohesion and reuse in an object oriented system. In: *Proceedings of the CM Symp. Software Reusability (SSR'95)*, 259–262, 1995.
- Binder, R.V., Design for testability in object-oriented systems. *Communication of the ACM*, **37**(9): 87–101, 1994.
- Bittner, K. and Spence, I., *Use Case Modeling*. Boston, MA: Addison-Wesley, 2003.
- Boehm, B., A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, **11**(4): 14–24, 1986.
- Boehm, B., *Characteristics of Software Quality*. Amsterdam: North Holland, 1978.
- Boehm, B., *IEEE Tutorial on Software Risk Management*. New York: IEEE Computer Society Press, 1989.
- Boehm, B., *Software Engineering Economics*. Upper Saddle River, NJ: Prentice-Hall, 1981.
- Boggs, W. and Boggs, M., *Mastering UML with Rational Rose*. New Delhi: BPB Publications, 2002.
- Booch, G., *Object-Oriented Design with Applications*. Redwood City, CA: Benjamin-Cummings, 1991.
- Briand, L., Daly, W. and Wust, J., A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, **25**: 91–121, 1999.
- Briand, L., Devanbu, P. and Melo, W., An investigation into coupling measures for C++. In: *Proceedings of the ICSE 97*, Boston, USA, 1997.
- Brooks, F.P., *No Silver Bullets—The Mythical Man Month: Essay on Software Engineering*, 2nd ed. Reading, MA: Addison-Wesley, Longman, pp. 179–209, 1995.
- Canfora, G. and Cimitile, A., *Software Maintenance*, pp. 1–33, 2000.
- Chidamber, S.R. and Kamerer, C.F., A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, **20**(6): 476–493, 1994.
- Coad, P. and Yourdon, E., *Object-Oriented Design*, 1st ed. Englewood Cliffs, NJ: Prentice-Hall, 1991.
- Coad, P. and Yourdon, E., *Object-Oriented Analysis*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- Cockburn, A., *Writing Effective Use Cases*. New Delhi: Pearson Education, 2001.
- Conte, S.D., Dunsmore, H.E. and Shen, V.Y., *Software Engineering Metrics and Models*. Redwood City, CA: Benjamin-Cummings, 1986.
- Costagliola, G. Ferrucci, F. and Tortora, G., Class point: An approach for the size estimation of object-oriented systems. *IEEE Transactions on Software Engineering*, **31**(1): 52–74, January 2005.
- DeMarco, T., *Controlling Software Projects: Management, Measurement & Estimation*. New Jersey: Yourdon Press, 1982.
- Fenton, N.E. and Pfleeger, S.L., *Software Metrics*. Singapore: Thomson Learning, 2004.

- Fournier, G., *Essential Software Testing—A Use Case Approach*. Boca Raton, FL: CRC Press, 2009.
- Fritz, B., et al., Software engineering: A report on a conference sponsored by NATO Science Committee. NATO, 1968.
- Goodman, P., *Practical Implementation of Software Metrics*. UK: McGraw Hill, 1993.
- Hair, J., Anderson, R., Tatham, R. and Black, W., *Multivariate Data Analysis*, 5th ed. Englewood Cliffs, NJ: Pearson Education, 2006.
- Harrison, R., Counsell, S.J. and Nithi, R.V., Empirical assessment of the effect of inheritance on the maintainability of object-oriented systems. *IEEE Transactions on Software Engineering*, **24**(6): 491–496, 1998.
- Henderson-Sellers, B. and Edwards, J., Object oriented systems life cycle. *Communications of the ACM*, **33**(9): 142–159, 1990.
- Henderson-Sellers, B., *Object-Oriented Metrics: Measures of Complexity*. Englewood, Cliffs, NJ: Prentice-Hall, 1996.
- Hopkins, W.G., A new view of statistics, Sport Science, 2003.
- IEEE Std 610.92, *Standard Glossary of Software Engineering Terminology*. Los Alamitos, CA: IEEE Computer Society Press, 1990.
- IEEE, IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830–1998), 1998.
- IEEE, Standard Glossary of Software Engineering Terminology, 2001.
- IFPUG. *Function Point Counting Practices Manual, Release 4.0*. Westerville, Ohio: International Function Point Users Group, 1994.
- Jacobson, I.V. et al., *Object Oriented Software Engineering*. New Delhi: Addison-Wesley, 1999.
- Jorgenson, P.C., *Software Testing: A Craftsman's Approach*, 3rd ed. USA: Auerbach Publications, 2007.
- Karner, G., Metrics of Objectory, Thesis. Linkoping University, Sweden, 1993.
- Lee, Y., Liang, B., Wu, S. and Wang, F., Measuring the coupling and cohesion of an object-oriented program based on information flow. In: *Proceedings of the International Conference on Software Quality*, Maribor, Slovenia, 81–90, 1995.
- Li, W. and Henry, S., Object oriented metrics that predict maintainability. *Journal of Systems and Software*, **23**(2): 111–122, 1993.
- Lientz, B.P. and Swanson, E.B., *Software Maintenance Management*. Reading, MA: Addison-Wesley, 1980.
- Lorenz, M. and Kidd, J., *Object Oriented Software Metrics*. Englewood Cliffs, NJ: Prentice-Hall, 1994.
- McCabe, T.J., A complexity metric. *IEEE Transactions on Software Engineering*, **2**(4): 308–320, 1976.
- McCall, J., Richards, P. and Walters, G., *Factors in Software Quality*, Vol I. Springfield, Virginia: National Technical Information Service, 1977.

