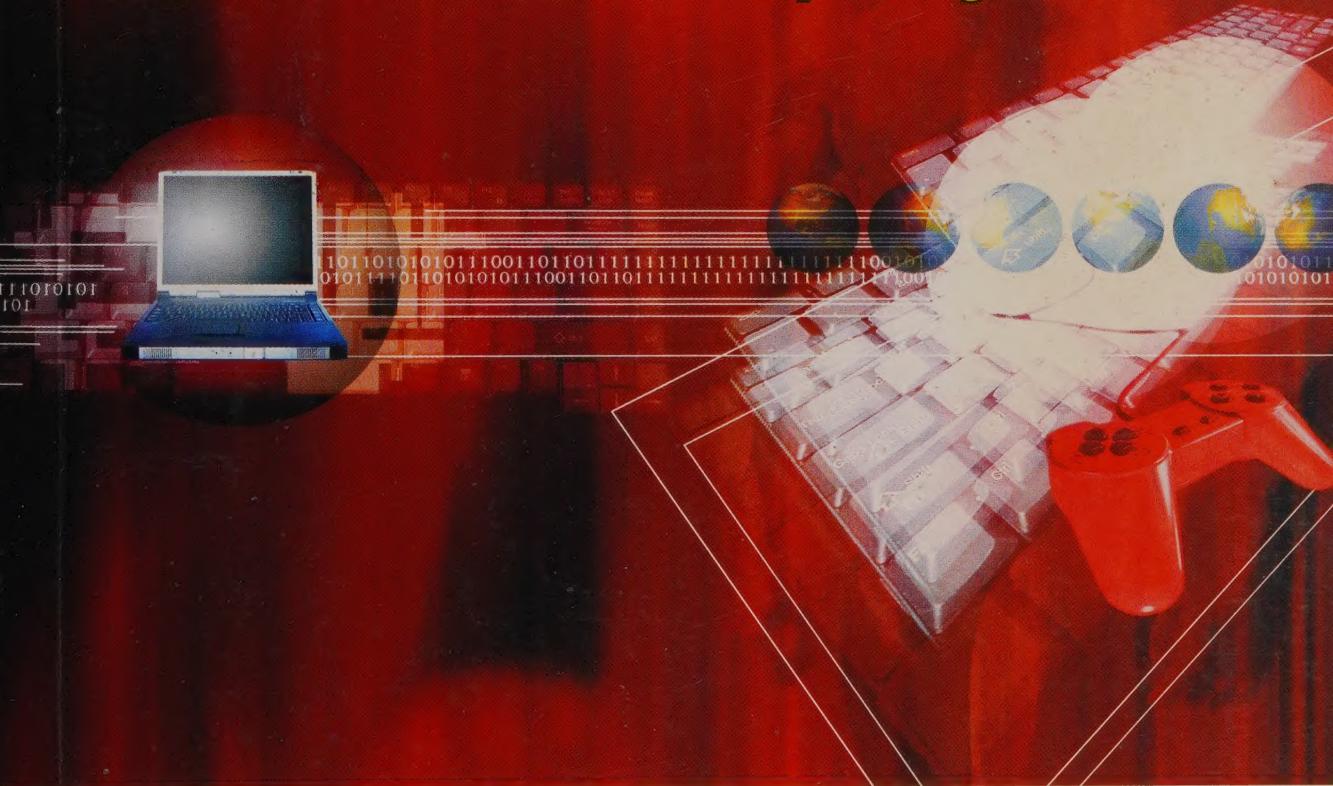


Revised Second Edition

SOFTWARE ENGINEERING



Programs • Documentation • Operating Procedures



K. K. Aggarwal & Yogesh Singh



NEW AGE INTERNATIONAL PUBLISHERS

SOFTWARE ENGINEERING

Programs • Documentation • Operating Procedures

Revised Second Edition

K. K. Aggarwal

Professor and Vice-Chancellor
Guru Gobind Singh Indraprastha University, Delhi

Yogesh Singh

Professor and Dean
University School of Information Technology
Guru Gobind Singh Indraprastha University, Delhi



NEW AGE

NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS

New Delhi • Bangalore • Chennai • Cochin • Guwahati • Hyderabad
Jalandhar • Kolkata • Lucknow • Mumbai • Ranchi

Copyright © 2005 New Age International (P) Ltd., Publishers

First Edition 2001

Revised Second Edition : 2005

Reprint : 2005

NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS

4835/24, Ansari Road, Daryaganj,

New Delhi - 110 002

Visit us at : www.newagepublishers.com

Offices at :

Bangalore, Chennai, Cochin, Guwahati, Hyderabad, Jalandhar,
Kolkata, Lucknow, Mumbai and Ranchi

This book or any part thereof may not be reproduced in any form
without the written permission of the publisher.

This book cannot be sold outside the country to which it is consigned
by the publisher without the prior permission of the publisher.

Rs. 215.00

ISBN : 81-224-1638-1

2 3 4 5 6 7 8 9 10

C-05-05-094

Published by New Age International (P) Ltd.,
4835/24, Ansari Road, Daryaganj, New Delhi-110 002 and

printed in India at Pack Printers, Delhi-110 064

typesetter Goswami Printers, Delhi

Preface to the Second Edition

As promised in the first edition, we have come up with the second edition of the text within four years. This edition is primarily based on suggestions from our readers and considered advice of the reviewers. The fact also remains that software engineering discipline is also maturing day by day. The increasing dependence of society on the software forces all of us to work hard to make software engineering as a stable discipline, where we can estimate development time and cost with reasonable accuracy and precision. Time is compelling us to improve software development processes in order to provide good quality maintainable software within reasonable cost and development time. As we know, quality is easy to feel but is difficult to define and impossible to measure. Hence, a quality software product means expecting a lot from the software engineering discipline, where every process is dominated by human beings.

These challenges are not only driving the industry but also making the universities to upgrade their curricula in the areas of Computer Science & Engineering, Computer Applications, Information Technology, Electronics & Communication Engineering and Electrical Engineering. The Second Edition is an attempt to bridge the gap between "What is taught in the classroom" and "What is practiced" in the industry. The concepts are discussed with the help of real life examples and numerical problems.

This book is designed as a textbook for the first course in Software Engineering for undergraduate and postgraduate students. This may also be helpful for software professionals to help them practice the software engineering concepts. We are indebted to Dr. Jitender Chhabra, Lecturer, National Institute of Technology, Kurukhshetra, Dr. Pravin Chandra, Lecturer, Guru Gobind Singh Indraprastha University, Delhi, Sh. R. K. Singh, Programme Co-ordinator, C-DAC, Noida and Ms. Arvinder Kaur, Lecturer, Guru Gobind Singh Indraprastha University for their valuable suggestions. We are extremely thankful to our students and other readers of first edition, from whom we have received more than we have given. We are also thankful to researchers and practitioners of the field whose ideas and techniques find a place in this book.

We do understand that there is nothing like a perfect product and same is true about this book. Hence we would welcome further suggestions from our readers. These suggestions shall motivate us to work on third edition of the book. Till then, good bye!

Prof. K.K. Aggarwal

Prof. Yogesh Singh

Preface to the First Edition

Developing software system is generally a quite complex and time consuming process. Moreover, the nature and complexity of software requirements have drastically changed in the last few decades and users all over the world have become much more demanding in terms of cost, schedule and quality. These three parameters, all being desirable, have an apparent contradiction at times which can only be resolved by optimum design of software using well established software engineering methodologies.

Software engineering methodologies constitute the framework that guides us in optimally developing the software systems. These frameworks define the different phases of software development, such as planning, requirements analysis, design testing and maintenance. The choice of which methodology to use in a specific development process is closely related to the size, complexity, reliability and maintainability of the software, and to the environment it is supposed to function in.

Given unlimited resources, the majority of software problems can probably be solved but the challenge confronting software developers is to produce high quality maintainable software with a finite amount of resources and to a specified schedule. This challenge forces us to adopt software engineering concepts, methodologies and practices in order to improve the software development process and thereby the product.

Keeping in view the software development potential the world over, software engineering is becoming an integral part of most of the universities' curricula in the discipline of Computer Science & Engineering, Computer Applications, Information Technology, Electronics & Communication Engineering and Electrical Engineering. By virtue of our experience with the industry, we realized that there is a wide gap between what is taught in the classroom and what is practised in the industry. In addition to theoretical concepts, sincere effort has therefore been made to bridge this gap between theory and practice in this text.

The book is intended to serve the requirements to a First Course on Software Engineering, whether at undergraduate or post-graduate level. While the students will find the text extremely useful, it will also serve the interests of the software professionals by making available to them the requisite material in one volume and help them to adapt their software development methodologies in the most suitable manner.

We are indebted to Dinesh Kumar, Pravin Chandra and Jitender Chhabra for their painstaking reading of the text. They found time in their busy schedule to not only read the text but also offer valuable suggestions for improvement. We wish to put on record the excellent services of Deepak Kumar and Sanjay Kumar for typing the draft of the manuscript. We owe a

sincere debt of gratitude to our students, from whom we have received more than we have given and to the many researchers and practitioners of software engineering whose ideas and techniques find a place in this book. We thank everyone at New Age International, especially Shri Anand Aswal, for their efforts in making this book a reality and that too in the most presentable form in a record time.

Being software developers, we do realise that there is nothing like a perfect product and hence we would welcome constructive comments and suggestions from our readers in order to further improve the next version (edition). Any feedback in this direction would be gratefully received and acknowledged.

Prof. K.K. Aggarwal

Prof. Yogesh Singh

Contents

Preface to the Second Edition	(v)
Preface to the First Edition	(vii)

1. INTRODUCTION	1-18
1.1 Software Crisis	2
1.1.1 No Silver Bullet	3
1.1.2 Software Myths	4
1.2 What is Software Engineering?	5
1.2.1 Definition	5
1.2.2 Program Versus Software	5
1.2.3 Software Process	7
1.2.4 Software Characteristics	9
1.2.5 Software Applications	11
1.3 Some Terminologies	12
1.3.1 Deliverables and Milestones	12
1.3.2 Product and Process	12
1.3.3 Measures, Metrics and Measurement	12
1.3.4 Software Process and Product Metrics	13
1.3.5 Productivity and Effort	13
1.3.6 Module and Software Components	13
1.3.7 Generic and Customised Software Products	13
1.4 Role of Management in Software Development	14
1.4.1 The People	14
1.4.2 The Product	15
1.4.3 The Process	15
1.4.4 The Project	15
References	15
Multiple Choice Questions	16
Exercises	18
2. SOFTWARE LIFE CYCLE MODELS	19-33
2.1 SDLC Models	20
2.1.1 Build and Fix Model	20

2.1.2	<i>The Waterfall Model</i>	21
2.1.3	<i>Prototyping Model</i>	23
2.1.4	<i>Iterative Enhancement Model</i>	24
2.1.5	<i>Evolutionary Development Model</i>	25
2.1.6	<i>Spiral Model</i>	26
2.1.7	<i>The Rapid Application Development (RAD) Model</i>	27
2.2	Selection of a Life Cycle Model	28
2.2.1	<i>Characteristics of Requirements</i>	28
2.2.2	<i>Status of Development Team</i>	29
2.2.3	<i>Involvement of Users</i>	29
2.2.4	<i>Type of Project and Associated Risk</i>	30
	<i>References</i>	31
	<i>Multiple Choice Questions</i>	31
	<i>Exercises</i>	32

3. SOFTWARE REQUIREMENTS ANALYSIS AND SPECIFICATIONS

35-126

3.1	Requirements Engineering	36
3.1.1	<i>Crucial Process Steps</i>	36
3.1.2	<i>Present State of Practice</i>	38
3.1.3	<i>Type of Requirements</i>	39
3.2	Requirements Elicitation	41
3.2.1	<i>Interviews</i>	42
3.2.2	<i>Brainstorming Sessions</i>	44
3.2.3	<i>Facilitated Application Specification Technique</i>	44
3.2.4	<i>Quality Function Deployment</i>	46
3.2.5	<i>The Use Case Approach</i>	47
3.3	Requirements Analysis	51
3.3.1	<i>Data Flow Diagrams</i>	53
3.3.2	<i>Data Dictionaries</i>	55
3.3.3	<i>Entity-Relationship Diagrams</i>	56
3.3.4	<i>Software Prototyping</i>	62
3.4	Requirements Documentation	65
3.4.1	<i>Nature of the SRS</i>	65
3.4.2	<i>Characteristics of a Good SRS</i>	66
3.4.3	<i>Organization of the SRS</i>	68
3.5	Student Result Management System—Example	84
3.5.1	<i>Problem Statement</i>	85
3.5.2	<i>Context Diagram</i>	86
3.5.3	<i>Level-n DFD</i>	87
3.5.4	<i>Entity Relationship Diagram</i>	90

3.5.5 Use Case Diagram	91
3.5.6 Use Cases	91
3.5.7 SRS Document	105
References	121
Multiple Choice Questions	123
Exercises	124
4. SOFTWARE PROJECT PLANNING	127-191
4.1 Size Estimation	129
4.1.1 Lines of Code (LOC)	130
4.1.2 Function Count	131
4.2 Cost Estimation	139
4.3 Models	139
4.3.1 Static, Single Variable Models	139
4.3.2 Static, Multivariable Models	140
4.4 The Constructive Cost Model (COCOMO)	141
4.4.1 Basic Model	141
4.4.2 Intermediate Model	144
4.4.3 Detailed COCOMO Model	146
4.5 COCOMO II	150
4.5.1 Application Composition Estimation Model	152
4.5.2 The Early Design Model	156
4.5.3 Post Architecture Model	162
4.6 The Putnam Resource Allocation Model	170
4.6.1 The Norden / Rayleigh Curve	171
4.6.2 Difficulty Metric	173
4.6.3 Productivity Versus Difficulty	175
4.6.4 The Trade-off between Time Versus Cost	176
4.6.5 Development Sub-cycle	177
4.7 Software Risk Management	183
4.7.1 What is Risk?	183
4.7.2 Typical Software Risks	184
4.7.3 Risk Management Activities	185
References	187
Multiple Choice Questions	187
Exercises	189

5. SOFTWARE DESIGN	193-240
5.1 What is Design ?	194
5.1.1 Conceptual and Technical Designs	195
5.1.2 Objectives of Design	196
5.1.3 Why Design is Important?	197

5.2	Modularity	197
5.2.1	<i>Module Coupling</i>	198
5.2.2	<i>Module Cohesion</i>	202
5.2.3	<i>Relationship between Cohesion & Coupling</i>	204
5.3	Strategy of Design	205
5.3.1	<i>Bottom-Up Design</i>	205
5.3.2	<i>Top-Down Design</i>	206
5.3.3	<i>Hybrid Design</i>	206
5.4	Function Oriented Design	207
5.4.1	<i>Design Notations</i>	208
5.4.2	<i>Functional Procedure Layers</i>	211
5.5	IEEE Recommended Practice for Software Design Descriptions	212
5.5.1	<i>Scope</i>	212
5.5.2	<i>References</i>	212
5.5.3	<i>Definitions</i>	212
5.5.4	<i>Purpose of an SDD</i>	212
5.5.5	<i>Design Description Information Content</i>	213
5.5.6	<i>Design Description Organisation</i>	214
5.6	Object Oriented Design	216
5.6.1	<i>Basic Concepts</i>	216
5.6.2	<i>Steps to Analyze and Design Object Oriented System</i>	222
5.6.3	<i>Case Study of Library Management System</i>	225
	<i>References</i>	238
	<i>Multiple Choice Questions</i>	239
	<i>Exercises</i>	240

6. SOFTWARE METRICS 241-286

6.1	Software Metrics: What & Why ?	242
6.1.1	<i>Definition</i>	243
6.1.2	<i>Areas of Applications</i>	243
6.1.3	<i>Problems During Implementation</i>	244
6.1.4	<i>Categories of Metrics</i>	245
6.2	Token Count	246
6.2.1	<i>Estimated Program Length</i>	247
6.2.2	<i>Potential Volume</i>	249
6.2.3	<i>Estimated Program Level/Difficulty</i>	250
6.2.4	<i>Effort & Time</i>	250
6.2.5	<i>Language Level</i>	250
6.3	Data Structure Metrics	256
6.3.1	<i>The Amount of Data</i>	257
6.3.2	<i>The Usage of Data within a Module</i>	259

6.3.3	<i>Program Weakness</i>	264
6.3.4	<i>The Sharing of Data Among Modules</i>	272
6.4	Information Flow Metrics	274
6.4.1	<i>The Basic Information Flow Model</i>	274
6.4.2	<i>A More Sophisticated Information Flow Model</i>	277
6.5	Metrics Analysis	278
6.5.1	<i>Using Statistics for Assessment</i>	278
6.5.2	<i>Problems with Metrics Data</i>	279
6.5.3	<i>The Common Pool of Data</i>	281
6.5.4	<i>A Pattern for Successful Applications</i>	281
	<i>References</i>	282
	<i>Multiple Choice Questions</i>	284
	<i>Exercises</i>	285

7. SOFTWARE RELIABILITY 287-344

7.1	Basic Concepts	288
7.1.1	<i>What is Software Reliability?</i>	289
7.1.2	<i>Software Reliability and Hardware Reliability</i>	290
7.1.3	<i>Failures and Faults</i>	290
7.1.4	<i>Environment</i>	294
7.1.5	<i>Uses of Reliability Studies</i>	296
7.2	Software Quality	297
7.2.1	<i>McCall Software Quality Model</i>	300
7.2.2	<i>Boehm Software Quality Model</i>	305
7.2.3	<i>ISO 9126</i>	306
7.3	Software Reliability Models	308
7.3.1	<i>Basic Execution Time Model</i>	309
7.3.2	<i>Logarithmic Poisson Execution Time Model</i>	314
7.3.3	<i>Calendar Time Component</i>	318
7.3.4	<i>The Jelinski-Moranda Model</i>	323
7.3.5	<i>The Bug Seeding Model</i>	325
7.4	Capability Maturity Model	327
7.4.1	<i>Maturity Levels</i>	327
7.4.2	<i>Key Process Areas</i>	330
7.4.3	<i>Common Features</i>	331
7.5	ISO 9000	332
7.5.1	<i>Mapping ISO 9001 to the CMM</i>	333
7.5.2	<i>Contrasting ISO 9001 and the CMM</i>	336
7.5.3	<i>Conclusion</i>	337
	<i>References</i>	337
	<i>Multiple Choice Questions</i>	339
	<i>Exercises</i>	343

8. SOFTWARE TESTING	345-437
8.1 Testing Process	346
8.1.1 <i>What is Testing?</i>	346
8.1.2 <i>Why should We Test?</i>	347
8.1.3 <i>Who should Do the Testing?</i>	348
8.1.4 <i>What should We Test?</i>	348
8.2 Some Terminologies	349
8.2.1 <i>Error, Mistake, Bug, Fault and Failure</i>	349
8.2.2 <i>Test, Test Case and Test Suite</i>	350
8.2.3 <i>Verification and Validation</i>	350
8.2.4 <i>Alpha, Beta and Acceptance Testing</i>	351
8.3 Functional Testing	352
8.3.1 <i>Boundary Value Analysis</i>	352
8.3.2 <i>Equivalence Class Testing</i>	370
8.3.3 <i>Decision Table Based Testing</i>	375
8.3.4 <i>Cause Effect Graphing Technique</i>	382
8.3.5 <i>Special Value Testing</i>	386
8.4 Structural Testing	386
8.4.1 <i>Path Testing</i>	387
8.4.2 <i>Cyclomatic Complexity</i>	402
8.4.3 <i>Graph Matrices</i>	406
8.4.4 <i>Data Flow Testing</i>	410
8.4.5 <i>Mutation Testing</i>	414
8.5 Levels of Testing	415
8.5.1 <i>Unit Testing</i>	416
8.5.2 <i>Integration Testing</i>	417
8.5.3 <i>System Testing</i>	419
8.6 Debugging	420
8.6.1 <i>Debugging Techniques</i>	421
8.6.2 <i>Debugging Approaches</i>	421
8.6.3 <i>Debugging Tools</i>	425
8.7 Testing Tools	425
8.7.1 <i>Static Testing Tools</i>	426
8.7.2 <i>Dynamic Testing Tools</i>	427
8.7.3 <i>Characteristics of Modern Tools</i>	429
References	429
Multiple Choice Questions	430
Exercises	434

9. SOFTWARE MAINTENANCE	439-472
9.1 What is Software Maintenance ?	440
9.1.1 <i>Categories of Maintenance</i>	440

9.1.2	<i>Problems During Maintenance</i>	442
9.1.3	<i>Maintenance is Manageable</i>	443
9.1.4	<i>Potential Solutions to Maintenance Problems</i>	444
9.2	The Maintenance Process	445
9.2.1	<i>Program Understanding</i>	445
9.2.2	<i>Generating Particular Maintenance Proposal</i>	446
9.2.3	<i>Ripple Effect</i>	446
9.2.4	<i>Modified Program Testing</i>	446
9.2.5	<i>Maintainability</i>	446
9.3	Maintenance Models	447
9.3.1	<i>Quick-fix Model</i>	447
9.3.2	<i>Iterative Enhancement Model</i>	447
9.3.3	<i>Reuse Oriented Model</i>	448
9.3.4	<i>Boehm's Model</i>	449
9.3.5	<i>Taute Maintenance Model</i>	450
9.4	Estimation of Maintenance Costs	451
9.4.1	<i>Belady and Lehman Model</i>	451
9.4.2	<i>Boehm Model</i>	452
9.5	Regression Testing	454
9.5.1	<i>Development Testing Versus Regression Testing</i>	454
9.5.2	<i>Regression Test Selection</i>	455
9.5.3	<i>Selective Retest Techniques</i>	456
9.6	Reverse Engineering	457
9.6.1	<i>Scope and Tasks</i>	457
9.6.2	<i>Levels of Reverse Engineering</i>	458
9.6.3	<i>Reverse Engineering Tools</i>	460
9.7	Software Re-engineering	461
9.7.1	<i>Source Code Translation</i>	463
9.7.2	<i>Program Restructuring</i>	463
9.8	Configuration Management	464
9.8.1	<i>Configuration Management Activities</i>	464
9.8.2	<i>Software Versions</i>	465
9.8.3	<i>Change Control Process</i>	465
9.9	Documentation	466
9.9.1	<i>User Documentation</i>	466
9.9.2	<i>System Documentation</i>	467
9.9.3	<i>Other Classification Schemes</i>	467
	<i>References</i>	468
	<i>Multiple Choice Questions</i>	469
	<i>Exercises</i>	471

1

Introduction

Contents

1.1 Software Crisis

- 1.1.1 No Silver Bullet
- 1.1.2 Software Myths

1.2 What is Software Engineering?

- 1.2.1 Definition
- 1.2.2 Program Versus Software
- 1.2.3 Software Process
- 1.2.4 Software Characteristics
- 1.2.5 Software Applications

1.3 Some Terminologies

- 1.3.1 Deliverables and Milestones
- 1.3.2 Product and Process
- 1.3.3 Measures, Metrics and Measurement
- 1.3.4 Software Process and Product Metrics
- 1.3.5 Productivity and Effort
- 1.3.6 Module and Software Components
- 1.3.7 Generic and Customised Software Products

1.4 Role of Management in Software Development

- 1.4.1 The People
- 1.4.2 The Product
- 1.4.3 The Process
- 1.4.4 The Project

1

Introduction

The nature and complexity of software have changed significantly in the last 30 years. In the 1970s, applications ran on a single processor, produced alphanumeric output, and received their input from a linear source. Today's applications are far more complex; typically have graphical user interface and client-server architecture. They frequently run on two or more processors, under different operating systems, and on geographically distributed machines.