- Myers, G.J., *The Art of Software Testing*. New York: John Wiley & Sons, 2004.
- Pfleeger, S.L., *Software Engineering Theory and Practice*. Upper Saddle River, NJ: Prentice-Hall, 2001.
- Pressman, R.S., *Software Engineering: A Practitioner's Approach*. New York: Tata-McGraw Hill, 2005.
- Putnam, L.H. and Putnam, D.T., A data verification of the software fourth power trade-off law. In: *Proc. Int'l. Soc. Parametric Analysts Conf.*, 1984.
- Quatrani, T., *Visual Modeling with Rational Rose 2002 and UML*. Boston: Addison-Wesley, 2003.
- Rational Software, Rational Unified Process Version, 2002.
- Rovce, W.W., Managing the development of large software systems. *Proceedings, IEEE WESCON*, pp. 1–9, August 1970.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 2004.
- Schach, S.R., *Classical and Object Oriented Software Engineering with UML and Java*. USA: McGraw Hill, 1999.
- Stark, G., Durst, R. and Pelnik, T., An evaluation of software testing metrics for NASA's Mission Control Center. *Software Quality Journal*, 1: 115–132, 1992.
- Symons, C.R., Function point analysis: Difficulties and improvements. *IEEE Transactions on Software Engineering*, 14(1): 1988.
- Tegarden, D.P. and Sheetz, S.D., Object-oriented System Complexity: An integrated model of structure and perceptions. In: *OOPSLA'92 Workshop on Metrics for Object-oriented Software Developments*, Vancouver, Canada, 1992.
- Thayer, R.H. and Dorfman, M., *Software Requirements Engineering*. Los Angeles: IEEE Computer Society, 1997.
- Wake, W.C., *Extreme Programming Explored*. Boston, MA: Addison-Wesley Professional, 2002.
- Weyuker, E., Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14: 1357–1365, 1998.
- Wiegers, K.E., *Software Requirements*. Washington, USA: Microsoft Press, 1999.
- Wikipedia, The free encyclopedia, www.wikipedia.org/wiki/, 2010.
- Yap, L.M. and Henderson-Sellers, B., Consistency considerations of object-oriented class libraries. Technical report, University of New South Wales, 1993.
- Young, R.R., *Effective Requirements Practices*. Boston, MA: Addison-Wesley Longman, 2001.
- Yourdon, E. and Constantine, L., *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
- Zuse, H., *Software Complexity: Measures and Methods*. Berlin: Walter de Gruyter, 1990.

Answers to Multiple Choice Questions

Chapter 1

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (a) | 2. (d) | 3. (d) | 4. (c) | 5. (b) | 6. (b) | 7. (d) |
| 8. (d) | 9. (b) | 10. (a) | 11. (b) | 12. (b) | 13. (c) | 14. (c) |
| 15. (a) | 16. (b) | 17. (c) | 18. (b) | 19. (a) | 20. (d) | 21. (c) |
| 22. (b) | 23. (a) | 24. (c) | 25. (d) | 26. (a) | 27. (c) | 28. (c) |
| 29. (d) | 30. (b) | 31. (d) | 32. (b) | 33. (c) | 34. (a) | 35. (c) |
| 36. (d) | 37. (d) | 38. (a) | 39. (b) | 40. (c) | 41. (b) | 42. (a) |
| 43. (d) | 44. (d) | 45. (a) | 46. (a) | 47. (b) | 48. (c) | 49. (c) |
| 50. (c) | | | | | | |

Chapter 2

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (b) | 2. (c) | 3. (b) | 4. (a) | 5. (c) | 6. (a) | 7. (d) |
| 8. (b) | 9. (a) | 10. (c) | 11. (d) | 12. (a) | 13. (a) | 14. (d) |
| 15. (b) | 16. (c) | 17. (a) | 18. (c) | 19. (b) | 20. (d) | 21. (d) |
| 22. (a) | 23. (a) | 24. (b) | 25. (d) | 26. (b) | 27. (a) | 28. (c) |
| 29. (d) | 30. (c) | 31. (a) | 32. (b) | 33. (d) | 34. (d) | 35. (a) |

Chapter 3

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (a) | 2. (b) | 3. (d) | 4. (a) | 5. (a) | 6. (c) | 7. (d) |
| 8. (b) | 9. (d) | 10. (b) | 11. (c) | 12. (b) | 13. (b) | 14. (a) |
| 15. (d) | 16. (a) | 17. (d) | 18. (c) | 19. (b) | 20. (a) | 21. (d) |
| 22. (c) | 23. (d) | 24. (c) | 25. (a) | 26. (b) | 27. (d) | 28. (c) |
| 29. (b) | 30. (d) | 31. (d) | 32. (c) | 33. (a) | 34. (c) | 35. (d) |

Chapter 4

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (a) | 2. (d) | 3. (c) | 4. (c) | 5. (a) | 6. (d) | 7. (d) |
| 8. (b) | 9. (b) | 10. (c) | 11. (c) | 12. (d) | 13. (c) | 14. (d) |
| 15. (a) | 16. (d) | 17. (d) | 18. (a) | 19. (a) | 20. (c) | 21. (d) |
| 22. (b) | 23. (c) | 24. (d) | 25. (b) | 26. (a) | 27. (b) | 28. (c) |
| 29. (c) | 30. (a) | | | | | |

Chapter 5

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (a) | 2. (b) | 3. (d) | 4. (b) | 5. (b) | 6. (c) | 7. (c) |
| 8. (b) | 9. (a) | 10. (c) | 11. (b) | 12. (a) | 13. (c) | 14. (a) |
| 15. (b) | 16. (c) | 17. (d) | 18. (b) | 19. (d) | 20. (d) | |

Chapter 6

- | | | | | | | |
|----------------|---------------|----------------|----------------|----------------|----------------|----------------|
| 1. (b) | 2. (a) | 3. (b) | 4. (a) | 5. (c) | 6. (c) | 7. (a) |
| 8. (b) | 9. (d) | 10. (c) | 11. (d) | 12. (d) | 13. (a) | 14. (b) |
| 15. (d) | | | | | | |

Chapter 7

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (b) | 2. (c) | 3. (b) | 4. (a) | 5. (a) | 6. (c) | 7. (d) |
| 8. (d) | 9. (a) | 10. (a) | 11. (b) | 12. (c) | 13. (b) | 14. (c) |
| 15. (b) | 16. (b) | 17. (d) | 18. (d) | 19. (d) | 20. (a) | |

Chapter 8

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (d) | 2. (d) | 3. (c) | 4. (b) | 5. (b) | 6. (a) | 7. (d) |
| 8. (c) | 9. (d) | 10. (d) | 11. (a) | 12. (d) | 13. (c) | 14. (a) |
| 15. (c) | 16. (b) | 17. (a) | 18. (b) | 19. (d) | 20. (d) | 21. (c) |
| 22. (d) | 23. (a) | 24. (b) | 25. (a) | 26. (b) | 27. (c) | 28. (d) |
| 29. (d) | 30. (b) | | | | | |