Rarely, in history has a field of endeavor evolved as rapidly as software development. The struggle to stay, abreast of new technology, deal with accumulated development backlogs, and cope with people issues has become a treadmill race, as software groups work as hard as they can, just to stay in place. The initial concept of one "guru", indispensable to a project and hostage to its continued maintenance has changed. The Software Engineering Institute (SEI) and group of "gurus" advise us to improve our development process. Improvement means "ready to change". Not every member of an organization feels the need to change. It is too easy to dismiss process improvement efforts as just the latest management fad. Therein lie the seeds of conflict, as some members of a team embrace new ways of working, while others mutter "over my dead body" [WIEG94].

Therefore, there is an urgent need to adopt software engineering concepts, strategies, practices to avoid conflict, and to improve the software development process in order to deliver good quality maintainable software in time and within budget.

1.1 SOFTWARE CRISIS

The software crisis has been with us since 1970. Since then, the computer industry has progressed at a break-neck speed through the computer revolution, and recently, the network revolution triggered and/or accelerated by the explosive spread of the internet and most recently the web. Computer industry has been delivering exponential improvement in price-performance, but the problems with software have not been decreasing. Software still come late, exceed budget and are full of residual faults. As per the latest IBM report, "31% of the projects get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts" [IBMG2K]. History has seen many software failures. Some of these are :

(i) The Y2K problem was the most crucial problem of last century. It was simply the ignorance about the adequacy or otherwise of using only last two digits of the year. The 4-digit date format, like 1964, was shortened to 2-digit format, like 64. The developers could not

visualise the problem of year 2000. Millions of rupees have been spent to handle this practically non-existent problem.

(ii) The “star wars” program of USA produced “Patriot missile” and was used first time in Gulf war. Patriot missiles were used as a defence for Iraqi Scud missiles. The Patriot missiles failed several times to hit Scud missiles, including one that killed 28 U.S. soldiers in Dhahran, Saudi Arabia. A review team was constituted to find the reason and result was software bug. A small timing error in the system’s clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

(iii) In 1996, a US consumer group embarked on an 18-month, \$1 million project to replace its customer database. The new system was delivered on time but did not work as promised, handling routine transactions smoothly but tripping over more complex ones. Within three weeks the database was shutdown, transactions were processed by hand and a new team was brought in to rebuild the system. Possible reasons for such a failure may be that the design team was over optimistic in agreeing to requirements and developers became fixated on deadlines, allowing errors to be ignored.

(iv) “One little bug, one big crash” of Ariane-5 space rocket, developed at a cost of \$7000 M over a 10 year period. The space rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles alongwith its payload of four expensive and uninsured scientific satellites. The reason was very simple. When the guidance system’s own computer tried to convert one piece of data—the sideways velocity of the rocket—from a 64-bit format to a 16-bit format; the number was too big, and an overflow error resulted after 36.7 seconds. When the guidance system shutdown, it passed control to an identical, redundant unit, which was there to provide backup in case of just such a failure. Unfortunately, the second unit had failed in the identical manner a few milliseconds before. In this case, the developers had decided that this particular velocity figure would never be large enough to cause trouble—after all, it never had been before.

We may discuss many such failures which have played with human safety and caused the project to fail in past. Hence, in order to handle such unfortunate events, a systematic and scientific discipline is required and this emerging discipline is software engineering.

1.1.1 No Silver Bullet

As we all know, the hardware cost continues to decline drastically. However, there are desperate cries for a silver bullet-something to make software costs drop as rapidly as computer hardware costs do. But as we look to the horizon of a decade, we see no silver bullet. There is no single development, either in technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.

Inventions in electronic design through transistors and large scale integration has significantly affected the cost, performance and reliability of the computer hardware. No other technology, since civilization began, has seen six orders of magnitude in performance-price gain in 30 years. The progress in software technology is not that rosy due to certain difficulties with this technology. Some of the difficulties are complexity, changeability and invisibility.

The hard part of building software is the specification, design and testing of this conceptual construct, not the labour of representing it and testing the correctness of

representation. We still make syntax errors, to be sure, but they are trivial as compared to the conceptual errors (logic errors) in most systems. That is why, building software is always hard and there is inherently no silver bullet.

Many people (especially CASE tool vendors) believe that CASE (Computer Aided Software Engineering) tools represent the so-called silver bullet that would rescue the software industry from the software crisis. Many companies have used these tools and spent large sums of money, but results were highly unsatisfactory, we learnt the hard way that there is no such thing as a silver bullet [BROO87].

1.1.2 Software Myths

There are number of myths associated with software development community. Some of them really affect the way, in which software development should take place. In this section, we list few myths, and discuss their applicability to standard software development. [PIER99, LEVE95].

1. Software is easy to change. It is true that source code files are easy to edit, but that is quite different than saying that software is easy to change. This is deceptive precisely because source code is so easy to alter. But making changes without introducing errors is extremely difficult, particularly in organizations with poor process maturity. Every change requires that the complete system be re-verified. If we do not take proper care, this will be an extremely tedious and expensive process.

2. Computers provide greater reliability than the devices they replace. It is true that software does not fail in the traditional sense. There are no limits to how many times a given piece of code can be executed before it “wears out”. In any event, the simple expression of this myth is that our general ledgers are still not perfectly accurate, even though they have been computerized. Back in the days of manual accounting systems, human error was a fact of life. Now, we have software error as well.

3. Testing software or “proving” software correct can remove all the errors. Testing can only show the presence of errors. It cannot show the absence of errors. Our aim is to design effective test cases in order to find maximum possible errors. The more we test, the more we are confident about our design.

4. Reusing software increases safety. This myth is particularly troubling because of the false sense of security that code re-use can create. Code re-use is a very powerful tool that can yield dramatic improvement in development efficiency, but it still requires analysis to determine its suitability and testing to determine if it works.

5. Software can work right the first time. If we go to an aeronautical engineer, and ask him to build a jet fighter craft, he will quote us a price. If we demand that it is to be put in production without building a prototype, he will laugh and may refuse the job. Yet, software engineers are often asked to do precisely this sort of work, and they often accept the job.

6. Software can be designed thoroughly enough to avoid most integration problems. There is an old saying among software designers: “Too bad, there is no complier for specifications”: This points out the fundamental difficulty with detailed specifications. They always have inconsistencies, and there is no computer tool to perform consistency checks on these. Therefore, special care is required to understand the specifications, and if there is an ambiguity, that should be resolved before proceeding for design.

7. **Software with more features is better software.** This is, of course, almost the opposite of the truth. The best, most enduring programs are those which do one thing well.

8. **Addition of more software engineers will make up the delay.** This is not true in most of the cases. By the process of adding more software engineers during the project, we may further delay the project. This does not serve any purpose here, although this may be true for any civil engineering work.

9. **Aim is to develop working programs.** The aim has been shifted from developing working programs to good quality, maintainable programs. Maintaining software has become a very critical and crucial area for software engineering community.

This list is endless. These myths, poor quality of software, increasing cost and delay in the delivery of the software have been the driving forces behind the emergence of software engineering as a discipline. In addition, following are the contributing factors:

- Change in ratio of hardware to software costs
- Increasing importance of maintenance
- Advances in software techniques
- Increased demand for software
- Demand for larger and more complex software systems.

1.2 WHAT IS SOFTWARE ENGINEERING?

Software has become critical to advancement in almost all areas of human endeavour. The art of programming only is no longer sufficient to construct large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products.

Software Engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the product.

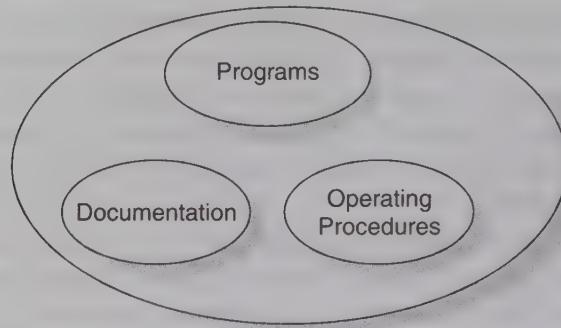
1.2.1 Definition

At the first conference on software engineering in 1968, Fritz Bauer [FRIT68] 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*". Stephen Schach [SCHA90] defined the same as "*A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements*".

Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

1.2.2 Program Versus Software

Software is more than programs. It consists of programs, documentation of any facet of the program and the procedures used to setup and operate the software system. The components of the software systems are shown in Fig. 1.1.



Software = Program + Documentation + Operating Procedures

Fig. 1.1: Components of software

Any program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared. Program is a combination of source code and object code. Documentation consists of different types of manuals as shown in Fig. 1.2.

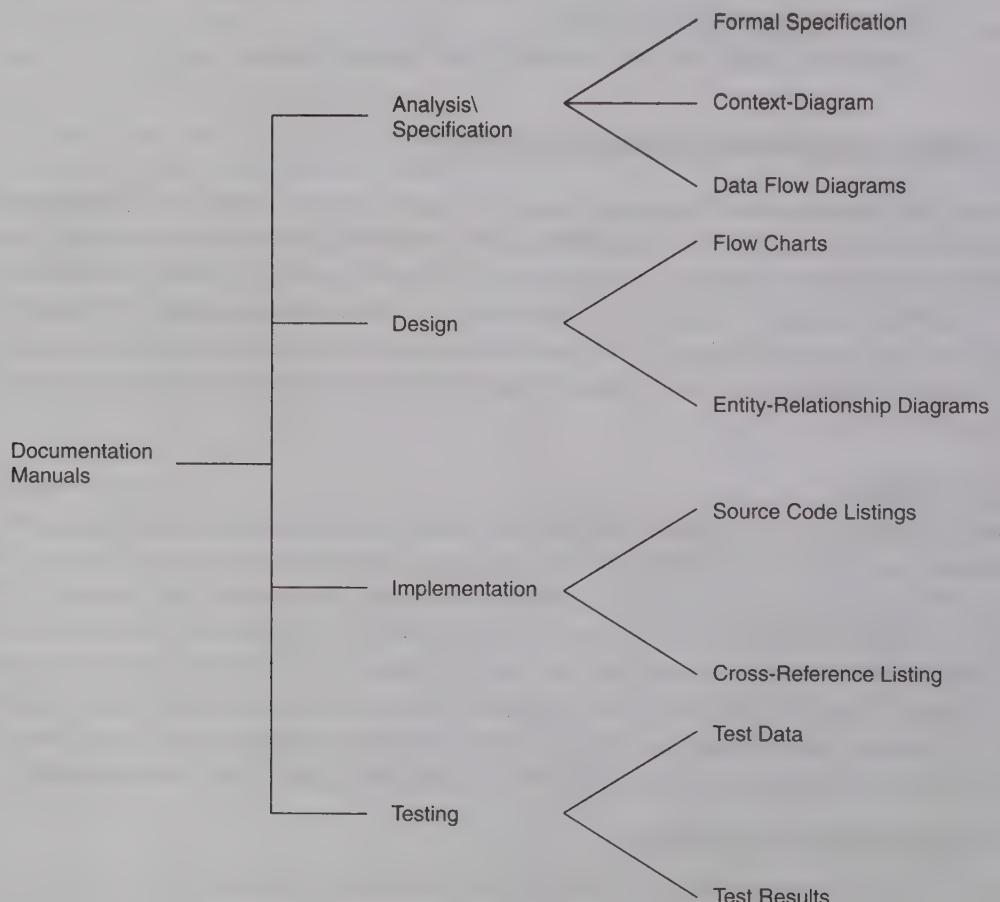


Fig. 1.2: List of documentation manuals.

Operating procedures consist of instructions to setup and use the software system and instructions on how to react to system failure. List of operating procedure manuals/documents is given in Fig. 1.3.

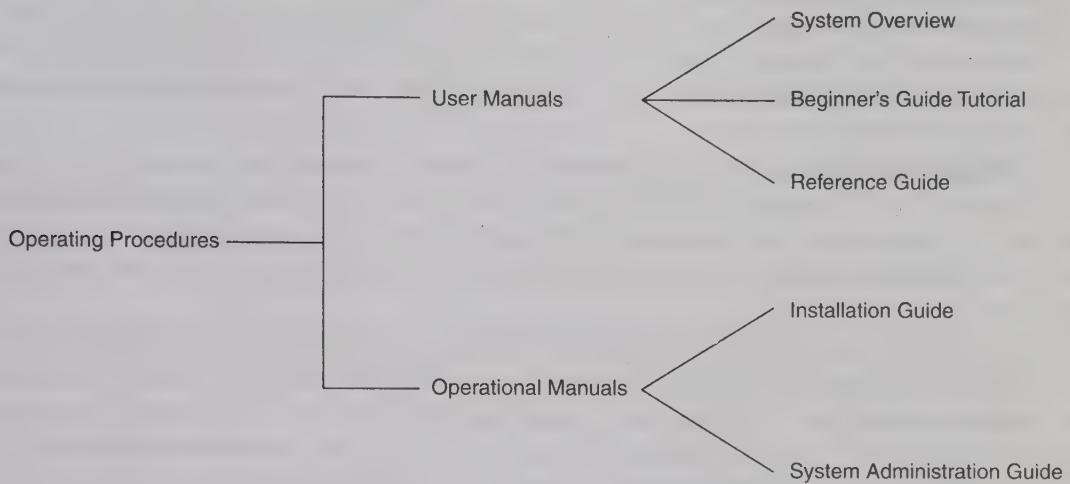


Fig. 1.3: List of operating procedure manuals.

1.2.3 Software Process

The software process is the way in which we produce software. This differs from organization to organization. Surviving in the increasingly competitive software business requires more than hiring smart, knowledgeable developers and buying the latest development tools. We also need to use effective software development processes, so that developers can systematically use the best technical and managerial practices to successfully complete their projects. Many software organizations are looking at software process improvement as a way to improve the quality, productivity, predictability of their software development, and maintenance efforts [WIEG96].

It seems straight forward, and the literature has a number of success stories of companies that substantially improved their software development and project management capabilities. However, many other organizations do not manage to achieve significant and lasting improvements in the way they conduct their projects. Here we discuss few reasons why it is difficult to improve software process [HUMP89, WIEG99] ?

1. Not enough time. Unrealistic schedules leave insufficient time to do the essential project work. No software groups are sitting around with plenty of spare time to devote to exploring what is wrong with their current development processes and what they should be doing differently. Customers and senior managers are demanding more software, of higher quality in minimum possible time. Therefore, there is always a shortage of time. One consequence is that software organizations may deliver release 1.0 on time, but then they have to ship release 1.01 almost immediately thereafter to fix the recently discovered bugs.

2. Lack of knowledge. A second obstacle to widespread process improvement is that many software developers do not seem to be familiar with industry best practices. Normally,

software developers do not spend much time reading the literature to find out about the best-known ways of software development. Developers may buy books on Java, Visual Basic or ORACLE, but do not look for anything about process, testing or quality on their bookshelves.

The industry awareness of process improvement frameworks such as the capability maturity model and ISO 9001 for software (discussed in Chapter 7) have grown in recent years, but effective and sensible application still is not that common. Many recognized best practices available in literature simply are not in widespread use in the software development world.

3. Wrong motivations. Some organizations launch process improvement initiatives for the wrong reasons. May be an external entity, such as a contractor, demanded that the development organization should achieve CMM level X by date Y. Or perhaps a senior manager learned just enough about the CMM and directed his organization to climb on the CMM bandwagon.

The basic motivation for software process improvement should be to make some of the current difficulties we experience on our projects to go away. Developers are rarely motivated by seemingly arbitrary goals of achieving a higher maturity level or an external certification (ISO 9000) just because someone has decreed it. However, most people should be motivated by the prospect of meeting their commitments, improving customer satisfaction, and delivering excellent products that meet customer expectations. The developers have resisted many process improvement initiatives when they were directed to do “the CMM thing”, without a clear explanation of the reasons why improvement was needed and the benefits the team expected to achieve.

4. Insufficient commitment. Many times, the software process improvement fails, despite best of intentions, due to lack of true commitment. It starts with a process assessment but fails to follow through with actual changes. Management sets no expectations from the development community around process improvement; they devote insufficient resources, write no improvement plan, develop no roadmap, and pilot no new processes.

The investment we make in process improvement will not have an impact on current productivity; because the time we spend developing better ways to work tomorrow is not available for today's assignment. It can be tempting to abandon the effort when skeptics see the energy they want to be devoted to immediate demands being siphoned off in the hope of a better future (refer Fig. 1.4). Software organizations should not give up, but should take

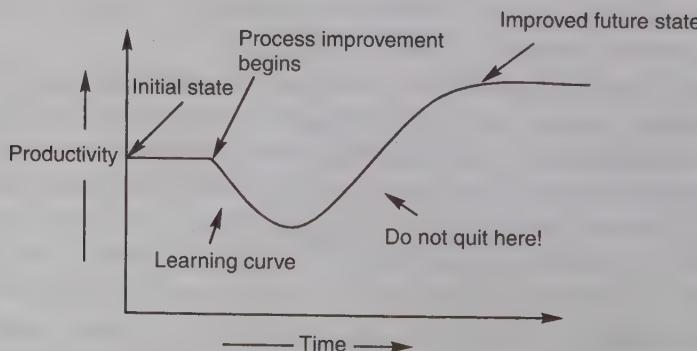


Fig. 1.4: The process improvement learning curve.

motivation from the very real, long-term benefits that many companies (including Motorola, Hewlett-Packard, Boeing, Microsoft etc.) have enjoyed from sustained software process improvement initiatives. Improvements will take place over time and organizations should not expect and promise miracles [WIEG2K] and should always remember the learning curve.

1.2.4 Software Characteristics

The software has a very special characteristic e.g., "it does not wear out". Its behaviour and nature is quite different than other products of human life. A comparison with one such case, i.e., constructing a bridge vis-a-vis writing a program is given in Table 1.1. Both activities require different processes and have different characteristics.

Table 1.1: A comparison of constructing a bridge and writing a program

Sr. No.	Constructing a bridge	Writing a program
1.	The problem is well understood.	Only some parts of the problem are understood, others are not.
2.	There are many existing bridges.	Every program is different and designed for special applications.
3.	The requirements for a bridge typically do not change much during construction.	Requirements typically change during all phases of development.
4.	The strength and stability of a bridge can be calculated with reasonable precision.	Not possible to calculate correctness of a program with existing methods.
5.	When a bridge collapses, there is a detailed investigation and report.	When a program fails, the reasons are often unavailable or even deliberately concealed.
6.	Engineers have been constructing bridges for thousands of years.	Developers have been writing programs for 50 years or so.
7.	Materials (wood, stone, iron, steel) and techniques (making joints in wood, carving stone, casting iron) change slowly.	Hardware and software changes rapidly.

Some of the important characteristics are discussed below:

(i) **Software does not wear out.**

There is a well-known "bath tub curve" in reliability studies for hardware products. The curve is given in Fig. 1.5. The shape of the curve is like "bath tub"; and is known as bath tub curve.

There are three phases for the life of a hardware product. Initial phase is burn-in phase, where failure intensity is high. It is expected to test the product in the industry before delivery. Due to testing and fixing

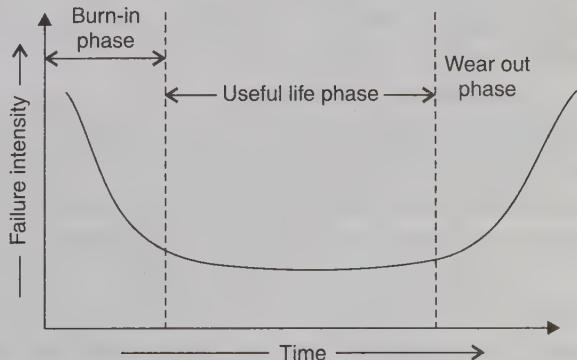


Fig. 1.5: Bath tub curve.

faults, failure intensity will come down initially and may stabilise after certain time. The second phase is the useful life phase where failure intensity is approximately constant and is called useful life of a product. After few years, again failure intensity will increase due to wearing out of components. This phase is called wear out phase. We do not have this phase for the software as it does not wear out. The curve for software is given in Fig. 1.6.