Chapter 9

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (c) | 2. (c) | 3. (b) | 4. (a) | 5. (d) | 6. (a) | 7. (c) |
| 8. (c) | 9. (b) | 10. (a) | 11. (c) | 12. (a) | 13. (d) | 14. (b) |
| 15. (c) | 16. (d) | 17. (a) | 18. (a) | 19. (a) | 20. (a) | 21. (a) |
| 22. (d) | 23. (a) | 24. (d) | 25. (b) | 26. (c) | 27. (a) | 28. (d) |
| 29. (d) | 30. (d) | 31. (c) | 32. (d) | 33. (d) | 34. (a) | 35. (d) |

Chapter 10

- | | | | | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 1. (b) | 2. (d) | 3. (c) | 4. (a) | 5. (a) | 6. (a) | 7. (b) |
| 8. (d) | 9. (a) | 10. (c) | 11. (c) | 12. (a) | 13. (d) | 14. (b) |
| 15. (a) | 16. (c) | 17. (d) | 18. (c) | 19. (b) | 20. (d) | |

Index

- α (alpha), 395
- ω (omega), 395
- Absolute scale, 307
- Abstraction, 15
- Acceptance testing, 42, 405
- Access specifiers, 7
- Action, 271
 - entries, 380
 - stubs, 380
- Activities, 49, 261
- Activity diagrams, 260
- Actor(s), 21, 73, 76
- Adaptability, 289
- Adaptive maintenance, 416
- Adjusted object points, 145
- Adjustment factor, 145
- Aggregation, 181
- Agile processes, 40
- Alpha testing, 405
- Analysis objects, 127
- Angular dimension, 38
- Annual change traffic, 425
- Annual maintenance effort, 425
- Artifacts, 50
- Association, 180
- Attribute(s), 8, 187
- Average inheritance depth, 335
- Base class, 11
- Baseline, 112
- Bath tub curve, 4
- Beta testing, 405
- Binary associations, 180
- Bivariate outliers, 313
- Black box testing, 356
- Boehm's software quality model, 297
- Booch methodology, 17
- Boundary classes, 176
- Boundary value analysis, 356
- Box length, 315
- Box plot, 313, 315
- Brainstorming, 148
 - sessions, 69
- Branch coverage, 327
- Branching, 262
- Bug, 24
- Build-and-fix model, 33, 55
- Capability maturity model, 301
- Case description, 78
- Centralized control structure, 209
- Change control, 114
 - authority, 114
 - board, 428
- Change management, 292
- Change procedure, 292
- Change request, 114
- Characteristics of a good requirement, 82
- Checklist, 352
- Chidamber and Kemerer, 328
- Class, 7
 - diagram, 192, 234
 - hierarchy depth, 247
 - point, 137