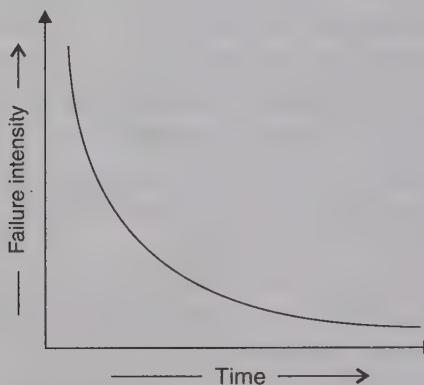


Fig. 1.6: Software curve.

Important point is software becomes reliable overtime instead of wearing out. It becomes obsolete, if the environment for which it was developed, changes. Hence software may be retired due to environmental changes, new requirements, new expectations, etc.

(ii) **Software is not manufactured.** The life of a software is from concept exploration to the retirement of the software product. It is one time development effort and continuous maintenance effort in order to keep it operational. However, making 1000 copies is not an issue and it does not involve any cost. In case of hardware product, every product costs us due to raw material and other processing expenses. We do not have assembly line in software development. Hence it is not manufactured in the classical sense.

(iii) **Reusability of components.** If we have to manufacture a TV, we may purchase picture tube from one vendor, cabinet from another, design card from third and other electronic components from fourth vendor. We will assemble every part and test the product thoroughly to produce a good quality TV. We may be required to manufacture only a few components or no component at all. We purchase every unit and component from the market and produce the finished product. We may have standard quality guidelines and effective processes to produce a good quality product.

In software, every project is a new project. We start from the scratch and design every unit of the software product. Huge effort is required to develop a software which further increases the cost of the software product. However, effort has been made to design standard components that may be used in new projects. Software reusability has introduced another area and is known as component based software engineering.

Hence developers can concentrate on truly innovative elements of design, that is, the parts of the design that represent something new. As explained earlier, in the hardware world, component reuse is a natural part of the engineering process. In software, there is only a humble beginning like graphical user interfaces are built using reusable components that

enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.

(iv) **Software is flexible.** We all feel that software is flexible. A program can be developed to do almost anything. Sometimes, this characteristic may be the best and may help us to accommodate any kind of change. However, most of the times, this “almost anything” characteristic has made software development difficult to plan, monitor and control. This unpredictability is the basis of what has been referred to for the past 35 years as the “Software Crisis”.

1.2.5 Software Applications

Software has become integral part of most of the fields of human life. We name a field and we find the usage of software in that field. Software applications are grouped in to eight areas for convenience as shown in Fig. 1.7.

(i) **System Software.** Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a collection of programs to provide service to other programs.

(ii) **Real Time Software.** These software are used to monitor, control and analyze real world events as they occur. An example may be software required for weather forecasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forecast the weather.

(iii) **Embedded Software.** This type of software is placed in “Read-Only-Memory (ROM)” of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software.

(iv) **Business Software.** This is the largest application area. The software designed to process business applications is called business software. Business software could be payroll, file monitoring system, employee management, account management. It may also be a data warehousing tool which helps us to take decisions based on available data. Management information system, enterprise resource planning (ERP) and such other software are popular examples of business software.

(v) **Personal Computer Software.** The software used in personal computers are covered in this category. Examples are word processors, computer graphics, multimedia and animating tools, database management, computer games etc. This is a very upcoming area and many big organisations are concentrating their effort here due to large customer base.

(vi) **Artificial Intelligence Software.** Artificial Intelligence Software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis [PRESOI]. Examples are expert systems, artificial neural network, signal processing software etc.

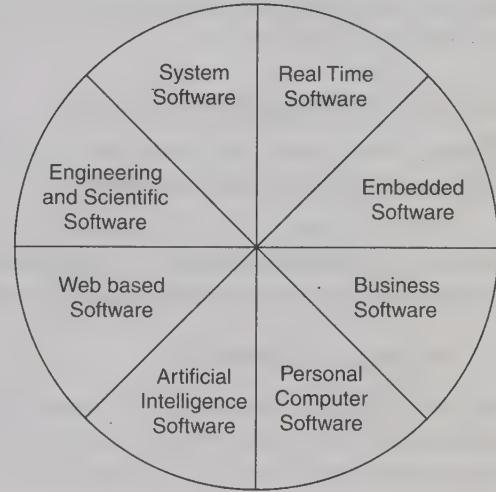


Fig. 1.7: Software applications.

(vii) **Web based Software.** The software related to web applications come under this category. Examples are CGI, HTML, Java, Perl, DHTML etc.

(viii) **Engineering and Scientific Software.** Scientific and engineering application software are grouped in this category. Huge computing is normally required to process data. Examples are CAD/CAM package, SPSS, MATLAB, Engineering Pro, Circuit analyzers etc.

1.3 SOME TERMINOLOGIES

Some terminologies are discussed in this section which are frequently used in the field of Software Engineering.

1.3.1 Deliverables and Milestones

Different deliverables are generated during software development. The examples are source code, user manuals, operating procedure manuals etc.

The milestones are the events that are used to ascertain the status of the project. Finalisation of specification is a milestone. Completion of design documentation is another milestone. The milestones are essential for project planning and management.

1.3.2 Product and Process

Product: What is delivered to the customer, is called a product. It may include source code, specification document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.

Process: Process is the way in which we produce software. It is the collection of activities that leads to (a part of) a product. An efficient process is required to produce good quality products.

If the process is weak, the end product will undoubtedly suffer, but an obsessive over-reliance on process is also dangerous.

1.3.3 Measures, Metrics and Measurement

The terms measures, metrics and measurement are often used interchangeably. It is interesting to understand the difference amongst these. A measure provides a quantitative indication of the extent, dimension, size, capacity, efficiency, productivity or reliability of some attributes of a product or process.

Measurement is the act of evaluating a measure. A metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute. Pressman [PRESO2] explained this very effectively with an example as given below:

“When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors in each module). A software metric relates the individual measures in some way (e.g., the average number of errors found per review).”

Hence we collect measures and develop metrics to improve the software engineering practices.

1.3.4 Software Process and Product Metrics

Software metrics are used to quantitatively characterise different aspects of software process or software products. Process metrics quantify the attributes of software development process and environment; whereas product metrics are measures for the software product. Examples of process metrics include productivity, quality, failure rate, efficiency etc. Examples of product metrics are size, reliability, complexity, functionality etc.

1.3.5 Productivity and Effort

Productivity is defined as the rate of output, or production per unit of effort, *i.e.*, the output achieved with regard to the time taken but irrespective of the cost incurred. Hence, there are two issues for deciding the unit of measure

- (i) quantity of output
- (ii) period of time.

In software, one of the measure for quantity of output is lines of code (LOC) produced. Time is measured in days or months.

Hence most appropriate unit of effort is Person Months (PMs), meaning thereby number of persons involved for specified months. So, productivity may be measured as LOC/PM (lines of code produced/person month).

1.3.6 Module and Software Components

There are many definitions of the term module. They range from “a module is a FORTRAN subroutine” to “a module is an Ada Package”, to “Procedures and functions of PASCAL and C”, to “C++ Java classes” to “Java packages” to “a module is a work assignment for an individual developer”. All these definitions are correct. The term subprogram is also used sometimes in place of module.

There are many definitions of software components. A general definition given by Alan W. Brown [BROW2K] is:

“An independently deliverable piece of functionality providing access to its services through interfaces”.

Another definition from unified modeling language (UML) [OMG2K] is:

“A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces”.

Hence, a reusable module is an independent and deliverable software part that encapsulates a functional specification and implementation for reuse by a third party.

However, a reusable component is an independent, deployable, and replaceable software unit that is reusable by a third party based on unit’s specification, implementation, and well defined contracted interfaces.

1.3.7 Generic and Customised Software Products

The software products are divided in two categories:

- (i) Generic products
- (ii) Customised products.

Generic products are developed for anonymous customers. The target is generally the entire world and many copies are expected to be sold. Infrastructure software like operating systems, compilers, analysers, word processors, CASE tools etc. are covered in this category.

The customised products are developed for particular customers. The specific product is designed and developed as per customer requirements. Most of the development projects (say about 80%) come under this category.

1.4 ROLE OF MANAGEMENT IN SOFTWARE DEVELOPMENT

The management of software development is heavily dependent on four factors: People, Product, Process, and Project. Order of dependency is as shown in Fig. 1.8.

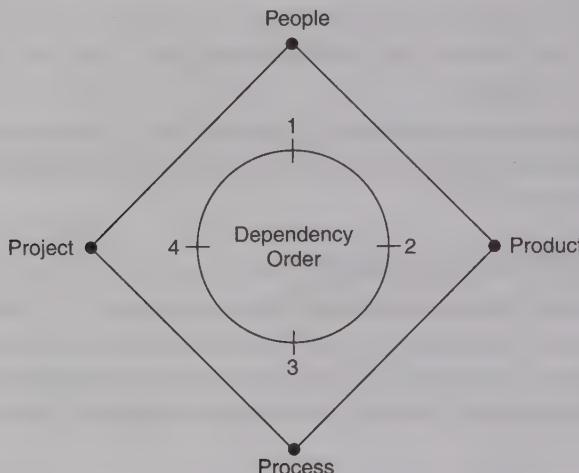


Fig. 1.8: Factors of management dependency (from People to Project).

Software development is a people centric activity. Hence, success of the project is on the shoulders of the people who are involved in the development.

1.4.1 The People

Software development requires good managers. The managers, who can understand the psychology of people and provide good leadership. A good manager can not ensure the success of the project, but can increase the probability of success. The areas to be given priority are: proper selection, training, compensation, career development, work culture etc.

Managers face challenges. It requires mental toughness to endure inner pain. We need to plan for the best, be prepared for the worst, expect surprises, but continue to move forward anyway. Charles Maurice once rightly said “I am more afraid of an army of one hundred sheep led by a lion than an army of one hundred lions led by a sheep”.

Hence, manager selection is most crucial and critical. After having a good manager, project is in safe hands. It is the responsibility of a manager to manage, motivate, encourage, guide and control the people of his/her team.

1.4.2 The Product

What do we want to deliver to the customer? Obviously, a product; a solution to his/her problems.

Hence, objectives and scope of work should be defined clearly to understand the requirements. Alternate solutions should be discussed. It may help the managers to select a “best” approach within constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces etc. Without well defined requirements, it may be impossible to define reasonable estimates of the cost, development time and schedule for the project.

1.4.3 The Process

The process is the way in which we produce software. It provides the framework from which a comprehensive plan for software development can be established. If the process is weak, the end product will undoubtedly suffer. There are many life cycle models and process improvements models. Depending on the type of project, a suitable model is to be selected. Now-a-days CMM (Capability Maturity Model) has become almost a standard for process framework. The process priority is after people and product, however, it plays very critical role for the success of the project. A small number of framework activities are applicable to all software projects, regardless of their size and complexity. A number of different task sets, tasks, milestones, work products, and quality assurance points, enable the framework activities to be adopted to the characteristics of the project and the requirements of the project team.

1.4.4 The Project

A proper planning is required to monitor the status of development and to control the complexity. Most of the projects are coming late with cost overruns of more than 100%. In order to manage a successful project, we must understand what can go wrong and how to do it right. We should define concrete requirements (although very difficult) and freeze these requirements. Changes should not be incorporated to avoid software surprises. Software surprises are always risky and we should minimise them. We should have a planning mechanism to give warning before the occurrence of any surprise.

All four factors (People, Product, Process and Project) are important for the success of the project. Their relative importance helps us to organise development activities in more scientific and professional way.

REFERENCES

- [FRIT68] Bauer, Fritz et al., “Software Engineering: A Report on a Conference Sponsored by NATO Science Committee”, NATO, 1968.
- [BOEH89] Boehm B., “Risk Management”, IEEE Computer Society Press, 1989.
- [BROO87] Brooks F.P., “No Silver Bullet : Essence and Accidents of Software Engineering”, IEEE Computer, 10—19, April, 1987.
- [BROW2K] Brown A.W., “Large Scale, Component based Development”, Englewood cliffs, NJ, PH, 2000.

- [HUMP89] Humphrey W.S., "Managing the Software Process", Addison-Wesley Pub. Co., Reading, Massachusetts, USA, 1989.
- [IBMG2K] IBM Global Services India Pvt. Ltd., Golden Tower, Airport Road, Bangalore, Letter of Bindu Subramani, Segment Manager—Corporate Training, March 9, 2000.
- [LEVE95] Leveson N.G., "Software, System Safety and Computers", Addison Wesley, 1995.
- [OMG2K] OMG Unified Modelling Language Specification, Version 1.4, Object Management Group, 2000.
- [PRES02] Pressman R., "Software Engineering", McGraw Hill, 2002.
- [PIER99] Piersal K., "Amusing Software Myths", www.bejeeber.org/Software-myths.html, 1999.
- [SCHA90] Schach, Stephen, "Software Engineering", Vanderbilt University, Aksen Association, 1990.
- [WIEG2K] Wiegers K.E., "Stop Promising Miracles" Software Development Magazine, February, 2000.
- [WIEG94] Wiegers K.E., "Creating a Software Engineering Culture", Software Development Magazine, July, 1994.
- [WIEG96] Wiegers K.E., "Software Process Improvement: Ten Traps to Avoid", Software Development Magazine, May, 1996.
- [WIEG99] Wiegers K.E., "Why is Process Improvement So Hard", Software Development Magazine, February, 1999.

MULTIPLE CHOICE QUESTIONS

Note : Select most appropriate answer of the following questions.

- 1.1. Software is

(a) superset of programs	(b) subset of programs
(c) set of programs	(d) none of the above.
- 1.2. Which is NOT the part of operating procedure manuals?

(a) User manuals	(b) Operational manuals
(c) Documentation manuals	(d) Installation manuals.
- 1.3. Which is NOT a software characteristic?

(a) Software does not wear out	(b) Software is flexible
(c) Software is not manufactured	(d) Software is always correct.
- 1.4. Product is

(a) Deliverables	(b) User expectations
(c) Organisation's effort in development	(d) none of the above.
- 1.5. To produce a good quality product, process should be

(a) Complex	(b) Efficient
(c) Rigorous	(d) None of the above.
- 1.6. Which is not a product metric?

(a) Size	(b) Reliability
(c) Productivity	(d) Functionality.
- 1.7. Which is not a process metric?

(a) Productivity	(b) Functionality
(c) Quality	(d) Efficiency.

EXERCISES

- 1.1. Why is the primary goal of software development now shifting from producing good quality software to good quality maintainable software?
- 1.2. List the reasons for the “software crisis”? Why are CASE tools not normally able to control it?
- 1.3. “The software crisis is aggravated by the progress in hardware technology?” Explain with examples.
- 1.4. What is software crisis? Was Y2K a software crisis?
- 1.5. What is the significance of software crisis in reference to software engineering discipline.
- 1.6. How are software myths affecting software process? Explain with the help of examples.
- 1.7. State the difference between program and software. Why have documents and documentation become very important?
- 1.8. What is software engineering? Is it an art, craft or a science? Discuss.
- 1.9. What is the aim of software engineering? What does the discipline of software engineering discuss?
- 1.10. Define the term “Software Engineering”. Explain the major differences between software engineering and other traditional engineering disciplines.
- 1.11. What is software process? Why is it difficult to improve it?
- 1.12. Describe the characteristics of software contrasting it with the characteristics of hardware.
- 1.13. Write down the major characteristics of a software. Illustrate with a diagram that the software does not wear out.
- 1.14. What are the components of a software? Discuss how a software differs from a program.
- 1.15. Discuss major areas of the applications of the software.
- 1.16. Is software a product or process? Justify your answer with examples.
- 1.17. Differentiate between the followings
 - (i) Deliverables and milestones
 - (ii) Product and process
 - (iii) Measures, metrics and measurement
- 1.18. What is software metric? How is it different from software measurement?
- 1.19. Discuss software process and product metrics with the help of examples.
- 1.20. What is productivity? How is it related to effort? What is the unit of effort?
- 1.21. Differentiate between module and software component.
- 1.22. Distinguish between generic and customised software products. Which one has larger share of market and why?
- 1.23. Describe the role of management in software development with the help of examples.
- 1.24. What are various factors of management dependency in software development? Discuss each factor in detail.
- 1.25. What is more important: Product or process? Justify your answer.

2

Software Life Cycle Models

Contents

2.1 SDLC Models

- 2.1.1 Build and Fix Model
- 2.1.2 The Waterfall Model
- 2.1.3 Prototyping Model
- 2.1.4 Iterative Enhancement Model
- 2.1.5 Evolutionary Development Model
- 2.1.6 Spiral Model
- 2.1.7 The Rapid Application Development (RAD) Model

2.2 Selection of a Life Cycle Model

- 2.2.1 Characteristics of Requirements
 - 2.2.2 Status of Development Team
 - 2.2.3 Involvement of Users
 - 2.2.4 Type of Project and Associated Risk
-

The goal of software engineering is to provide models and processes that lead to the production of well-documented maintainable software in a manner that is predictable. For a mature process, it should be possible to determine in advance how much time and effort will be required to produce the final product. This can only be done using data from past experience, which requires that we must measure the software process.

Software development organizations follow some process when developing a software product. In immature organizations, the process is usually not written down. In mature organizations, the process is in writing and is actively managed. A key component of any software development process is the life cycle model on which the process is based. The particular life cycle model can significantly affect overall life cycle costs associated with a software product [RAKI97]. Life cycle of the software starts from concept exploration and ends at the retirement of the software.

In the IEEE standard Glossary of software Engineering Terminology, the software life cycle is:

“The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirement phase, design phase, implementation phase, test phase, installation and check out phase, operation and maintenance phase, and sometimes retirement phase”.

A software life cycle model is a particular abstraction that represents a software life cycle. A software life cycle model is often called a software development life cycle (SDLC).

2.1 SDLC MODELS

A variety of life cycle models have been proposed and are based on tasks involved in developing and maintaining software. Few well known life cycles models are discussed in this chapter.

2.1.1 Build and Fix Model

Sometimes a product is constructed without specifications or any attempt at design. Instead, the developer simply builds a product that is reworked as many times as necessary to satisfy the client [SCHA96].

This is an adhoc approach and not well defined. Basically, it is a simple two-phase model. The first phase is to write code and the next phase is to fix it as shown in Fig. 2.1. Fixing in this context may be error correction or addition of further functionality [TAKA96].

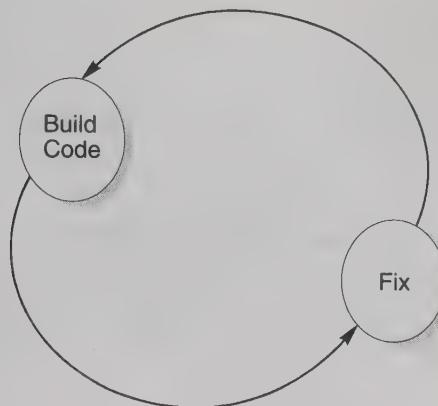


Fig. 2.1: Build and fix model.

Although this approach may work well on small programming exercises 100 or 200 lines long, this model is totally unsatisfactory for software of any reasonable size. Code soon becomes unfixable and unenhanceable. There is no room for design or any aspect of development process to be carried out in a structured or detailed way. The cost of the development using this approach is actually very high as compared to the cost of a properly specified and carefully designed product. In addition, maintenance of the product can be extremely difficult without specification or design documents.

2.1.2 The Waterfall Model

The most familiar model is the waterfall model, which is given in Fig. 2.2. This model has five phases: Requirements analysis and specification, design, implementation and unit testing, integration and system testing, and operation and maintenance. The phases always occur in this order and do not overlap. The developer must complete each phase before the next phase begins. This model is named “Waterfall Model”, because its diagrammatic representation resembles a cascade of waterfalls.

1. Requirement analysis and specification phase. The goal of this phase is to understand the exact requirements of the customer and to document them properly. This activity is usually executed together with the customer, as the goal is to document all functions, performance and interfacing requirements for the software. The requirements describe the “what” of a system, not the “how”. This phase produces a large document, written in a natural language, contains a description of what the system will do without describing how it will be done. The resultant document is known as software requirement specification (SRS) document.

The SRS document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure to implement the contracted system.

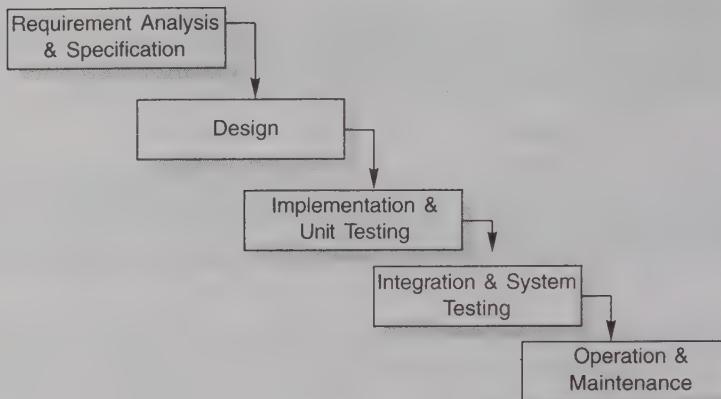


Fig. 2.2: Waterfall model

2. Design phase. The SRS document is produced in the previous phase, which contains the exact requirements of the customer. The goal of this phase is to transform the requirements specification into a structure that is suitable for implementation in some programming language. Here, overall software architecture is defined, and the high level and detailed design work is performed. This work is documented and known as software design description (SDD) document. The information contained in the SDD should be sufficient to begin the coding phase.

3. Implementation and unit testing phase. During this phase, design is implemented. If the SDD is complete, the implementation or coding phase proceeds smoothly, because all the information needed by the software developers is contained in the SDD.

During testing, the major activities are centered around the examination and modification of the code. Initially, small modules are tested in isolation from the rest of the software product. There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? Such problems are solved in this phase and modules are tested after writing some overhead code.

4. Integration and system testing phase. This is a very important phase. Effective testing will contribute to the delivery of higher quality software products, more satisfied users, lower maintenance costs, and more accurate and reliable results. It is a very expensive activity and consumes one-third to one half of the cost of a typical development project.

As we know, the purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing is performed. System testing involves the testing of the entire system, whereas software is a part of the system. This is essential to build confidence in the developers before software is delivered to the customer or released in the market.