- point method, 134
testing, 393
- Class-attribute (CA) interaction, 330
- Class-method (CM) interaction, 330
- Class to leaf depth, 335
- Code smells, 282
- Cohesion, 244
- Collaboration(s), 22
diagrams, 222
- Commercially off-the-shelf (COTS), 6
- Component, 47
- Composition, 182
- Concurrency control and recovery, 275
- Condition
coverage, 327
entries, 380
stubs, 380
- Configuration
accounting, 426, 428
control, 426
identification, 426
management, 294, 426
- Construction, 54
- Control classes, 176
- Control object, 177
- Conventional approaches, 1
- Corrective maintenance, 415
- Correlation analysis, 319
- Coupling, 247, 328
between objects, 328
- Coverage analysis tools, 406
- CP₁ measure, 135
- CP₂ measure, 135
- Critical path method, 152
- Customers, 21, 64
- Data
abstraction coupling, 329
element types, 141
hiding, 7
independence, 275
- Decentralized control structure, 209
- Decision, 380
- Decision table-based testing, 380
- Decision to decision path graph, 385
- Dependency, 183
- Dependent variables, 319
- Depth of inheritance tree, 334
- Design clarity, 247
- Design constraints, 109
- Detailed design, 234
- Developers, 21
- Disciplines, 51
- Documentation, 282
- Documents, 3
- Domain classes, 175
- Drawback of XP, 43
- Dynamic
binding, 419
structure, 48
testing, 349
testing tools, 406
- Editions, 427
- Effort, 137
adjustment factor, 425
- Elaboration, 53
- Encapsulation, 8, 10
- Entity classes, 175
- Equivalence classes, 370
- Equivalence class testing, 376
- Equivalent mutants, 401
- Estimated program level, 322
- Event, 270
- Export or import coupling, 330
- Extend relationship, 75
- External attributes, 306
- External classes, 143
- Extreme programming, 40, 56
- Facilitated application specification technique, 70
- Facilitator, 69
- Failure, 24
- Fan-in, 324
- Fan-out, 324
- Fault, 24
- File type referenced, 142
- Flattening of classes, 403
- Focus of control, 205
- Fork, 262
- Forward engineering, 421
- Fountain model, 44, 56
- Functional audits, 291
- Functionality, 288
- Functional/regression testing tools, 406
- Functional requirements, 102
- Functional testing, 356
- Function overriding, 13
- Generalization, 183
- Guard condition, 262, 271

- Halstead software science metrics, 321
 Has-a, 181
 Histogram, 312
 History state, 273
 IEEE standard 830-1998, 86
 Implementation model, 54
 Inception, 52
 Include relationship, 76
 Independent paths, 388
 Independent variables, 319
 Information flow-based
 cohesion, 334
 coupling, 331
 inheritance coupling, 331
 Information hiding, 8
 Initial requirements document (IRD), 71
 Inspections, 350
 Integration, 35
 testing, 404
 Interaction diagram, 204
 Interclass testing, 404
 Interface classes, 176
 Internal
 attributes, 306
 classes, 143
 scale, 307
 Is-a, 11, 183
 Is-a relationship, 16
 ISO 9000, 299
 ISO 9001, 299
 ISO 9126, 300
 Iteration planning, 42, 52
 Iterations, 46
 Iterative enhancement, 55
 Iterative enhancement model, 37
 Jacobson's methodology, 19
 Join, 262
 Key classes, 125
 Key process areas, 303
 Killed mutants, 401
 Lack of cohesion in methods (LCOM) metric, 331
 Legacy systems, 422
 Lifeline, 205
 Links, 222
 Live mutants, 401
 Load, 406
 Logical database requirements, 109
 Loop coverage, 327
 Loose class cohesion, 332
 Lorenz and Kidd method for estimation, 124
 Maintainability, 289
 McCall's software quality, 296
 Mean, 310
 Measure(s), 22
 of central tendency, 310, 312
 of dispersion, 311
 Median, 310
 Message(s), 8, 205
 passing coupling, 329
 Method, 8
 Method-method (MM) interaction, 330
 Methodology, 16
 Metric, 307
 Mitigation strategies, 151
 Mode, 310
 Modelling, 20
 Most likely time, 160
 Multiple condition coverage, 327
 Multiplicity, 182
 Multivariate outliers, 313, 317
 Mutation
 score, 401
 testing, 397
 Nominal scale, 307
 Non-inheritance information flow-based coupling, 331
 Non-metric data, 307
 Non-structured interview, 66
 Normal curve, 312
 Number of
 ancestors, 335
 attributes, 135
 children, 335
 descendants, 335
 external methods, 135
 methods added, 335
 methods inherited, 335
 methods overridden, 335
 parents, 335
 service requested, 135
 Object(s), 6, 7
 composition, 15
 management group, 20