5. Operation and maintenance phase. Software maintenance is a task that every development group has to face, when the software is delivered to the customer's site, installed and is operational. Therefore, release of software inaugurates the operation and maintenance phase of the life cycle. The time spent and effort required to keep the software operational

after release is very significant. Despite the fact that it is a very important and challenging task; it is routinely the poorly managed headache that nobody wants to face.

Software maintenance is a very broad activity that includes error correction, enhancement of capabilities, deletion of obsolete capabilities, and optimization. The purpose of this phase is to preserve the value of the software over time. This phase may span for 5 to 50 years whereas development may be 1 to 3 years.

This model is easy to understand and reinforces the notion of “define before design” and “design before code”. This model expects complete and accurate requirements early in the process, which is unrealistic. Working software is not available until relatively late in the process, thus delaying the discovery of serious errors. It also does not incorporate any kind of risk assessment.

Problems of waterfall model

- (i) It is difficult to define all requirements at the beginning of a project.
- (ii) This model is not suitable for accomodating any change.
- (iii) A working version of the system is not seen until late in the project's life.
- (iv) It does not scale up well to large projects.
- (v) Real projects are rarely sequential.

Due to these weaknesses, the application of waterfall model should be limited to situations where the requirements and their implementation are well understood. For example, if an organisation has experience in developing accounting systems then building a new accounting system based on existing designs could be easily managed with the waterfall model.

2.1.3 Prototyping Model

A disadvantage of waterfall model as discussed in the last section is that the working software is not available until late in the process, thus delaying the discovery of serious errors. An alternative to this is to first develop a working prototype of the software instead of developing the actual software. The working prototype is developed as per current available requirements. Basically, it has limited functional capabilities, low reliability, and untested performance (usually low).

The developers use this prototype to refine the requirements and prepare the final specification document. Because the working prototype has been evaluated by the customer, it is reasonable to expect that the resulting specification document will be correct. When the prototype is created, it is reviewed by the customer. Typically this review gives feedback to the developers that helps to remove uncertainties in the requirements of the software, and starts an iteration of refinement in order to further clarify requirements as shown in Fig. 2.3.

The prototype may be a usable program, but is not suitable as the final software product. The reason may be poor performance, maintainability or overall quality. The code for the prototype is thrown away; however the experience gathered from developing the prototype helps in developing the actual system. Therefore, the development of a prototype might involve extra cost, but overall cost might turnout to be lower than that of an equivalent system developed using the waterfall model.

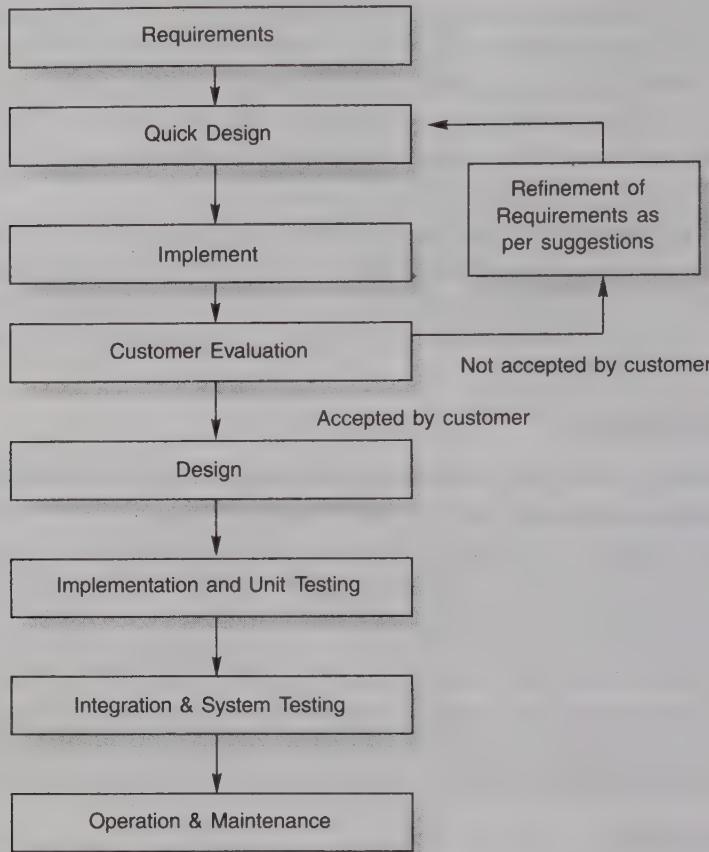


Fig. 2.3: Prototyping model

The developers should develop prototype as early as possible to speed up the software development process. After all, the 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 [SCHA96].

After the finalization of software requirement and specification (SRS) document, the prototype is discarded and actual system is then developed using the waterfall approach. Thus, it is used as an input to waterfall model and produces maintainable and good quality software. This model requires extensive participation and involvement of the customer, which is not always possible.

2.1.4 Iterative Enhancement Model

This model has the same phases as the waterfall model, but with fewer restrictions. Generally the phases occur in the same order as in the waterfall model, but these may be conducted in several cycles. A useable product is released at the end of each cycle, with each release providing additional functionality [BASI75].

During the first requirements analysis phase, customers and developers specify as many requirements as possible and prepare a SRS document. Developers and customers then prioritize

these requirements. Developers implement the specified requirements in one or more cycles of design, implementation and test based on the defined priorities. The model is given in Fig. 2.4.

The aim of the waterfall and prototyping models is the delivery of a complete, operational and good quality product. In contrast, this model does deliver an operational quality product at each release, but one that satisfies only a subset of the customer's requirements. The complete product is divided into releases, and the developer delivers the product release by release. A typical product will usually have many releases as shown in Fig. 2.4. At each release, customer has an operational quality product that does a portion of what is required. The customer is able to do some useful work after first release. With this model, first release may be available within few weeks or months, whereas the customer generally waits months or years to receive a product using the waterfall and prototyping model.

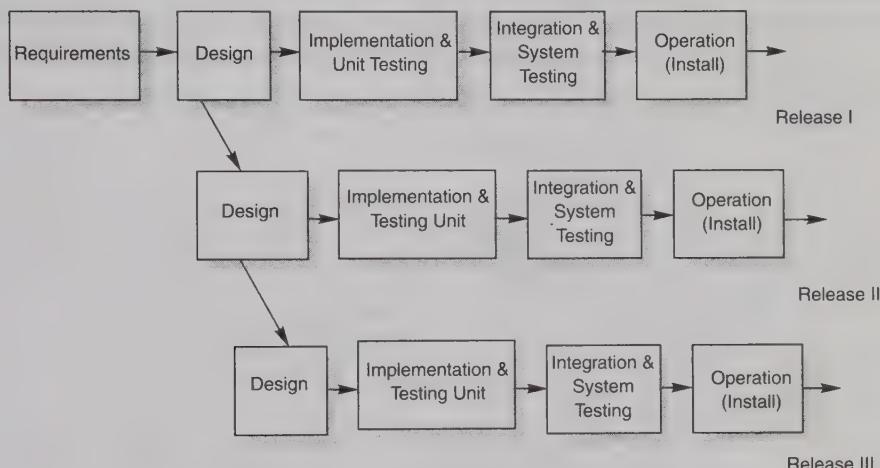


Fig. 2.4: Iterative enhancement model.

2.1.5 Evolutionary Development Model

Evolutionary development model resembles iterative enhancement model. The same phases as defined for the waterfall model occur here in a cyclical fashion. This model differs from iterative enhancement model in the sense that this does not require a useable product at the end of each cycle. In evolutionary development, requirements are implemented by category rather than by priority.

For example, in a simple database application, one cycle might implement the graphical user interface (GUI); another file manipulation; another queries; and another updates. All four cycles must complete before there is working product available. GUI allows the users to interact with the system; file manipulation allows data to be saved and retrieved; queries allow users to get data out of the system; and updates allow users to put data into the system. With any one of those parts missing, the system would be unusable.

In contrast, an iterative enhancement model would start by developing a very simplistic, but usable database. On the completion of each cycle, the system would become more sophisticated. It, however, provide all the critical functionality by the end of the first

cycle. Evolutionary development and iterative enhancement are somewhat interchangeable. Evolutionary development should be used when it is not necessary to provide a minimal version of the system quickly.

This model is useful for projects using new technology that is not well understood. This is also used for complex projects where all functionality must be delivered at one time, but the requirements are unstable or not well understood at the beginning.

2.1.6 Spiral Model

The problem with traditional software process models is that they do not deal sufficiently with the uncertainty, which is inherent to software projects. Important software projects have failed because project risks were neglected and nobody was prepared when something unforeseen happened. Barry Boehm recognized this and tried to incorporate the “project risk” factor into a life cycle model. The result is the spiral model, which was presented in 1986 [BOEH86] and is shown in Fig. 2.5.

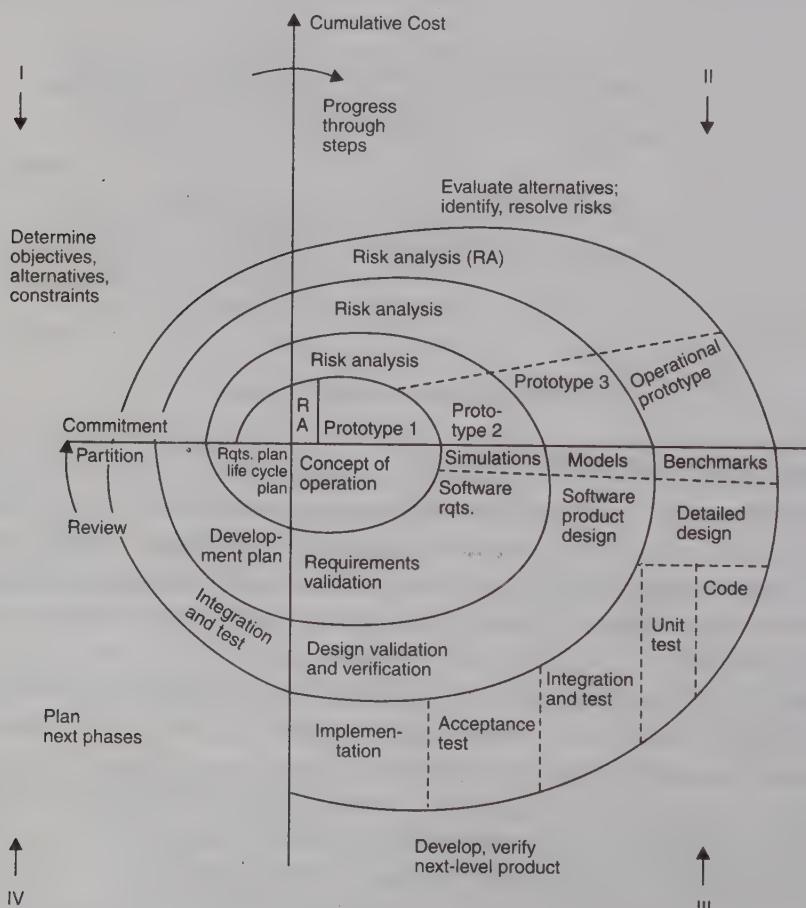


Fig. 2.5: Spiral model

The radial dimension of the model represents the cumulative costs. Each path around the spiral is indicative of increased costs. The angular dimension represents the progress made

in completing each cycle. Each loop of the spiral from X-axis clockwise through 360° represents one phase. One phase is split roughly into four sectors of major activities:

- Planning: Determination of objectives, alternatives and constraints
- Risk Analysis: Analyze alternatives and attempts to identify and resolve the risks involved
- Development: Product development and testing product
- Assessment: Customer evaluation

During the first phase, planning is performed, risks are analyzed, prototypes are built, and customers evaluate the prototype. During the second phase, a more refined prototype is built, requirements are documented and validated, and customers are involved in assessing the new prototype. By the time third phase begins, risks are known, and a somewhat more traditional development approach is taken [RAKI97].

The focus is the identification of problems and the classification of these into different levels of risks, the aim being to eliminate high-risk problems before they threaten the software operation or cost.

An important feature of the spiral model is that each phase is completed with a review by the people concerned with the project (designers and programmers). This review consists of a review of all the products developed up to that point and includes the plans for the next cycle. These plans may include a partition of the product in smaller portions for development or components that are implemented by individual groups or persons. If the plan for the development fails, then the spiral is terminated. Otherwise, it terminates with the initiation of new or modified software.

The advantage of this model is the wide range of options to accommodate the good features of other life cycle models. It becomes equivalent to another life cycle model in appropriate situations. It also incorporates software quality objectives into software development. The risk analysis and validation steps eliminate errors in the early phases of development.

The spiral model has some difficulties that need to be resolved before it can be a universally applied life cycle model. These difficulties include lack of explicit process guidance in determining objectives, constraints, alternatives; relying on risk assessment expertise; and providing more flexibility than required for many applications.

2.1.7 The Rapid Application Development (RAD) Model

This model was proposed by IBM in the 1980s through the book of James Martin entitled “Rapid Application Development”. Here, user involvement is essential from requirement phase to delivery of the product. The continuous user participation ensures the involvement of user's expectations and perspective in requirements elicitation, analysis and design of the system.

The process is started with building a rapid prototype and is given to user for evaluation. The user feedback is obtained and prototype is refined. The process continues, till the requirements are finalised. We may use any grouping technique (like FAST, QFD, Brainstorming Sessions; for details refer chapter 3) for requirements elicitation. Software requirement and specification (SRS) and design documents are prepared with the association of users.

There are four phases in this model and these are shown in Fig. 2.6.

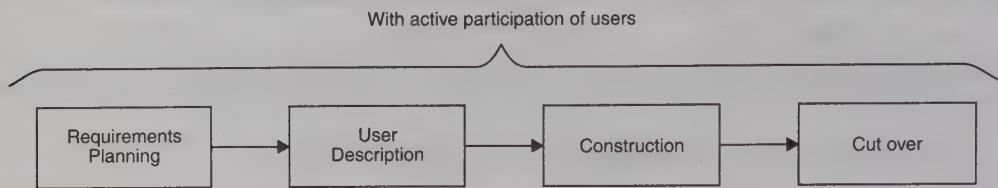


Fig. 2.6: RAD Model

(i) **Requirements planning phase.** Requirements are captured using any group elicitation technique. Some techniques are discussed in chapter 3. Only issue is the active involvement of users for understanding the project.

(ii) **User Description.** Joint teams of developers and users are constituted to prepare, understand and review the requirements. The team may use automated tools to capture information from the other users.

(iii) **Construction phase.** This phase combines the detailed design, coding and testing phase of waterfall model. Here, we release the product to customer. It is expected to use code generators, screen generators and other types of productivity tools.

(iv) **Cut over phase.** This phase incorporates acceptance testing by the users, installation of the system, and user training.

In this model, quick initial views about the product are possible due to delivery of rapid prototype. The development time of the product may be reduced due to use of powerful development tools. It may use CASE tools and frameworks to increase productivity. Involvement of user may increase the acceptability of the product.

If user cannot be involved throughout the life cycle, this may not be an appropriate model. Development time may not be reduced very significantly, if reusable components are not available. Highly specialized and skilled developers are expected and such developers may not be available very easily. It may not be effective, if system can not be properly modularised.

2.2 SELECTION OF A LIFE CYCLE MODEL

The selection of a suitable model is based on the following characteristics/categories:

- (i) Requirements
- (ii) Development team
- (iii) Users
- (iv) Project type and associated risk.

2.2.1 Characteristics of Requirements

Requirements are very important for the selection of an appropriate model. There are number of situations and problems during requirements capturing and analysis. The details are given in Table 2.1.

Table 2.1: Selection of a model based on characteristics of requirements

<i>Requirements</i>	<i>Waterfall</i>	<i>Prototype</i>	<i>Iterative enhancement</i>	<i>Evolutionary development</i>	<i>Spiral</i>	<i>RAD</i>
Are requirements easily understandable and defined?	Yes	No	No	No	No	Yes
Do we change requirements quite often?	No	Yes	No	No	Yes	No
Can we define requirements early in the cycle?	Yes	No	Yes	Yes	No	Yes
Requirements are indicating a complex system to be built	No	Yes	Yes	Yes	Yes	No

2.2.2 Status of Development Team

The status of development team in terms of availability, effectiveness, knowledge, intelligence, team work etc., is very important for the success of the project. If we know above mentioned parameters and characteristics of the team, then we may choose an appropriate life cycle model for the project. Some of the details are given in Table 2.2.

Table 2.2: Selection based on status of development team

<i>Development team</i>	<i>Waterfall</i>	<i>Prototype</i>	<i>Iterative enhancement</i>	<i>Evolutionary development</i>	<i>Spiral</i>	<i>RAD</i>
Less experience on similar projects	No	Yes	No	No	Yes	No
Less domain knowledge (new to the technology)	Yes	No	Yes	Yes	Yes	No
Less experience on tools to be used	Yes	No	No	No	Yes	No
Availability of training, if required	No	No	Yes	Yes	No	Yes

2.2.3 Involvement of Users

Involvement of users increases the understandability of the project. Hence user participation, if available, plays a very significant role in the selection of an appropriate life cycle model. Some issues are discussed in Table 2.3.

Table 2.3: Selection based on user's participation

Involvement of Users	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
User involvement in all phases	No	Yes	No	No	No	Yes
Limited user participation	Yes	No	Yes	Yes	Yes	No
User have no previous experience of participation in similar projects	No	Yes	Yes	Yes	Yes	No
Users are experts of problem domain	No	Yes	Yes	Yes	No	Yes

2.3.4 Type of Project and Associated Risk

Very few models incorporate risk assessment. Project type is also important for the selection of a model. Some issues are discussed in Table 2.4.

Table 2.4: Selection based on type of project with associated risk

Project type and risk	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Project is the enhancement of the existing system	No	No	Yes	Yes	No	Yes
Funding is stable for the project	Yes	Yes	No	No	No	Yes
High reliability requirements	No	No	Yes	Yes	Yes	No
Tight project schedule	No	Yes	Yes	Yes	Yes	Yes
Use of reusable components	No	Yes	No	No	Yes	Yes
Are resources (time, money people etc.) scarce?	No	Yes	No	No	Yes	No

An appropriate model may be selected based on options given in four Tables (*i.e.*, Table 2.1 to 2.4). Firstly, we have to answer the questions presented for each category by circling a yes or no in each table. Rank the importance of each category, or question within the category, in terms of the project for which we want to select a model. The total number of circled responses for each column in the tables decide an appropriate model. We may also use the category ranking to resolve the conflicts between models if the total in either case is close or the same.

REFERENCES

- [BASI75] Basili V.R., "Iterative Enhancement: A Practical Technique for Software Development", IEEE Trans on Software Engineering, SE-1, No. 4, 390–396, December 1975.
 - [BOEH86] Boehm B., "A Spiral Model for Software Development and Enhancement", ACM Software Engineering Notes, 14–24, August, 1986.
 - [RAKI97] Rakitin S.R., "Software Verification and Validation", Artech House Inc., Norwood, MA, 1997.
 - [SCHA96] Schach S., "Classical and Object Oriented Software Engineering", IRWIN, USA, 1996.
 - [TAKA96] Takang A.A. and P.A. Grubb, "Software Maintenance—Concepts and Practice", Int. Thomson Computer Press, Cambridge, U.K., 1996.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions.

EXERCISES

- 2.1. What do you understand by the term Software Development Life Cycle (SDLC)? Why is it important to adhere to a life cycle model while developing a large software product?
 - 2.2. What is software life cycle? Discuss the generic waterfall model.
 - 2.3. List the advantages of using waterfall model instead of adhoc build and fix model.
 - 2.4. Discuss the prototype model. What is the effect of designing a prototype on the overall cost of the software project?

- 2.5. What are the advantages of developing the prototype of a system?
- 2.6. Describe the type of situations where iterative enhancement model might lead to difficulties.
- 2.7. Compare iterative enhancement model and evolutionary development model.
- 2.8. Sketch a neat diagram of spiral model of software life cycle.
- 2.9. Compare the waterfall model and the spiral model of software development.
- 2.10. As we move outward along with process flow path of the spiral model, what can we say about the software that is being developed or maintained?
- 2.11. How does “project risk” factor affect the spiral model of software development?
- 2.12. List the advantages and disadvantages of involving a software engineer throughout the software development planning process.
- 2.13. Explain the spiral model of software development. What are the limitations of such a model?
- 2.14. Describe the rapid application development (RAD) model. Discuss each phase in detail.
- 2.15. What are the characteristics to be considered for the selection of a life cycle model?
- 2.16. What is the role of user participation in the selection of a life cycle model?
- 2.17. Why do we feel that characteristics of requirements play a very significant role in the selection of a life cycle model?
- 2.18. Write short note on “status of development team” for the selection of a life cycle model.
- 2.19. Discuss the selection process parameters for a life cycle model.

3

Software Requirements Analysis and Specifications

Contents

3.1 Requirements Engineering

- 3.1.1 Crucial Process Steps
- 3.1.2 Present State of Practice
- 3.1.3 Types of Requirements

3.2 Requirements Elicitation

- 3.2.1 Interviews
- 3.2.2 Brainstorming Sessions
- 3.2.3 Facilitated Application Specification Techniques
- 3.2.4 Quality Function Deployment
- 3.2.5 The Use Case Approach

3.3 Requirements Analysis

- 3.3.1 Data Flow Diagrams
- 3.3.2 Data Dictionaries
- 3.3.3 Entity Relationship Diagrams
- 3.3.4 Software Prototyping

3.4 Requirements Documentation

- 3.4.1 Nature of SRS
- 3.4.2 Characteristics of a Good SRS
- 3.4.3 Organisation of the SRS

3.5 Student Result Management System—Example

- 3.5.1 Problem Statement
- 3.5.2 Context Diagram
- 3.5.3 Data Flow Diagrams
- 3.5.4 ER Diagrams
- 3.5.5 Use Case Diagrams
- 3.5.6 Use Cases
- 3.5.7 SRS Document

3

Software Requirements Analysis and Specifications

When we receive a request for a new software project from the customer, first of all, we would like to understand the project. The new project may replace the existing system such as preparation of students semester results electronically rather than manually. Sometimes, the new project is an enhancement or extension of a current (manual or automated) system. For example, a web enabled student result declaration system that would enhance the capabilities of the current result declaration system. No matter, whether its functionality is old or new, each project has a purpose, usually expressed in what the system can do. Hence, goal is to understand the requirements of the customer and document them properly. A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfil the system's purpose.

The hardest part of building a software system is deciding precisely what to build. No other part of the conceptual work is so 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 [BR0095]. Throughout software industry's history, we have struggled with this truth. Defining and applying good, complete requirements is hard to work, and success in this endeavor has eluded many of us. Yet, we continue to make progress.

3.1 REQUIREMENTS ENGINEERING

Requirements describe the “what” of a system, not the “how”. Requirements engineering produces one large document, written in a natural language, contains a description of what the system will do without describing how it will do. The input to requirements engineering is the problem statement prepared by the customer. The problem statement may give an overview of the existing system alongwith broad expectations from the new system

3.1.1 Crucial Process Steps

The quality of a software product is only as good as the process that creates it. Requirements engineering is one of the most crucial activity in this creation process. Without well-written requirements specifications, developers do not know what to build, customers do not know what to expect, and there is no way to validate that the built system satisfies the requirements.

Requirements engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behaviour and its associated constraints [HSIA93]. This process consists of four steps as shown in Fig. 3.1.

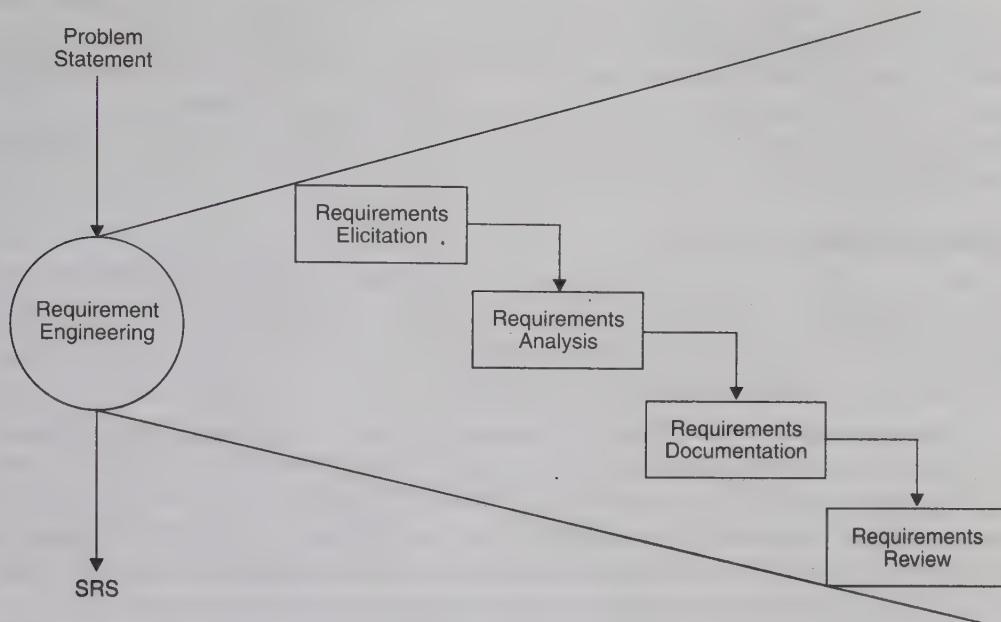


Fig. 3.1: Crucial process steps of requirement engineering.

- (i) **Requirements Elicitation:** This is also known as gathering of requirements. Here, requirements are identified with the help of customer and existing systems processes, if available.
- (ii) **Requirements Analysis:** Analysis of requirements starts with requirement elicitation. The requirements are analysed in order to identify inconsistencies, defects, omissions etc. We describe requirements in terms of relationships and also resolve conflicts, if any.
- (iii) **Requirements Documentation:** This is the end product of requirements elicitation and analysis. The documentation is very important as it will be the foundation for the design of the software. The document is known as software requirements specification (SRS).
- (iv) **Requirements Review:** The review process is carried out to improve the quality of the SRS. It may also be called as requirements verification. For maximum benefits, review and verification should not be treated as a discrete activity to be done only at the end of the preparation of SRS. It should be treated as continuous activity that is incorporated into the elicitation, analysis, and documentation.

The primary output of requirements engineering is requirements specifications. If it describes both hardware and software, it is a system requirements specification. If it describes only software, it is a software requirements specification. In either case, a requirements specification must treat the system as a black box. It must delineate inputs, outputs, the functional requirements that show external behaviour in terms of input, output, and their relationships, and nonfunctional requirements and their constraints, including performance, portability, and reliability.

The software requirements specification (SRS) should be internally consistent; consistent with existing documents; correct and complete with respect to satisfying needs; understandable to users, customers, designers, and testers; and capable of serving as a basis for both design and test. This SRS document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure in implementing the contracted system.

3.1.2 Present State of Practice

Most software development organizations agree to the fact that there should be a set of activities called requirements engineering and their success is vital to the success of the entire project. So why is the state of the practice no better than it is? There are several reasons, not all of them obvious [BERR98, DAV194, HSIA93]; and some are discussed below:

1. Requirements are difficult to uncover: Today we are automating virtually every kind of task-some that were previously done manually and some that have never been done before. In either kind of application, it is difficult, if not impossible to identify all the requirements, regardless of the techniques we use. No one can see a brand new system in its entirety. Even if someone could, the description is always incomplete at start. Users and developers must resort to trial and error to identify problems and solutions.

2. Requirements change: Because no user can come up with a complete list of requirements at the outset, the requirements get added and changed as the user begins to understand the system and his or her real needs. That is why we always have requirement changes. But, project schedule is seldom adjusted to reflect these modifications. Fluid requirements make it difficult to establish a baseline from which to design and test. Finally, it is hard to justify spending resources to make a requirement specification “perfect”, because it will soon change anyway. This is the biggest problem, and there is as yet no technology to overcome it. This problem is often used as an excuse to either eliminate or scale back requirements engineering effort.

3. Over-reliance on CASE tools: Computer Aided Software Engineering (CASE) tools are often sold as panaceas. These exaggerated claims, have created a false sense of trust, which could inflict untold damage on the Software Industry. CASE tools are as important to developers (including requirement writers) as word processors are to authors. However, we must not rely on requirements engineering tools without first understanding and establishing requirements engineering principles, techniques and processes. Furthermore, we must have realistic expectations from the tools.

4. Tight project schedule: Because of either lack of planning or unreasonable customer demand, many projects start with insufficient time to do a decent job. Sometimes, even the allocated time is reduced while the project is under way. It is also customary to reduce time set apart to analyze requirements, for early start of designing and coding, which frequently leads to disaster.

5. Communication barriers: Requirement engineering is communication intensive activity. Users and developers have different vocabularies, professional backgrounds, and tastes. Developers usually want more precise specifications while users prefer natural language. Selecting either results in misunderstanding and confusion.

6. Market-driven software development: Many of the software development is today market driven, developed to satisfy anonymous customers and to keep them coming back to buy upgrades.

7. Lack of resources: There may not be enough resources to build software that can do everything the customer wants. It is essential to rank requirements so that, in the face of pressure to release the software quickly, the most important can be implemented first.

Requirement problems are expensive and plague almost all systems and software development organizations. In most cases, the best we can hope for it is to detect errors in the requirements in time to contain them before the software is released [SAWY99]. Because of the concern for public safety, reputation and capital investments; developers began to recognize the need for clear, concise and complete requirements [COUN99].

3.1.3 Type of Requirements

There are different types of requirements such as:

- (i) Known requirements—Something a stakeholder believes to be implemented.
- (ii) Unknown requirements—Forgotten by the stakeholder because they are not needed right now or needed only by another stakeholder.
- (iii) Undreamt requirements—Stakeholder may not be able to think of new requirements due to limited domain knowledge.

The term stakeholder is used to refer to any one who may have some direct or indirect influence on the system requirements. Stakeholder includes end-users who will interact with the system and every one else in an organisation who will be affected by it [SOMM01].

A known, unknown, or undreamt requirement may be functional or nonfunctional. Functional requirements describe what the software has to do. They are often called product features.

Non-functional requirements are mostly quality requirements that stipulate how well the software does what it has to do. Non functional quality requirements that are especially important to users include specifications of desired performance, availability, reliability, usability and flexibility. Non functional requirements for developers are maintainability, portability, and testability.

Some requirements are architectural, such as component-naming compatibility, interfaceability, upgradability, etc. Other requirements are constraints, such as system design constraints, standards conformance, legal issues and organisational issues. Constraints can come from users or organisations and may be functional or non-functional [ROBE02].

Example 3.1: 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 expectation from the new software system.

Solution: The problem statement is prepared by the Examination division of the University and is given below:

"A University conducts a 4-semester M. Tech programme. The students are offered four theory papers and two Lab papers (practicals) during Ist, IIInd and IIIrd semesters. The theory

papers offered in these semesters are categorized as either 'Core' or 'Elective'. Core papers do not have an alternative subject, whereas elective papers may have two or more alternative subjects. Thus a student can study any subject out of the choices available for an elective paper.

In Ist, IInd and IIIrd semesters, 2 core papers and 2 elective papers are offered to each student. The students are also required to submit a term paper minor project in IInd and IIIrd semesters each. In IVth semester the students have to give a seminar and submit a dissertation on a topic/subject area of their interest.

The evaluation of each subject is done out of 100 marks. During the semester, minor exams are conducted for each semester. Students are also required to submit assignments as directed by the corresponding faculty and maintain Lab records for practicals. Based on the students' performance in minor exams, assignments, Lab records and their attendance, marks out of 40 are given in each theory paper and practical paper. These marks out of 40 account for internal evaluation of the students. At the end of each semester, major exams are conducted in each subject (theory as well as practical). These exams are evaluated out of 60 marks and account for external evaluation of the students. Thus, the total marks of a student in a subject are obtained by adding the marks obtained in internal and external evaluation.

Every subject has some credit points assigned to it. If the total marks of a student are $>= 50$ in a subject, he/she is considered 'Pass' in that subject otherwise the student is considered 'Fail' in that subject. If a student passes in a subject he/she earns all the credit points assigned to that subject, but if the student fails in a subject he/she does not earn any credit point in that subject. At any time, the latest information about subjects being offered in various semesters and their credit points can be obtained from University Website.

It is required to develop a system that will manage information about subjects offered in various semesters, students enrolled in various semesters, elective (s) opted by various students in different semesters, marks and credit points obtained by students in different semesters. The system should also have the ability to generate printable mark sheets for each student. Semester-wise detailed mark lists and student performance reports also need to be generated.

Example 3.2: A university wishes to develop a software system for library management activities. Design the problem statement for the software company.

Solution: The problem statement prepared by the library staff of the university is given below. Here activities are explained point wise rather than paragraph wise.

A Software has to be developed for automating the manual library system of a University. The system should be standalone in nature. It should be designed to provide functionalities as explained below:

1. Issue of Books:

- (a) A student of any course should be able to get books issued.
- (b) Books from **General section** are issued to all but **Book bank** books are issued only for their respective courses.
- (c) A limitation is imposed on the number of books a student can be issued.

- (d) A maximum of 4 books from Book bank and 3 books from General section per student is allowed.
- (e) The books from Book bank are issued for entire semester while books from General section are issued for 15 days only.
- (f) The software takes the current system date as the date of issue and calculates the corresponding date of return.
- (g) A bar code detector is used to save the student information as well as book information.
- (h) The due date for return of the book is stamped on the book.

2. Return of Books:

- (a) Any person can return the issued books.
- (b) The student information is displayed using the bar code detector.
- (c) The system displays the student details on whose name the books were issued as well as the date of issue and return of the book.
- (d) The system operator verifies the duration for the issue and if the book is being returned after the specified due date, a fine of Re 1 is charged for each day.
- (e) The information is saved and the corresponding updations take place in the database.

3. Query Processing:

- (a) The system should be able to provide information like:
 - (i) Availability of a particular book
 - (ii) Availability of books of any particular author.
 - (iii) Number of copies available of the desired book.
- (b) The system should be able to reserve a book for a particular student for 24 hrs if that book is not currently available.

The system should also be able to generate reports regarding the details of the books available in the library at any given time.

The corresponding printouts for each entry (issue/return) made in the system should be generated.

Security provisions like the login authenticity should be provided. Each user should have a user id and a password. Record of the users of the system should be kept in the log file.

Provision should be made for full backup of the system.

3.2 REQUIREMENTS ELICITATION

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 [WIEG99].

The real requirements actually reside in user's mind. Hence the most important goal of requirement engineering is to find out what users really need. Users need can be identified only if we understand the expectations of the users from the desired software.

It is the activity that helps to understand the problem to be solved. Requirements are gathered by asking questions, writing down the answers, asking other questions, etc. Hence, requirements gathering is the most communications intensive activity of software development. Developers and Users have different mind set, expertise and vocabularies. Due to communication gap, there are chances of conflicts that may lead to inconsistencies, misunderstanding and omission of requirements.

Therefore, requirements elicitation requires the collaboration of several groups of participants who have different background. On the one hand, customers and users have a solid background in their domain and have a general idea of what the software should do. However, they may have little knowledge of software development processes. On the other hand, the developers have experience in developing software but may have little knowledge of everyday environment of the users. Moreover each group may be using incompatible terminologies.

There are number of requirements elicitation methods and few of them are discussed in the following sections. Some people think that one methodology is applicable to all situations, however, generally speaking, one methodology cannot possibly be sufficient for all conditions [MACA96]. We select a particular methodology for the following reason(s):

- (i) It is the only method that we know.
- (ii) It is our favorite method for all situations.
- (iii) We understand intuitively that the method is effective in the present circumstances.

Clearly, third reason demonstrates the most maturity and leads to improved understanding of stakeholder's needs and thus resulting system will satisfy those needs. Unfortunately, most of us do not have the insight necessary to make such an informed decision, and therefore rely on the first two reasons [HICK03].

3.2.1 Interviews

After receiving the problem statement from the customer, the first step is to arrange a meeting with the customer. During the meeting or interview, both the parties would like to understand each other. Normally specialised developers, often called 'requirement engineers' interact with the customer. The objective of conducting an interview is to understand the customer's expectations from the software. Both parties have different feelings, goals, opinions, vocabularies, understandings, but one thing is common, both want the project to be a success. With this in mind, requirement engineers normally arrange interviews. Requirement engineers must be open minded and should not approach the interview with pre-conceived notions about what is required.

Interview may be open-ended or structured. In open-ended interview, there is no pre-set agenda. Context free questions may be asked to understand the problem and to have an overview of the situation. For example, for a "result management system", requirement engineer may ask:

- Who is the controller of examination ?
- Who has requested for such a software ?
- How many officers are placed in the examination division ?
- Who will use the software ?
- Who will explain the manual system ?

- Is there any opposition for this project ?
- How many stakeholders are computer friendly ?

Such questions help to identify all stakeholders who will have interest in the software to be developed.

In structured interview, agenda of fairly open questions is prepared. Sometimes a proper questionnaire is designed for the interview. Interview may be started with simple questions to set people at ease. After making atmosphere comfortable and calm, specific questions may be asked to understand the requirements. The customer may be allowed to voice his or her perceptions about a possible solution.

Selection of Stakeholder. It will be impossible to interview every stakeholder. Thus, representatives from groups must be selected based on their technical expertise, domain knowledge, credibility, and accessibility. There are several groups to be considered for conducting interviews:

- (i) Entry level personnel: They may not have sufficient domain knowledge and experience, but may be very useful for fresh ideas and different views.
- (ii) Mid-level stakeholders: They have better domain knowledge and experience of the project. They know the sensitive, complex and critical areas of the project. Hence, requirement engineers may be able to extract meaningful and useful information. Project leader should always be interviewed.
- (iii) Managers or other Stakeholders: Higher level management officers like vice-Presidents, General Managers, Managing Directors should also be interviewed. Their expectations may provide different but rich information for the software development.
- (iv) Users of the software: This group is perhaps the most important because they will spend more time interacting with the software than any one else. Their information may be eye opener and may be original at times. Only caution required is that they may be biased towards existing systems.

Types of questions: Questions should be simple and short. Two or three questions rolled into one can lead to compound requirements statements that are difficult to interpret and test. It is important to prepare questions, but reading from the questionnaire or only sticking to it is not desirable. We should be open for any type of discussion and any direction of the interview. For the “result management system” we may ask:

- Are there any problems with the existing system ?
- Have you faced calculation errors in past ?
- What are the possible reasons of malfunctioning ?
- How many students are enrolled presently ?
- What are the possible benefits of computerising this system ?
- Are you satisfied with current processes and policies ?
- How are you maintaining the records of previous students ?
- What data, required by you, exists in other systems ?
- What problems do you want this system to solve ?

- Do you need additional functionality for improving the performance of the system ?
- What should be the most important goal of the proposed development ?

These questions will help to start the communication that is essential for understanding the requirements. At the end of this, we may have wide variety of expectations from the proposed software.

3.2.2 Brainstorming Sessions

Brainstorming is a group technique that may be used during requirements elicitation to understand the requirements. The group discussions may lead to new ideas quickly and help to promote creative thinking.

It is intended to generate lots of ideas, with full understanding that they may not be useful. The theory is that having a long list of requirements from which to choose is far superior to starting with a blank slate. Requirements in the long list can be categorized, prioritized, and pruned [ROBE02].

Brainstorming has become very popular and is being used by most of the companies. It promotes creative thinking, generates new ideas and provides platform to share views, apprehensions expectations and difficulties of implementation. All participants are encouraged to say whatever ideas come to mind, whether they seem relevant or not. No one will be criticized for any idea, no matter how goofy it seems, as the responsibility of the participant is to generate views and not to vet them.

This group technique may be carried out with specialised groups like actual users, middle level managers etc., or with total stakeholders. Sometimes unnatural groups are created that may not be appreciated and are uncomfortable for participants. At times, only superficial responses may be gathered to technical questions. In order to handle such situations, a highly trained facilitator may be required. The facilitator may handle group bias and group conflicts carefully. The facilitator should also be cautious about individual egos, dominance and will be responsible for smooth conduct of brainstorming sessions. He or she will encourage the participants, ensure proper individual and group behaviour and help to capture the ideas. The facilitator will follow a published agenda and restart the creative process if it falters.

Every idea will be documented in such a way that everyone can see it. White boards, overhead transparencies or a computer projection system can be used to make it visible to every participant. After the session, a detailed report will be prepared and facilitator will review the report. Every idea will be written in simple english so that it conveys same meaning to every stakeholder. Incomplete ideas may be listed separately and should be discussed at length to make them complete ideas, if possible. Finally, a document will be prepared which will have list of requirements and their priority, if possible.

3.2.3 Facilitated Application Specification Technique

This approach is similar to brainstorming sessions and the objective is to bridge the expectation gap – a difference between what developers think they are supposed to build and what customers think they are going to get. In order to reduce expectation gap, a team oriented approach is developed for requirements gathering and is called Facilitated Application Specification Technique (FAST).

This approach encourages the creation of a joint team of customers and developers who work together to understand the expectations and propose a set of requirements. The basic guidelines for FAST are given below:

- Arrange a meeting at a neutral site for developers and customers.
- Establishment of rules for preparation and participation.
- Prepare an informal agenda that encourages free flow of ideas.
- Appoint a facilitator to control the meeting. A facilitator may be a developer, a customer, or an outside expert.
- Prepare a definition mechanism-Board, flip charts, worksheets, wall stickies, etc.
- Participants should not criticize or debate.

FAST session preparations

Each FAST attendee is asked to make a list of objects that are:

- (i) part of the environment that surrounds the system
- (ii) produced by the system
- (iii) used by the system.

In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size) and performance criteria (e.g., speed, accuracy) are 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 [PRES2K].

Activities of FAST session

The activities during FAST session may have the following steps:

- Each participant presents his or her lists of objects, services, constraints, and performance for discussion. Lists may be displayed in the meeting by using board, large sheet of paper or any other mechanism, so that they are visible to all the participants.
- The combined lists for each topic are prepared by eliminating redundant entries and adding new ideas.
- The combined lists are again discussed and consensus lists are finalised by the facilitator.
- Once the consensus lists have been completed, the team is divided into smaller subteams, each works to develop mini-specifications for one or more entries of the lists.
- Each subteam then presents mini-specifications to all FAST attendees. After discussion, additions or deletions are made to the lists. We may get new objects, services, constraints, or performance requirements to be added to original lists.
- During all discussions, the team may raise an issue that cannot be resolved during the meeting. An issues list is prepared so that these ideas will be considered later.
- Each attendee prepares a list of validation criteria for the product/system and presents the list to the team. A consensus list of validation criteria is then created.

- A subteam may be asked to write the complete draft specifications using all inputs from the FAST meeting.

FAST is not a panacea of the problems encountered in early requirements elicitation but it helps to understand the requirements and bridge the expectation gap of develops and customers.

3.2.4 Quality Function Deployment

It is a quality management technique that helps to incorporate the voice of the customer. The voice is then translated into technical requirements. These technical requirements are documented and results is the software requirements and specification document. These requirements are further translated into design requirements. Here, customer satisfaction is of prime concern and thus QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the software engineering process [PRES2K]. Three types of requirements are identified [ZULT92]:

(i) **Normal requirements.** The objectives and goals of the proposed software are discussed with the customer. If this category of requirements (normal) are present, the customer is satisfied. Examples related to result management system might be: entry of marks, calculation of results, merit list report, failed students report, etc.

(ii) **Expected requirements.** These requirements are implicit to the software product and may be so obvious that customer does not explicitly state them. If such requirements are not present, customer will be dissatisfied with the software. Examples of expected requirements may be: protection from unauthorised access, some warning system for wrong entry of data, the feasibility for modification of any record only by a fool proof system for the identification of person alongwith date and time of modification, etc.