- Object-oriented design, 203
Object-oriented function point, 140
Object-oriented modelling, 20
Object-oriented software estimation, 124
Operating procedure manuals, 3
Operation(s), 189
 coverage, 327
Operator overloading, 15
Optimistic time, 160
Ordinal scale, 307
Outlier(s), 313
 analysis, 313
- Pair programming, 42
Path coverage, 327
Path testing, 385
Peer reviews, 350
Perfective maintenance, 416
Performance requirements, 109
Performance testing, 406
Persistent objects, 275
PERT, 160
Pessimistic time, 160
Physical audits, 291
Polymorphism, 13, 183
Postconditions, 79
Potential volume, 322
Preconditions, 79
Preventive maintenance, 416
Primary actors, 74
Process(es), 21, 305
 management tools, 406
 metrics, 306
Product(s), 21, 305
 metrics, 306
Productive period, 326
Program, 3
 evaluation review technique, 152
 graph, 385
 length, 321
 level, 322
Project control list, 37
Project repository, 52
Prototyping model, 36, 55, 71
- Quality
 assurance, 23
 attributes, 65
 charts, 327
 control, 23
 system, 289
- Quartile, 311
Questionnaire, 326
- Radial dimension, 38
Rational software, 46
Rational unified process, 46, 56
Ratio scales, 307
Record element types, 141
Redocumentation, 423
Refactoring, 42, 282
Regression testing, 428
Relationships, 180
Relative user efficiency, 326
Release, 427
 planning, 41
Reliability, 289
Requirements change management, 111
Requirements traceability, 112
Responsibility, 15, 22
Restructuring, 424
Reusability, 6, 282
Reuse ratio, 336
Reverse engineering, 421
Reviews, 291
Ripple effect, 417
Risk, 146, 147
 analysis and prioritization, 149
 avoidance strategies, 151
 identification, 148
 management, 147
 monitoring, 151
Robustness testing, 359
Roles, 49
Run-time polymorphism, 419
- Scatter plot(s), 313, 317
Scenario matrix, 81
Scenario scripts, 124
Secondary actors, 74
Sequence diagrams, 205
Services, 143
Simple classes and objects, 247
Size metrics, 320
Software, 3
 architecture description document, 54
 design, 1
 development effort, 425
 engineering, 2
 maintenance, 35, 414
 metrics, 22, 304
 quality, 22, 287

- re-engineering, 422
- rejuvenation, 421
- reliability, 22
- requirements specification, 86
- testing, 405
- validation, 349
- verification, 349
- verification techniques, 349
- Specialization ratio, 336
- Spiral, 55
 - model, 38
- SRS writer, 86
- Stable requirements, 37
- Stakeholder(s), 5, 64
- Standard deviation, 162, 311
- State, 269
 - start, 270
 - stop, 270
- State-based testing, 395
- Statechart diagrams, 268
- Statement coverage, 327
- State transitions, 270
- Static structure, 48
- Static testing, 349
 - tools, 405
- Stress, 406
- Structural testing, 385
- Structured
 - analysis, 174
 - interview, 66
- Subclasses, 11
- Substates, 272
- Subsystems, 21
- Superstate, 272
- Support classes, 125
- Swimlanes, 263
- Synchronization, 262
- System, 21
- System testing, 404
- Task effectiveness, 326
- Technical complexity factor, 128
- Temporal efficiency, 326
- Testability, 289
- Test case, 238
 - case matrix, 241
 - coverage, 238
 - design and procedure, 238
 - focus metric, 327
 - result, 238
 - suite, 238
- Testing coverage metrics, 327
- Thread-based testing, 404
- Throwaway prototype, 36
- Tight class cohesion, 332
- Traceability relationship, 113
- Transaction, 127
- Transition, 54, 262
- Unadjusted object points, 144
- Unadjusted use case points, 127, 128
- Unadjusted use case weight, 128
- Unified modelling language, 20
- Unit testing, 35, 403
- Univariate outliers, 313, 317
- Usability, 288
 - metrics, 326
- Use case(s), 21, 73, 76
 - based testing, 404
 - diagram(s), 73, 76
 - points, 130
 - point method, 126
- Use case scenario, 21, 73, 80
 - matrix, 240
- Users, 21, 64
- User stories, 41
- Validation, 23
- Variables, 319
- Variance, 311
- Verification, 23
- Versions, 427
- View a student, 106
- Vision document, 52
- Visual modelling, 47
- Vocabulary of a program, 321
- Volume of the program, 321
- Walkthrough, 350
- Waterfall model, 34, 55
 - advantages, 35
 - disadvantage, 35
- White box testing, 385
- Whole-part, 181
- Workflows, 51
- Worst case testing, 361
- Z-score value, 317