(iii) **Exciting requirements.** Some features go beyond the customer's expectations and prove to be very satisfying when present. Examples of exciting requirements for result management system may be: if an unauthorised access is noticed by the software, it should immediately shutdown all the processes and an E-mail is generated to the system administrator, an additional copy of important files is maintained and may be accessed by system administrator only, sophisticated virus protection system etc.

The QFD method has the following steps [ROBE02]:

- Identify all the stakeholders e.g., customers, users, and developers. Also identify any initial constraints identified by the customer that affect requirements development.
- List out requirements from customer ; inputs, considering different viewpoints. Requirements are expression of what the system will do, which is both perceptible and of value to customers. Some customer's expectations may be unrealistic or ambiguous and may be translated into realistic or unambiguous requirements if possible.
- A value indicating a degree of importance, is assigned to each requirement. Thus, customer determines the importance of each requirement on a scale of 1 to 5 as given below:

5 points:	Very important
4 points:	important

- | | |
|-----------|---|
| 3 points: | not important, but nice to have |
| 2 points: | not important |
| 1 point: | unrealistic, requires further exploration |

Stakeholders will have their own unique set of criteria for determining the ‘importance’, or ‘value’ of a requirement. It may be based on cost / benefit analysis particular to the project.

Requirement Engineers may review the final list of requirements and categorise like:

- (i) it is possible to achieve
- (ii) it should be deferred and the reason thereof
- (iii) it is impossible and should be dropped from consideration.

The first category requirements will be implemented as per priority (Importance value) assigned with every requirement. If time and effort permits, second category requirements may be reviewed and few of them may be transferred to category first for implementation.

3.2.5 The Use Case Approach

For many years, requirement engineers have used stories or scenarios to explain the interaction of a user with the proposed software system in order to gather the requirements. More recently, Ivar Jacobson and others [JACO99] formalised this into use case approach to requirements elicitation and modeling. Initially, use cases were designed for object oriented software development world, however, they can be applied to any project that follow any development approach because the user does not care how we develop the software. The focus on what the users need to do with the system is much more powerful than the traditional elicitation approach of asking users what they want the system to do [WIEG99].

This approach uses a combination of text and pictures in order to improve the understanding of requirements. The Use cases describe what of a system and not ‘how’. They only give functional view of the system.

The terms use case, use case scenario, and use case diagram are often interchanged, but in fact they are different. Use cases are structured outline or templates for the description of user requirements, modeled in a structured language like english. Use case scenarios are unstructured descriptions of user requirements. Use case diagrams are graphical representations that may be decomposed into further levels of abstraction. The following components are used for the design of the use case approach.

Actor: An actor or external agent, lies outside the system model, but interacts with it in some way. An actor may be a person, machine, or an information system that is external to the system model. An actor is represented as stick figure and is not part of the system itself. Customers, users, external devices, or any external entity interacting with the system are treated as actors.

We should not confuse the actors with the devices they use. Devices are typically mechanisms that actors use to communicate with the system, but they are not actors themselves. We are writing this book on a computer, but the keyboard is not the user of the word processing program; but we are. Other devices, such as disk drives, tape drives, or communication equipment including printers have no place in use case diagram, although they are important to the

design of the system. The purpose of devices is to support some required behaviour 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 [BITT03].

Cockburn [COCK01] distinguishes between primary and secondary actors. A primary actor is one having a goal requiring the assistance of the system. A secondary actor is one from which the system needs assistance.

Use Cases: A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to deliver the services that satisfies the goal. It also includes possible variants of this sequence, e.g., alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure to complete the service because of exceptional behaviour, error handling etc. The system is treated as a 'black box', and the interactions with the system, including responses, are as perceived from outside the system.

Thus, use cases capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals. A complete set of use cases specifies all the different ways to use the system, and therefore, defines all behaviour required of the system, bounding the scope of the system.

Use cases are written in an easy to understand structured narrative—the vocabulary of the domain. The users may validate the use cases and may involve in the process of gathering and defining the requirements [MALA01].

There is no standard use case template for writing use cases. The Jacobson et al. [JACO99] proposed a template for writing use cases and is given in Table 3.1(a). This template captures requirements in an effective way and is therefore becoming popular. Another similar template is also given in Table 3.1(b) which is also used by many organisations.

Use case guidelines

The following provides an outline of a process for creating use cases:

- Identify all the different users of the system.
- Create a user profile for each category of users, including all the roles the users play that are relevant to the system. For each role, identify all the significant goals the users have that the system will support. A statement of the system's value proposition is useful in identifying significant goals.
- Create a use case for each goal, following the use case template. Maintain the same level of abstraction throughout the use case. Steps in higher level use cases may be treated as goals for lower level (*i.e.*, more detailed), sub-use cases.
- Structure the use cases. Avoid over-structuring, as this can make the use cases harder to follow.
- Review and validate with users.

Table 3.1(a): 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. <ul style="list-style-type: none"> 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 flows 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.	Pre-conditions. Pre-conditions that need to be satisfied for the use case to perform.
6.	Post-conditions. Define the different states in which you expect the system to be in, after the use case executes.
7.	Extension Points.

Table 3.1(b): Use case template

1.	Introduction. Describe brief purpose of the use case.
2.	Actors. List the actors that interact and participate in this use case.
3.	Pre-condition. Condition that need to be satisfied for the use case to execute.
4.	Post-condition. After the execution of the use case, different states of the systems are defined here.
5.	Flow of Events. <ul style="list-style-type: none"> 5.1. Basic flow. List the primary events that will occur when this use case is executed. 5.2. Alternate flow. Any other possible flow in this use case, if there, should be separately listed. A use case may have many alternate flows.
6.	Special Requirements. Business rules for the basic and alternate flows should be listed as special requirements. Both success and failure scenarios should be described.
7.	Related use cases. List the related use cases, if any.

Use case diagrams

A use case diagram visually represents what happens when an actor interacts with a system. Hence, a use case diagram captures the functional aspects of a system. The system is shown as a rectangle with the name of the system (or subsystem) inside, the actors are shown as stick figures (even the non human ones), the use cases are shown as solid bordered ovals labeled with the name of the use case, and relationships are lines or arrows between actors and use cases and/or between the use cases themselves. These components are given in Fig. 3.2.

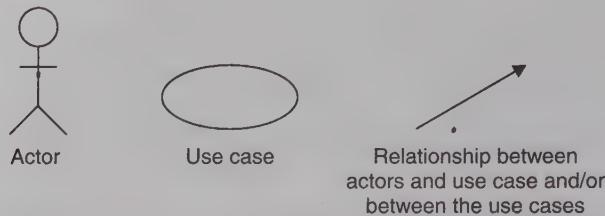


Fig. 3.2: Components of use case diagram.

Actors appear outside of the rectangle since they are external to the system. Use cases appear within the rectangle, providing functionality. A relationship or association is a solid line between an actor and each use case in which actor participates—the involvement can be any kind, not necessarily one of the actor initiating the use case functionality.

Fig. 3.3 shows an example of a use case diagram whose “Problem statement” is given in Table 3.2.

Table 3.2: Problem statement for railway reservation system

PROBLEM STATEMENT FOR RAILWAY RESERVATION SYSTEM

A Software has to be developed for automating the manual railway reservation system. The system should be distributed in nature. It should be designed to provide functionalities as explained below:

1. **Reserve Seat:** A passenger should be able to reserve seats in the train. A reservation form is filled by the passenger and given to the clerk, who then checks for the availability of seats for the specified date of journey. If seats are available, then the entries are made in the system regarding the train name, train number, date of journey, boarding station, destination, person name, sex and total fare. Passenger is asked to pay the required fare and the tickets are printed. If the seats are not available then the passenger is informed.
2. **Cancel Reservation:** A passenger wishing to cancel a reservation is required to fill a form. The passenger then submits the form and the ticket to the clerk. The clerk then deletes the entries in the system and changes in the reservation status of that train. The clerk crosses the ticket by hand to mark as cancelled.
3. **Update Train Information:** Only the administrator enters any changes related to the train information like change in the train name, train number, train route etc. in the system.
4. **Report Generation:** Provision for generation of different reports should be given in the system. The system should be able to generate reservation chart, monthly train report etc.
5. **Login:** For security reasons all the users of the system are given a user *id* and a password. Only if the *id* and password are correct the user is allowed to enter the system.
6. **View Reservation Status:** All the users should be able to see the reservation status of the train online. The user needs to enter the train number and the pin number printed on his ticket so that the system can display his current reservation status like confirmed, RAC or Wait listed.
7. **View Train Schedule:** Provision should be given to see information related to the train schedules for the entire train network. The user should be able to see the train name, train number, boarding and destination stations, duration of journey etc.

Use cases should not be used to capture all the details of system. The granularity to which we define use cases in a diagram should be enough to keep the use case diagram uncluttered and readable, yet, be complete without missing significant aspects of the required

functionality. Design issues should not be discussed at all. Use cases are meant to capture “what” the system is, and not “how” the system will be designed or built. Hence use cases should be free of any design characteristics. If we end up defining design characteristics in a use case, we need to go back to the drawing board and start again.

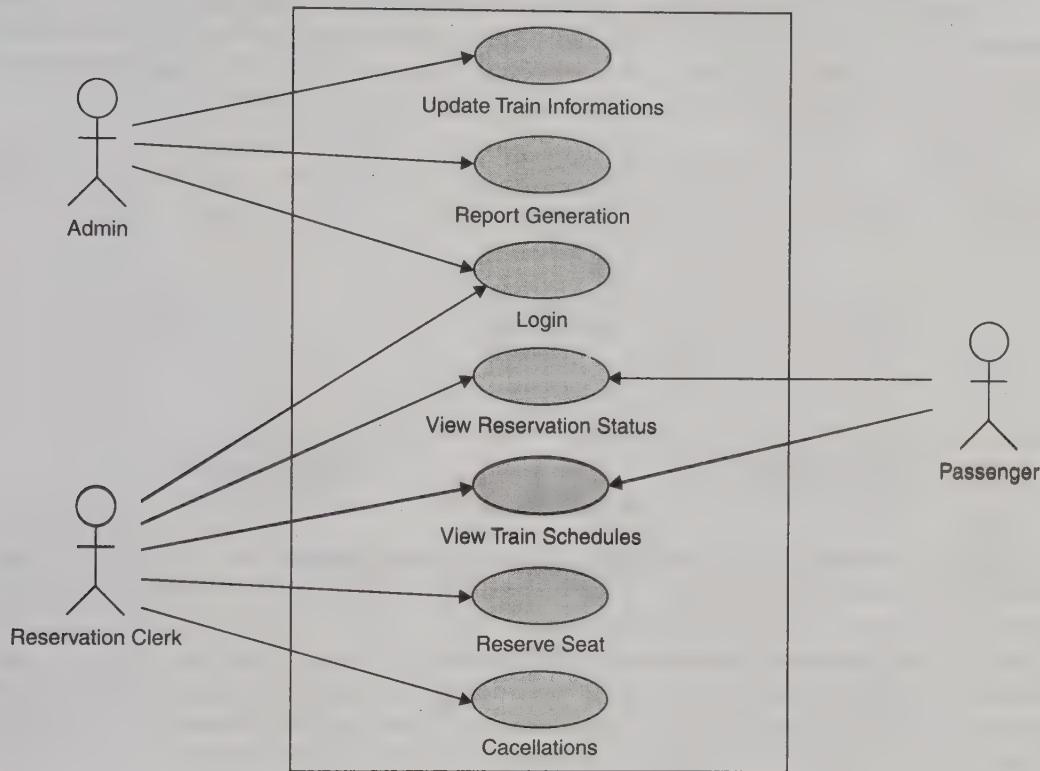


Fig. 3.3: Use Case Diagram for Railway Reservation System.

3.3 REQUIREMENTS ANALYSIS

Requirements analysis is very important and essential activity after elicitation. We analyze, refine and scrutinize the gathered requirements in order to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the

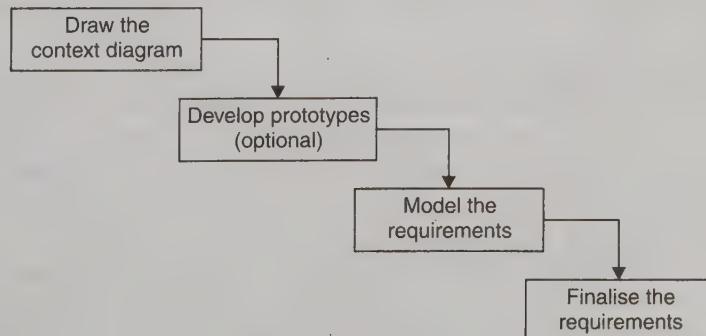
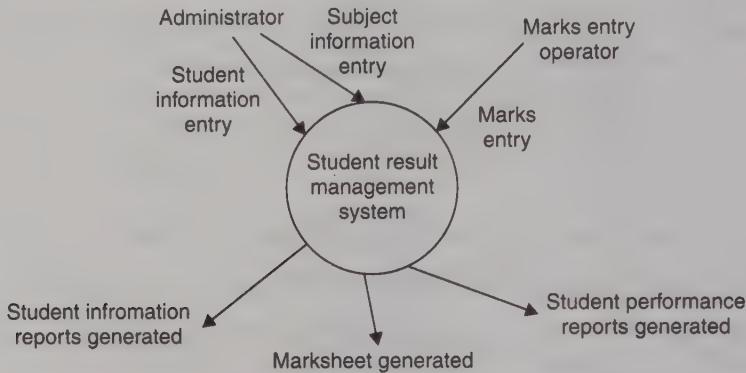


Fig. 3.4: Requirements analysis steps.

entire system. After the completion of analysis, it is expected that the understandability of the project may improve significantly. Here, we may also interact with the customer to clarify points of confusion and to understand which requirements are more important than others. The various steps of requirements analysis are shown in Fig. 3.4.

(i) **Draw the context diagram.** The context diagram is a simple model that defines the boundaries and interfaces of the proposed system with the external world. It identifies the entities outside the proposed system that interact with the system. The context diagram of student result management system (as discussed earlier) is given below:



(ii) **Development of a prototype (optional).** One effective way to find out what the customer really wants is to construct a prototype, something that looks and preferably acts like a part of the system they say they want.

We can use their feedback to continuously modify the prototype until the customer is satisfied. Hence, prototype helps the client to visualise the proposed system and increase the understanding of requirements. When developers and users are not certain about some of the requirements, a prototype may help both the parties to take a final decision.

Some projects are developed for general market. In such cases, the prototype should be shown to some representative sample of the population of potential purchasers. Even though, persons who try out a prototype may not buy the final system, but their feedback may allow us to make the product more attractive to others. Some projects are developed for a specific customer under contract. On such projects, only that customer's opinion counts, so the prototype should be shown to the prospective users in the customer organisation.

The prototype should be built quickly and at a relatively low cost. Hence it will always have limitations and would not be acceptable in the final system. This is an optional activity. Although many organisations are developing prototypes for better understanding before the finalisation of SRS.

(iii) **Model the requirements.** This process usually consists of various graphical representations of the functions, data entities, external entities and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing and superfluous requirements. Such models include data flow diagrams, entity relationship diagrams, data dictionaries, state-transition diagrams etc.

(iv) **Finalise the requirements.** After modeling the requirements, we will have better understanding of the system behaviour. The inconsistencies and ambiguities have been identified and corrected. Flow of data amongst various modules has been analysed. Elicitation and analysis activities have provided better insight to the system. Now we finalise the analysed requirements and next step is to document these requirements in a prescribed format.

3.3.1 Data Flow Diagrams

Data flow diagrams (DFD) are used widely for modeling the requirements. They have been used for many years prior to the advent of computers. DFDs show the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or a bubble chart.

The following observations about DFDs are important [DAV190]:

1. All names should be unique. This makes it easier to refer to items in the DFD.
2. Remember that a DFD is not a flow chart. Arrows in a flow chart represent the order of events; arrows in DFD represent flowing data. A DFD does not imply any order of events.
3. Suppress logical decisions. If we ever have the urge to draw a diamond-shaped box in a DFD, suppress that urge! A diamond-shaped box is used in flow charts to represent decision points with multiple exit paths of which only one is taken. This implies an ordering of events, which makes no sense in a DFD.
4. Do not become bogged down with details. Defer error conditions and error handling until the end of the analysis.

Standard symbols for DFDs are derived from the electric circuit diagram analysis and are shown in Fig. 3.5 [SAGE90].

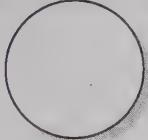
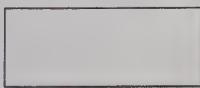
Symbol	Name	Function
	Data Flow	Used to connect processes to each other, to sources or sinks; the arrowhead indicates direction of data flow.
	Process	Performs some transformation of input data to yield output data.
	Source or Sink (External Entity)	A source of system inputs or sink of system outputs.
	Data Store	A repository of data; the arrowheads indicate net inputs and net outputs to store.

Fig. 3.5: Symbols for data flow diagrams.

A circle (bubble) shows a process that transforms data inputs into data outputs. A curved line shows flow of data into or out of a process or data store. A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have element or group of elements. Source or sink is an external entity and acts as a source of system inputs or sink of system outputs.

Leveling

The DFD may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. A level-0 DFD, also called a fundamental system model or *context diagram* represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively [PRES2K]. Then the system is decomposed and represented as a DFD with multiple bubbles. Parts of the system represented by each of these bubbles are then decomposed and documented as more and more detailed DFDs. This process may be repeated at as many levels as necessary until the problem at hand is well understood. It is important to preserve the number of inputs and outputs between levels; this concept is called leveling by DeMacro. Thus, if bubble “A” has two inputs, x_1 and x_2 , and one output y , then the expanded DFD, that represents “A” should have exactly two external inputs and one external output as shown in Fig. 3.6 [DEMA79, DAV190].

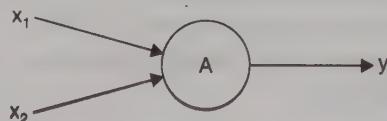


Fig. 3.6: Level-0 DFD.

The level-0 DFD, also called context diagram of result management system is shown in Fig. 3.7. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flows may also need to be decomposed. Level-1 DFD of result management system is given in Fig. 3.8.

This provides a detailed view of requirements and flow of data from one bubble to the another.

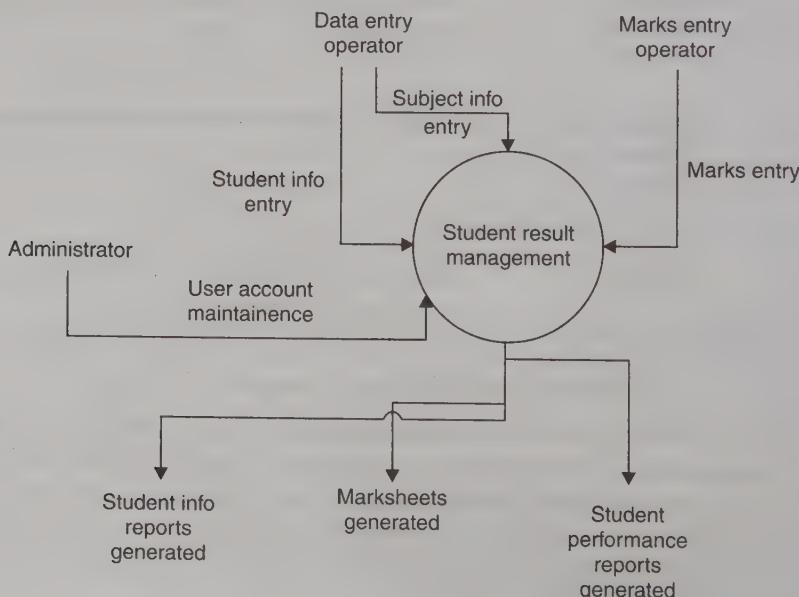


Fig. 3.7: 0-Level DFD or context diagram of result management system.

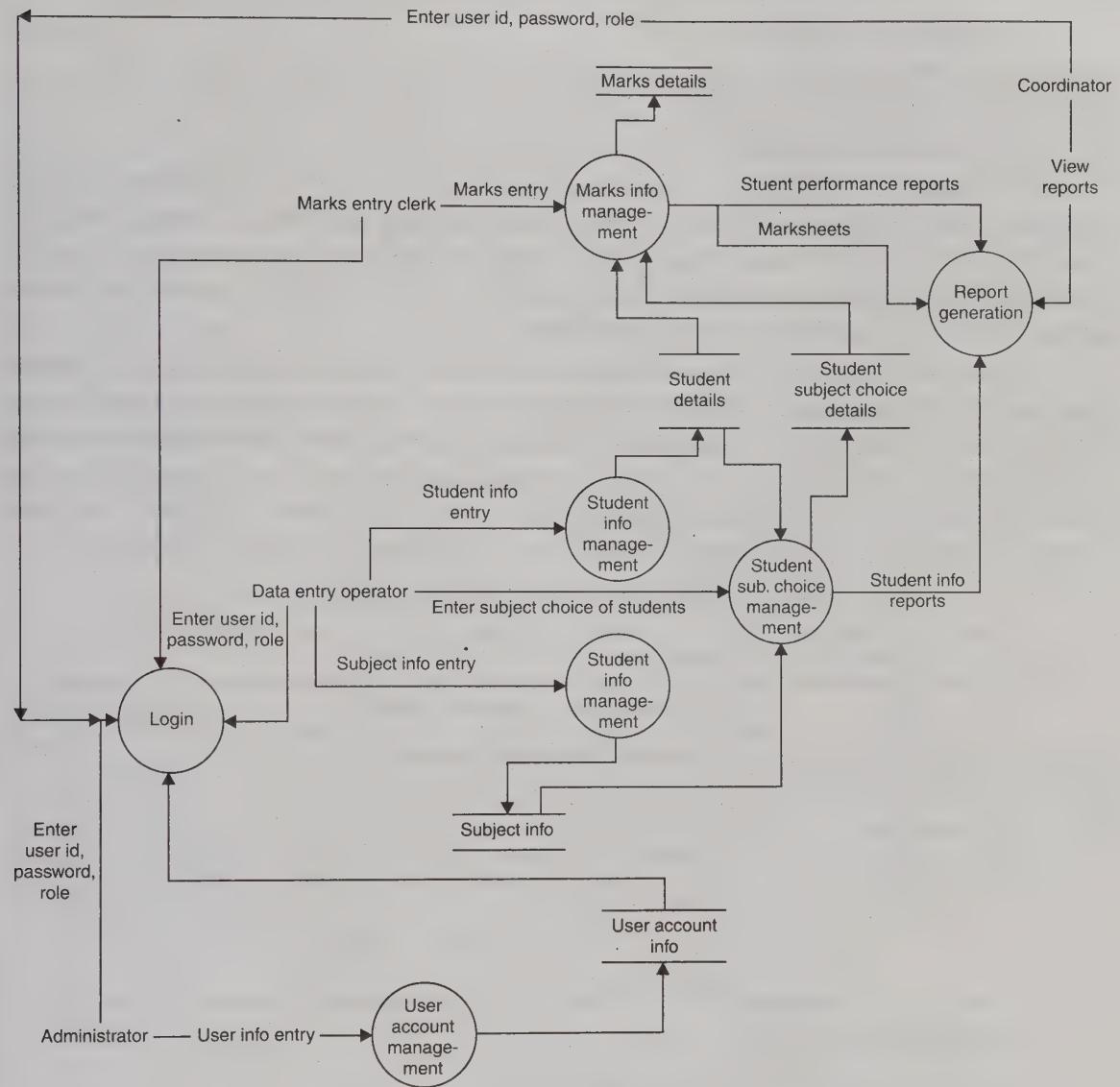


Fig. 3.8: Level-1 DFD of result management system.

3.3.2 Data Dictionaries

Families of DFDs can become quite complex. One way to manage this complexity is to augment DFDs with data dictionaries (DD). Data dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developer use the same definitions and terminologies. Typical information stored includes:

- Name of the data item
- Aliases (other names for item)

- Description/purpose
- Related data items
- Range of values
- Data structure definition/form

The name of the data item is self-explanatory. Aliases include other names by which this data item is called *e.g.*, DEO for Data Entry Operator and DR for deputy Registrar. Description/Purpose is a textual description of what the data item is used for or why it exists. Related data items capture relationships between data items *e.g.*, total_marks must always equal to internal_marks plus external_marks.

Range of values records all possible values, *e.g.*, total marks must be positive and between 0 to 100. Data flows capture the names of the processes that generate or receive the data item. If data item is primitive, then data structure definition/form captures the physical structure of the data item. If the data is itself a data aggregate, then data structure definition/form captures the composition of the data items in terms of other data items [DAV190]. The mathematical operators used within the data dictionary are defined in Table 3.3 [DEMA79].

Table 3.3: Data dictionary notation and mathematical operators

Notation	Meaning
$x = a + b$	x consists of data elements a and b
$x = [a/b]$	x consists of either data element a or b
$x = a$	x consists of an optional data element a
$x = y\{a\}$	x consists of y or more occurrences of data element a
$x = \{a\}z$	x consists of z or fewer occurrences of data element a
$x = y\{a\}z$	x consists of some occurrences of data element a which are between y and z .

The data dictionary can be used to:

- Create an ordered listing of all data items.
- Create an ordered listing of a subset of data items.
- Find a data item name from a description.
- Design the software and test cases.

3.3.3 Entity-Relationship Diagrams

Another tool for requirement analysis is the entity-relationship diagram, often called as “E-R diagram” [CHEN76]. It is a detailed logical representation of the data for an organization and uses three main constructs *i.e.*, data entities, relationships, and their associated attributes.

Entities

An entity is a fundamental thing of an organization about which data may be maintained. An entity has its own identity, which distinguishes it from each other entity. An entity type is the description of all entities to which a common definition and common relationships and attributes apply.

Consider a university that offers both regular and distance education programmes. These Programmes are offered to national and international students.

PROGRAMME and STUDENT are both entity types in this example. Regular and distance education are entities of PROGRAMME whereas national and international are entities of STUDENT.

We use capital letters in naming an entity type and in an ER diagram the name is placed inside a rectangle representing that entity as shown in Fig. 3.9.

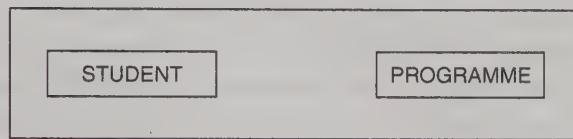


Fig. 3.9: Two entity types in an E-R diagram.

Relationships

A relationship is a reason for associating two entity types. These relationships are sometimes called binary relationships because they involve two entity types. Some forms of data model allow more than two entity types to be associated. A STUDENT is registered for a PROGRAMME. Relationships are represented by diamond notation in the E-R diagram as shown in Fig. 3.10.



Fig. 3.10: Relationships added to ERD.

We consider another example in which, a teaching department of a university is interested in tracking which subjects each of its students has completed. This leads to a relationship called “completes” between the STUDENT and SUBJECT entity types. This is shown in Fig. 3.11.

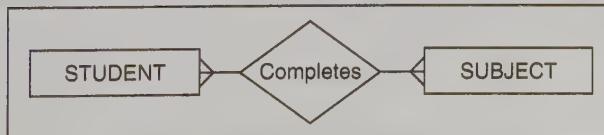


Fig. 3.11: Relationships in ERD.

As indicated by the arrows, this is a many-to-many relationship. Each student may complete more than one subject, and more than one student may complete each subject.

Degree of relationships

The degree of a relationship is the number of entity types that participate in that relationship. Thus, relationship “completes” shown in Fig. 3.11 is of degree two, since there are two entity types: STUDENT and SUBJECT. The three most common relationships in E-R models are unary (degree 1), binary (degree 2), and ternary (degree three). Higher-degree relationships are possible, but they are rarely encountered in practice.

Unary relationship

This is also called recursive relationship. It is a relationship between the instances of one entity type. An instance is a single occurrence of an entity type. There may be many instances of an entity type. For example, there is one STUDENT entity type in universities, but there may be hundreds of instances of this entity type in the database. In Fig. 3.12, Is-Married-to is shown as one to one relationship between instances of the PERSON entity type. That is, each person may be currently married to one other person. In the second example, “Is friend of” is shown as one to many relationships between instances of the STUDENT entity type.

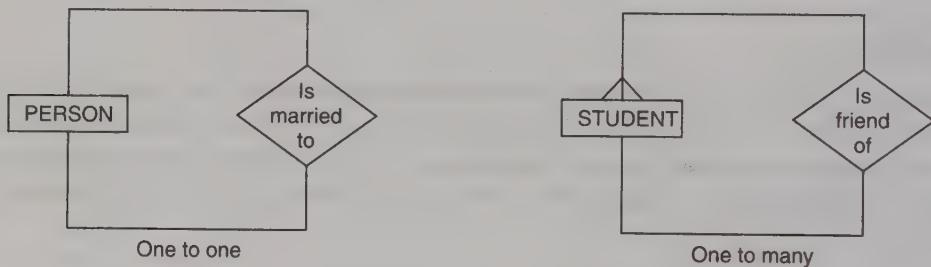


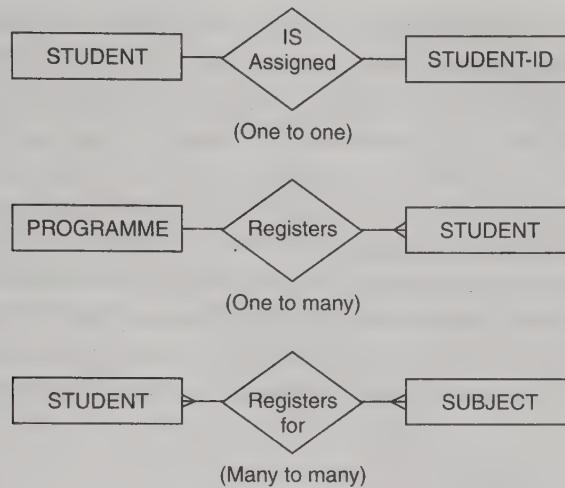
Fig. 3.12: Unary relationships.

Binary relationship

It is a relationship between instances of two entity types and is the most common type of relationship encountered in E-R diagrams. Fig. 3.13 shows three examples.

The first (one to one) indicates that a STUDENT is assigned a STUDENT-ID, and each STUDENT-ID is assigned to a STUDENT. The second (one to many) indicates that a PROGRAMME may have many students, and each STUDENT belongs to only one PROGRAMME.

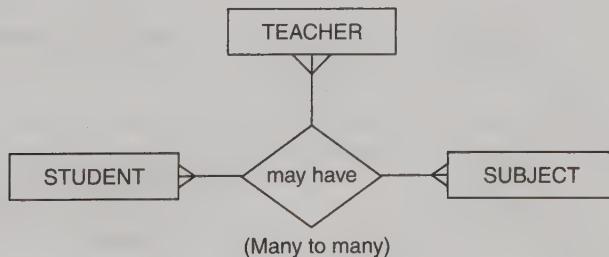
The third (many to many) shows that a STUDENT may register for more than one SUBJECT, and that each SUBJECT may have many STUDENT registrants.

**Fig. 3.13:** Binary relationships.

Ternary relationships

It is a simultaneous relationship amongst instances of three entity types. In Fig. 3.14, the relationship “may have” provides the association of three entities *i.e.*, TEACHER, STUDENT and SUBJECT. All three entities are many-to many participants. There may be one or many participants in a ternary relationship.

In general, “*n*” entities can be related by the same relationship and is known as *n*-ary relationship.

**Fig. 3.14:** Ternary relationship.

Cardinalities and optionality

Suppose that there are two entity types, A and B, that are connected by a relationship. The cardinality of a relationship is the number of instances of entity B that can be associated with each instance of entity A.

Consider the example shown in Fig. 3.15. a student may register for many subjects.

**Fig. 3.15:** Use of cardinality.

In the terminology, we have discussed so far, this example has “one-to many” relationship. Yet it may also be true that a subject may not have any student at specific instance of time. We need a more precise notation to indicate the range of cardinalities for a relationship.

The minimum cardinality of a relationship is the minimum number of instances of entity B that may be associated with each instance of entity A. If minimum number of students available for a subject is zero, we say that subject is an optional participant in the “register for” relationship. When the minimum cardinality of a relationship is one, then we say entity B is a mandatory participant in the relationship. The maximum cardinality is the maximum number of instances. In our example, maximum is “many”. The modified E-R diagram is given in Fig. 3.16. The zero through the line near the SUBJECT entity means a minimum cardinality of zero, while the crow’s foot notation means a “many” maximum cardinality.

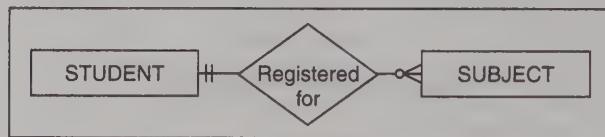


Fig. 3.16: Modified ER diagram.

Cardinality of relationships

It can be used to identify relationships between entity types. The cardinality of relationships is given in Fig. 3.17.

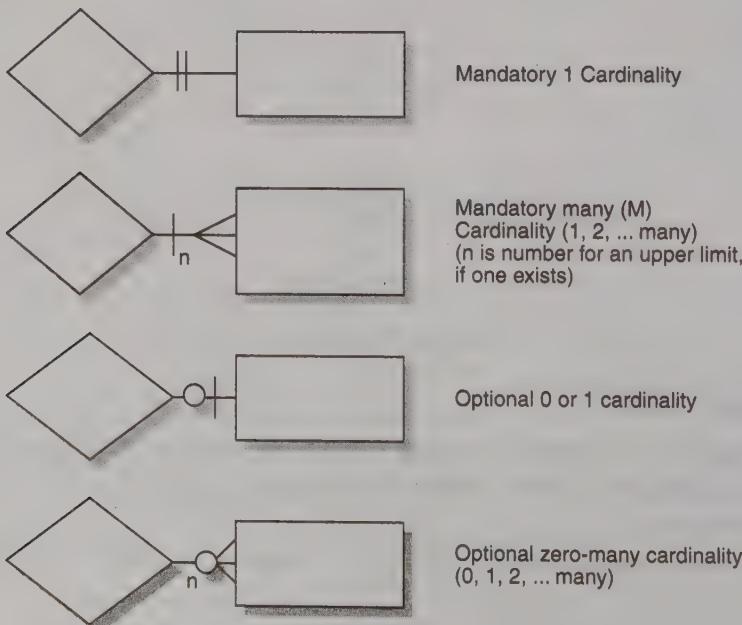


Fig. 3.17: Relationship cardinality [HOFF99]

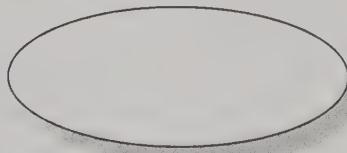
Attributes

Each entity type has a set of attributes associated with it. An attribute is a property or characteristic of an entity that is of interest to the organization. Following are some typical entity types and associated attributes:

STUDENT: Student_ID, Student_Name, Address, Phone_Number

EMPLOYEE: Employee_ID, Employee_Name, Address.

We use an initial capital letter, followed by lowercase letters, and nouns in naming an attribute. In E-R diagram, we can visually represent an attribute by placing its name as an ellipse with a line connecting it to the associated entity. Notation for attribute is

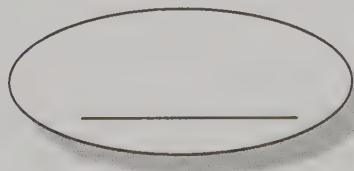


Attribute

Candidate keys and identifier

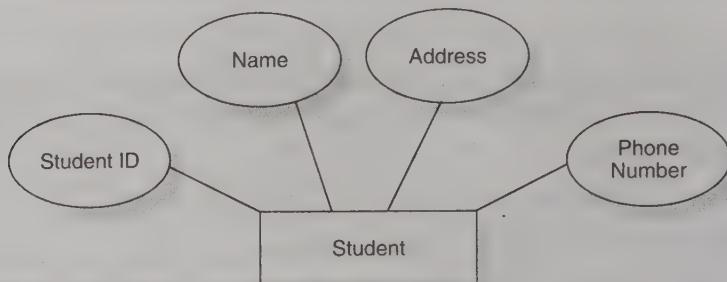
Every entity type must have an attribute or set of attributes that distinguishes one instance from other instances of the same type. A candidate key is an attribute (or combination of attributes) that uniquely identifies each instance of an entity type. A candidate key for a STUDENT entity type might be student_ID.

Some entities may have more than one candidate key. One candidate key for EMPLOYEE is Employee_ID, a second is the combination of Employee_Name and Address. If there is more than one candidate, the designer must choose one of the candidate keys as the identifier. An identifier is a candidate key that has been selected to be used as the unique characteristic for an entity type. Notation for identifier is



Identifier

The following diagram shows the representation for a STUDENT entity type using E-R notation [HOFF99].



Using entity relationship diagram to represent data is still an important technique today. It should not, however, be used in isolation, but together with techniques that fully represent the business objects we encounter every day.

The data flow diagram and the E-R diagram, each highlight a different aspect of the same system. As a consequence, there are one-to-one correspondences that must be checked to ensure that an E-R diagram and a data flow diagram are consistent over all applications. This suggests that it is desirable to use both methods such that we can view the logical issues from the two perspectives generated by these approaches.

3.3.4 Software Prototyping

Prototyping is the technique of constructing a partial implementation of a system so that customers, users, or developers can learn more about a problem or a solution to that problem. It is a partial implementation because if it were full implementation, it would be the system, not a prototype of it.

It allows users to explore and criticize proposed systems before undergoing the cost of a full-scale development. The field of prototyping software systems has emerged around two prototyping technologies, *i.e.*, throwaway and evolutionary. In throwaway approach, the prototype software is constructed in order to learn about the problem or its solution and is usually discarded after the desired knowledge is gained. In the evolutionary approach, the prototype is constructed in order to learn about the problem or its solution in successive steps. Once the prototype has been used and the requisite knowledge is gained, the prototype is then adapted to satisfy the, now better-understood, needs. The prototype is then used again, more is learned, and the prototype is re-adapted. This process repeats indefinitely until the prototype system satisfies all needs and thus evolves into the real system [DAV190]. Hence, in evolutionary prototyping the focus is on achieving functionality for demonstrating a portion of the system to the end user for feedback and system growth. The prototype emerges as the actual system downstream in the software life cycle. As with each iteration in development, functionality is added and then translated to an efficient implementation.

The benefits of developing a prototype early in the software process are [SOMM96]

1. Misunderstanding between software developers and customers may be identified as the system functions are demonstrated.
2. Missing user requirements may be detected.
3. Difficult-to-use or confusing user requirements may be identified and refined.

4. A working system is available quickly to demonstrate the feasibility and usefulness of the application to management.
5. The prototype serves as a basis for writing the specification of the system.

Software prototyping taxonomy

A range of possibilities exists for prototyping software systems. Any form of prototyping is perceived better than not prototyping at all. Several taxonomies have been proposed and served as basis for prototyping [RATC88, HOOP89, CERI86, CARE90, HEKM87]. However, the two most popular prototyping approaches mentioned earlier are briefly described as:

1. Throw-away prototyping. In this approach, prototype is constructed with the idea that it will be discarded, after the analysis is complete, and the final system is built from the scratch. This prototype is generally built quickly so as to enable the user to rapidly interact with the requirements determination early and thoroughly. Since the prototype will ultimately be discarded, it need not necessarily be fast operating, maintainable and having extensive fault tolerant capabilities.

During the requirement phase, a quick and dirty throwaway prototype can be constructed and given to user in order to determine the feasibility of a requirement, validate that a particular function is really necessary, uncover missing requirements and determine the viability of a user interface. During preliminary and detailed design, a quick and dirty prototype can be built to give a feeling and overview of final system to the user. Here, development of prototype should be quick, because its advantage exists only if results from its use are available in a timely fashion. It can be dirty because there is no justification for building quality into a product that will be discarded. Among the dirty characteristics to be considered are no design, no comments, no test plans, no idea about coupling and cohesion etc.

The most common steps for this approach are: (i) Writing a preliminary SRS (ii) implementing the prototype based on those requirements (iii) achieving user experience with the prototype (iv) Writing the real SRS and then (v) developing the real product.

2. Evolutionary Prototyping. In this approach, the prototype is built with the idea that it will eventually be converted into the final system. It will not be built in a “dirty” fashion. The evolutionary prototype evolves into the final product, and thus it must exhibit all the quality attributes of the final product and must follow the traditional life cycle. It is required to deploy the product, obtain experience using it, then based on that experience go back and redo the requirements, redesign, recode, retest, and redeploy. After gaining more experience, it is time to repeat the entire process again. This ensures the creation of all necessary documents and the presence of all necessary reviews. In fact, the only shortcuts that should be taken in building evolutionary prototypes are (i) building only those parts of the product that are understood (leaving other parts to later generations of the prototype) (ii) lowering the importance of performance. Using this will increase the probability that version $i + 1$ will meet user’s real needs because users have already used version i and supplied feedback on its performance.

The differences between these two approaches are given below:

Sr. No.	Approach and Characteristics	Throwaway	Evolutionary
1	Development Approach	Quick and Dirty , No rigor	No Sloppiness, Rigorous
2	What to build	Build only difficult parts	Build understood parts first and build on solid foundations
3	Design drivers	Optimize development time	Optimize Modifiability
4	Ultimate Goal	Throw it away	Evolve it

Prototyping pitfalls

Prototyping has not been as successful as anticipated in some organizations for a variety of reasons [TOZE87]. Training, efficiency, applicability, and behaviour can each have a negative impact on using software prototyping techniques.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort. Prototyping can have execution inefficiencies with the associated tools and this question may be argued as a negative aspect of prototyping.

This new approach of providing feedback early to the end user may result in a problem related to the behavior of the end user and developers. An end user with a previously unfortunate system development effort can be biased in future interactions with development teams.

Prototyping opportunities

Not to prototype at all should simply not be an option in software development. The benefits of software prototyping are obvious and established. The end user cannot throw the ambiguous and incomplete software needs and expect the development team to return the finished software system after some period of time with no problems in the deliverables.

One of the major problems incorporating this technology is the large investment that exists in software system maintenance. The idea of completely re-engineering an existing software system with current technology is not feasible. There is, however, a threshold that exists where the expected life span of a software system justifies that the system would be better maintained after being re-engineered in this technology. Total re-engineering should be planned rather than as a reaction to a crisis situation. At minimum, prototyping technology could be used on critical portion of an existing software system. This minimal approach could be used as a means to transition an organization to total re-engineering.

Software prototyping must be integrated within an organization through training, case studies, and library development. In situations where this full range of commitment to this technology is lacking, e.g. only developers training is provided, when problems begin to arise in using the technology a normal reaction of management is to revert back to what has worked in the past.

The end user involvement becomes enhanced when changes in requirements can be prototyped and agreed to before any development proceeds. Similarly, during development of

the actual system or even later into maintenance, should the requirements change; the prototype is enhanced and agreed to before the actual changes become confirmed.

3.4 REQUIREMENTS DOCUMENTATION

Requirements documentation is very important activity after the requirements elicitation and analysis. This is the way to represent requirements in a consistent format. Requirements document is called Software Requirements Specification (SRS).

The SRS is a specification for a particular Software product, program or set of programs that performs certain functions in a specific environment. It serves a number of purposes depending on who is writing it. First, the SRS could be written by the customer of a system. Second, the SRS could be written by a developer of the system. The two scenarios create entirely different situations and establish entirely different purposes for the document. First case, SRS is used to define the needs and expectations of the users. The second case, SRS is written for different purpose and serve as a contract document between customer and developer.

This reduces the probability of the customer being disappointed with the final product. The SRS written by developer (second case) is of our interest and discussed in the subsequent sections.

3.4.1 Nature of the SRS

The basic issues that SRS writer(s) shall address are the following:

1. **Functionality:** What the software is supposed to do?
2. **External Interfaces:** How does the software interact with people, the system's hardware, other hardware, and other software?
3. **Performance:** What is the speed, availability, response time, recovery time, etc. of various software functions?
4. **Attributes:** What are the considerations for portability, correctness, maintainability, security, reliability etc.?
5. **Design constraints imposed on an implementation:** Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

Since the SRS has a specific role to play in the software development process, SRS writer(s) should be careful not to go beyond the bounds of that role. This means the SRS

1. should correctly define all the software requirements. A software requirement may exist because of the nature of the task to be solved or because of a special characteristic of the project.
2. should not describe any design or implementation details. These should be described in the design stage of the project.
3. should not impose additional constraints on the software. These are properly specified in other documents such as a software quality assurance plan.

Therefore, a properly written SRS limits the range of valid designs, but does not specify any particular design.

3.4.2 Characteristics of a good SRS

The SRS should be:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable

Each of the above mentioned characteristics is discussed below: [THAY97, IEEE87, IEEE97].

Correct

The SRS is correct if, and only if; every requirement stated therein is one that the software shall meet. There is no tool or procedure that assures correctness. If the software must respond to all button presses within 5 seconds and the SRS stated that “the software shall respond to all buttons presses within 10 second”, then that requirement is incorrect.

Unambiguous

The SRS is unambiguous if, and only if; every requirement stated therein has only one interpretation. Each sentence in the SRS should have the unique interpretation. Imagine that a sentence is extracted from the SRS, given to ten people who are asked for their interpretation. If there is more than one such interpretation, then that sentence is probably ambiguous.

In cases, where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific. The SRS should be unambiguous to both those who create it and to those who use it. However, these groups often do not have the same background and therefore do *not tend to describe* software requirements in the same way.

Requirements are often written in natural language (for example, English). Natural language is inherently ambiguous. A natural language SRS should be reviewed by an independent party to identify ambiguous use of a language so that it can be corrected. This can be avoided by using a particular requirement specification language. Its language processors automatically detect many lexical, syntactic, and semantic errors. Disadvantage is the time required to learn the language which may also not be understandable to the customers/users. Moreover, these languages tend to be better at expressing only certain types of requirements and addressing certain types of systems.

Complete

The SRS is complete if, and only if; it includes the following elements:

1. All significant requirements, whether relating to functionality, performance, design constraints, attributes or external interfaces.

2. Definition of their responses of the software to all realizable classes of *input data* in *all realizable classes of situations*. Note that it is important to specify the responses to both valid and invalid values.
3. Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.

Consistent

The SRS is consistent if, and only if, no subset of individual requirements described in it conflict. There are three types of likely conflicts in the SRS:

1. The specified characteristics of real-world objects may conflict. For example.
 - (a) The format of an output report may be described in one requirement as tabular but in another as textual.
 - (b) One requirement may state that all lights shall be green while another states that all lights shall be blue.
2. There may be logical or temporal conflict between two specified actions, for example,
 - (a) One requirement may specify that the program will add two inputs and another may specify that the program will multiply them.
 - (c) One requirement may state that "A" must always follow "B", while another requires that "A and B" occur simultaneously.
3. Two or more requirements may describe the same real-world object but use different terms for that object. For example, a program's request for a user input may be called a "prompt" in one requirement and a "cue" in another. The use of standard terminology and definitions promotes consistency.

Ranked for importance and/or stability

The SRS is ranked for importance and/or stability if each requirement in it has an identifier to indicate either the importance or stability of that particular requirement.

Typically, all requirements are not equally important. Some requirements may be essential, especially for life critical applications, while others may be desirable. Each requirement should be identified to make these differences clear and explicit. Another way to rank requirements is to distinguish classes of requirements as essential, conditional and optional.

Verifiable

The SRS is verifiable, if and only if, every requirement stated therein is verifiable. A requirement is verifiable, if and only if, there exists some finite cost-effective process with which a person or machine can check that the software meets the requirements. In general any ambiguous requirement is not verifiable.

Nonverifiable requirements include statement, such as "works well", "good human interface", and "shall usually happen". These requirements cannot be verified because it is impossible to define the terms "good", "well", or "usually". The statement that "the program

shall never enter an infinite loop" is nonverifiable because the testing of this quality is theoretically impossible.

An example of a verifiable statement is "output of the program shall be produced within 20 seconds of event \times 60% of the time; and shall be produced within 30 seconds of event \times 100% of the time." This statement can be verified because it uses concrete terms and measurable quantities.

If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised.

Modifiable

The SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.

The requirements should not be redundant. Redundancy itself is not an error, but it can easily lead to errors. Redundancy can occasionally help to make an SRS more readable, but a problem can arise when the redundant document is updated. For instance, a requirement may be altered in only one of the places out of the many places where it appears.

The SRS then becomes inconsistent. Whenever redundancy is necessary, the SRS should include explicit cross-references to make it modifiable.

Traceable

The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Two types of traceability are recommended.

1. Backward traceability: This depends upon each requirement explicitly referencing its source in earlier documents.
2. Forward traceability: This depends upon each requirement in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially important when the software product enters the operation and maintenance phase. As code and design documents are modified, it is essential to be able to ascertain the complete set of requirements that may be affected by those modifications.

3.4.3 Organization of the SRS

The Institute of Electrical and Electronics Engineers (IEEE) has published guidelines and standards to organize an SRS document [IEEE87, IEEE94]. Different projects may require their requirements to be organized differently, that is, there is no one method that is suitable for all projects. It provides different ways of structuring the SRS. The first two sections of the SRS are the same in all of them. The specific tailoring occurs in section 3 entitled "specific requirements". The general organization of an SRS is given in Fig. 3.18 [IEEE93].

1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions, Acronyms, and Abbreviations
 - 1.4 References
 - 1.5 Overview
2. The Overall Description
 - 2.1 Product Perspective
 - 2.1.1 System Interfaces
 - 2.1.2 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. Change Management Process
5. Document Approvals
6. Supporting Information

Fig. 3.18: Organisation of SRS [IEEE-std. 830-1993].

1. Introduction

The following subsections of the Software Requirements Specifications (SRS) document should provide an overview of the entire SRS.

1.1 Purpose

Identify the purpose of this SRS and its intended audience. In this subsection, describe the purpose of the particular SRS and specify the intended audience for the SRS.

1.2 Scope

In this subsection:

- (i) Identify the software product(s) to be produced by name
- (ii) Explain what the software product(s) will, and, if necessary, will not do
- (iii) Describe the application of the software being specified, including relevant benefits, objectives, and goals
- (iv) Be consistent with similar statements in higher-level specifications if they exist.

1.3 Definitions, Acronyms, and Abbreviations

Provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendices in the SRS or by reference to documents. This information may be provided by reference to an Appendix.

1.4 References

In this subsection:

- (i) Provide a complete list of all documents referenced elsewhere in the SRS
- (ii) Identify each document by title, report number (if applicable), date, and publishing organization
- (iii) Specify the sources from which the references can be obtained.

This information can be provided by reference to an appendix or to another document.

1.5 Overview

In this subsection:

- (i) Describe what the rest of the SRS contains
- (ii) Explain how the SRS is organized.

2. The Overall Description

Describe the general factors that affect the product and its requirements. This section does not state specific requirements. Instead, it provides a background for those requirements, which are defined in section 3, and makes them easier to understand.

2.1 Product Perspective

Put the product into perspective with other related products. If the product is independent and totally self-contained, it should be so stated here. If the SRS defines a product that is a component of a larger system, as frequently occurs, then this subsection relates the require-

ments of the larger system to functionality of the software and identifies interfaces between that system and the software.

A block diagram showing the major components of the large system, interconnections, and external interfaces can be helpful.

The following subsections describe how the software operates inside various constraints.

2.1.1 System Interfaces

List each system interface and identify the functionality of the software to accomplish the system requirement and the interface description to match the system.

2.1.2 Interfaces

Specify:

- (i) The logical characteristics of each interface between the software product and its users.
- (ii) All the aspects of optimizing the interface with the person who must use the system.

2.1.3 Hardware Interfaces

Specify the logical characteristics of each interface between the software product and the hardware components of the system. This includes configuration characteristics. It also covers such matters as what devices are to be supported, how they are to be supported and protocols.

2.1.4 Software Interfaces

Specify the use of other required software products and interfaces with other application systems. For each required software product, include:

- (i) Name
- (ii) Mnemonic
- (iii) Specification number
- (iv) Version number
- (v) Source

For each interface, provide:

- (i) Discussion of the purpose of the interfacing software as related to this software product
- (ii) Definition of the interface in terms of message content and format.

2.1.5 Communications Interfaces

Specify the various interfaces to communications such as local network protocols, etc.

2.1.6 Memory Constraints

Specify any applicable characteristics and limits on primary and secondary memory.

2.1.7 Operations

Specify the normal and special operations required by the user such as:

- (i) The various modes of operations in the user organization
- (ii) Periods of interactive operations and periods of unattended operations

- (iii) Data processing support functions
- (iv) Backup and recovery operations.

2.1.8 Site Adaptation Requirements

In this section:

- (i) Define the requirements for any data or initialization sequences that are specific to a given site, mission, or operational mode
- (ii) Specify the site or mission-related features that should be modified to adapt the software to a particular installation.

2.2 Product Functions

Provide a summary of the major functions that the software will perform. Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product.

For clarity:

- (i) The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time.
- (ii) Textual or graphic methods can be used to show the different functions and their relationships. Such a diagram is not intended to show a design of a product but simply shows the logical relationships among variables.

2.3 User Characteristics

Describe those general characteristics of the intended users of the product including educational level, experience, and technical expertise. Do not state specific requirements but rather provide the reasons why certain specific requirements are later specified in sections 3.

2.4 Constraints

Provide a general description of any other items that will limit the developer's options. These can include:

- (i) Regulatory policies
- (ii) Hardware limitations (for example, signal timing requirements)
- (iii) Interface to other applications
- (iv) Parallel operation
- (v) Audit functions
- (vi) Control functions
- (vii) Higher-order language requirements
- (viii) Signal handshake protocols (for example, XON-XOFF, ACK-NACK)
- (ix) Reliability requirements
- (x) Criticality of the application
- (xi) Safety and security considerations.

2.5 Assumptions and Dependencies

List each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operating system would be available on the hardware designated for the software product. If, in fact, the operating system were not available, the SRS would then have to change accordingly.

2.6 Apportioning of Requirements

Identify requirements that may be delayed until future versions of the system.

3. Specific Requirements

This section contains all the software requirements at a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements. Throughout this section, every stated requirement should be externally perceivable by users, operators, or both external systems. These requirements should include at a minimum a description of every input into the system, every output from the system and all functions performed by the system in response to an input or in support of an output. The following principles apply:

(i) Specific requirements should be stated with all the characteristics of a good SRS

- correct
- unambiguous
- complete
- consistent
- ranked for importance and/or stability
- verifiable
- modifiable
- traceable

(ii) Specific requirements should be cross-referenced to earlier documents that relate

(iii) All requirements should be uniquely identifiable

(iv) Careful attention should be given to organizing the requirements to maximize readability.

Before examining specific ways of organizing the requirements it is helpful to understand the various items that comprise requirements as described in the following subsections.

3.1 External Interfaces

This contains a detailed description of all inputs into and outputs from the software system. It complements the interface descriptions in *section 2* but does not repeat information there.

It contains both content and format as follows:

- Name of item
- Description of purpose
- Source of input or destination of output
- Valid range, accuracy and/or tolerance
- Units of measure

- Timing
- Relationships to other inputs/outputs
- Screen formats/organization
- Window formats/organization
- Data formats
- Command formats
- End messages.

3.2 Functions

Functional requirements define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as “shall” statements starting with “The system shall...”

These include:

- Validity checks on the inputs
- Exact sequence of operations
- Responses to abnormal situation, including
 - Overflow
 - Communication facilities
 - Error handling and recovery
- Effect of parameters
- Relationship of outputs to inputs, including
 - Input/Output sequences
 - Formulas for input to output conversion.

It may be appropriate to partition the functional requirements into sub-functions or sub-processes. This does not imply that the software design will also be partitioned that way.

3.3 Performance Requirements

This subsection specifies both the static and the dynamic numerical requirements placed on the software or on human interaction with the software, as a whole. 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

Static numerical requirements are sometimes identified under a separate section entitled capacity.

Dynamic numerical requirements may include, for example, the number of transactions and tasks and the amount of data to be processed within certain time periods for both normal and peak workload conditions.

All of these requirements should be stated in measurable terms:

For example,

95% of the transactions shall be processed in less than 1 second rather than,
An operator shall not have to wait for the transaction to complete.

(Note: Numerical limits applied to one specific function are normally specified as part of the processing subparagraph description of that function).

3.4 Logical Database Requirements

This section specifies the logical requirements for any information that is to be placed into a database. This may include:

- Types of information used by various functions
- Frequency of use
- Accessing capabilities
- Data entities and their relationships
- Integrity constraints
- Data retention requirements.

3.5 Design Constraints

Specify design constraints that can be imposed by other standards, hardware limitations, etc.

3.5.1 Standards Compliance

Specify the requirements derived from existing standards or regulations. They might include:

- (i) Report format
- (ii) Data naming
- (iii) Accounting procedures
- (iv) Audit Tracing

For example, this could specify the requirement for software to process activity. Such traces are needed for some applications to meet minimum regulatory or financial standard. An audit trace requirement may, for example, state that all changes to a payroll database must be recorded in a trace file with before and after values.

3.6 Software System Attributes

There are a number of quality attributes of software that can serve as requirements. It is important that required attributes be specified so that their achievement can be objectively verified. Fig. 3.19 has the definitions of the quality attributes of the software discussed in this subsection [ROBE02]. The following items provide a partial list of examples.

3.6.1 Reliability

Specify the factors required to establish the required reliability of the software system at time of delivery.

3.6.2 Availability

Specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery, and restart.

3.6.3 Security

Specify the factors that would protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to:

- Utilize certain cryptographic techniques
- Keep specific log or history data sets
- Assign certain functions to different modules
- Restrict communications between some areas of the program
- Check data integrity for critical variables.

3.6.4 Maintainability

Specify attributes of software that relate to the ease of maintenance of the software itself. There may be some requirement for certain modularity, interfaces, complexity, etc. Requirements should not be placed here just because they are thought to be good design practices.

3.6.5 Portability

Specify attributes of software that relate to the ease of parting the software to other host machines and/or operating systems. This may include:

- Percentage of components with host-dependent code
- Percentage of code that is host dependent
- Use of a proven portable language
- Use of a particular compiler or language subset
- Use of a particular operating system.

S. No.	Quality Attributes	Definition
1.	Correctness	extent to which program satisfies specifications, fulfills user's mission objectives
2.	Efficiency	amount of computing resources and code required to perform function
3.	Flexibility	effort needed to modify operational program
4.	Interoperability	effort needed to couple one system with another
5.	Reliability	extent to which program performs with required precision
6.	Reusability	extent to which it can be reused in another application
7.	Testability	effort needed to test to ensure performance as intended
8.	Usability	effort required to learn, operate, prepare input, and interpret output
9.	Maintainability	effort required to locate and fix an error during operation
10.	Portability	effort needed to transfer from one hardware or software environment to another.
11.	Integrity/security	extent to which access to software or data by unauthorised people can be controlled.

Fig. 3.19: Definitions of quality attributes.

3.7 Organizing the Specific Requirements

For anything but trivial systems the detailed requirements tend to be extensive. For this reason, it is recommended that careful consideration be given to organizing these in a manner optimal for understanding. There is no one optimal organization for all systems. Different classes of systems lend themselves to different organizations of requirements. Some of these organizations are described in the following subclasses.

3.7.1 System Mode

Some systems behave quite differently depending on the mode of operation. When organizing by mode there are two possible outlines. The choice depends on whether interfaces and performance are dependent on mode.

3.7.2 User Class

Some systems provide different sets of functions to different classes of users.

3.7.3 Objects

Objects are real-world entities that have a counterpart within the system. Associated with each object is a set of attributes and functions. These functions are also called services, methods, or processes. Note that sets of objects may share attributes and services. These are grouped together as classes.

3.7.4 Feature

A feature is an externally desired service by the system that may require a sequence of inputs to effect the desired result. Each feature is generally described as sequence of stimulus-response pairs.

3.7.5 Stimulus

Some systems can be best organized by describing their functions in terms of stimuli.

3.7.6 Response

Some systems can be best organized by describing their functions in support of the generation of a response.

3.7.7 Functional Hierarchy

When none of the above organizational schemes prove helpful, the overall functionality can be organized into a hierarchy of functions organized by either common inputs, common outputs, or common internal data access. Data flow diagrams and data dictionaries can be used to show the relationships between and among the functions and data.

3.8 Additional Comments

Whenever a new SRS is contemplated, more than one of the organizational techniques given in 3.7 may be appropriate. In such cases, organize the specific requirements for multiple hierarchies tailored to the specific needs of the system under specification.

There are many notations, methods, and automated support tools available to aid in the documentation of requirements. For the most part, their usefulness is a function of organization. For example, when organizing by mode, finite state machines or state charts may prove helpful; when organizing by object, object-oriented analysis may prove helpful; when organizing by feature, stimulus-response sequences may prove helpful; when organizing by functional hierarchy, data flow diagrams and data dictionaries may prove helpful.

In any of the outlines below, those sections called “Functional Requirement i” may be described in native language, in pseudocode, in a system definition language, or in four subsections titled: Introduction, Inputs, Processing, Outputs.

4. Change Management Process

Identify the change management process to be used to identify, log, evaluate, and update the SRS to reflect changes in project scope and requirements.

5. Document Approval

Identify the approvers of the SRS document. Approver’s name, signature, and date should be used.

6. Supporting Information

The supporting information makes the SRS easier to use. It includes:

- Table of Contents
- Index
- Appendices

The Appendices are not always considered part of the actual requirements specification and are not always necessary. They may include:

- (a) Sample I/O formats, descriptions of cost analysis studies, results of user surveys
- (b) Supporting or background information that can help the readers of the SRS
- (c) A description of the problems to be solved by the software
- (d) Special packaging instructions for the code and the media to meet security, export, initial loading, or other requirements.

When Appendices are included, the SRS should explicitly state whether or not the Appendices are to be considered part of the requirements.

Tables on the following pages provide alternate ways to structure section 3 on the specific requirements.

Outline for SRS Section 3 Organized by Mode: Version 1

- 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 requirements 1.1**
 - 3.2.1.n Functional requirements 1.n**
 - 3.2.2 Mode 2**
 - 3.2.m Mode 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**

Outline for SRS Section 3 Organized by Mode: Version 2

- 3 Specific Requirements**
 - 3.1 Functional Requirements**
 - 3.1.1 Mode 1**
 - 3.1.1.1 External interfaces**
 - 3.1.1.1 User Interfaces**
 - 3.1.1.2 Hardware interfaces**
 - 3.1.1.3 Software interfaces**
 - 3.1.1.4 Communications interfaces**
 - 3.1.1.2 Functional Requirement**
 - 3.1.1.2.1 Functional requirement 1**
 - 3.1.1.2.1.n Functional requirement n**
 - 3.1.1.3 Performance**
 - 3.1.2 Mode 2**
 - 3.1.m Mode m**
 - 3.2 Design constraints**
 - 3.3 Software system attributes**
 - 3.4 Other requirements.**

Outline for 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 requirements 1.1
 - 3.2.1.*n* Functional requirement 1.*n*
 - 3.2.2 User class 2
 - 3.2.*m* User class *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

Outline for 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.*n* Attribute *n*
 - 3.2.1.2 Functions (services, methods, direct or inherited)
 - 3.2.1.2.1 Functional requirement 1.1
 - 3.2.1.2.*m* Functional requirement 1.*m*
 - 3.2.1.3 Messages (communications received or sent)
 - 3.2.2 Class/Object 2
 - 3.2.*p* Class/Object *p*
 - 3.3 Performance Requirements
 - 3.4 Design Constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

Outline for 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.n Functional requirement n
 - 3.2.2 System Feature 2
 - 3.2.m System Feature m
 - 3.3 Performance Requirements
 - 3.4 Design Constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

Outline for 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
 - 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

Outline for SRS Section 3 Organized by Response

- 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 Response 1
 - 3.2.1.1 Functional requirement 1.1
 - 3.2.1.*n* Functional requirement 1.*n*
 - 3.2.2 Response 2
 - 3.2.*m* Response *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

Outline for 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
 - 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.*n* Data flow diagram *n*
 - 3.2.1.*n*.1 Data entities
 - 3.2.1.*n*.2 Pertinent processes
 - 3.2.1.*n*.3 Topology

(Contd)...

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.m Process m

- 3.2.2.m.1 Input data entities
- 3.2.2.m.2 Algorithm or formula of process
- 3.2.2.m.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.p Construct p

- 3.2.3.p.1 Record type
- 3.2.3.p.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
- 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**

Outline for SRS Section 3 Showing Multiple Organizations

- 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 Feature 1.1
 - 3.2.1.1.1 Introduction/Purpose of feature
 - 3.2.1.1.2 Stimulus/Response sequence
 - 3.2.1.1.3 Associated functional requirements
 - 3.2.1.2 Feature 1.2
 - 3.2.1.2.1 Introduction/Purpose of feature
 - 3.2.1.2.2 Stimulus/Response sequence
 - 3.2.1.2.3 Associated functional requirements
 - 3.2.1.m Feature 1.m
 - 3.2.1.m.1 Introduction/Purpose of feature
 - 3.2.1.m.2 Stimulus/Response sequence
 - 3.2.1.m.3 Associated functional requirements
 - 3.2.2 User class 2
 - 3.2.n User class n
 - 3.3 Performance Requirements
 - 3.4 Design Constraints
 - 3.5 Software system attributes
 - 3.6 Other requirements

3.5 STUDENT RESULT MANAGEMENT SYSTEM—EXAMPLE

A university has decided to engage a software company for the automation of student result management system of its M. Tech. Programme. The following documents are required to be prepared.

- (i) Problem statement
- (ii) Context diagram
- (iii) Data flow diagrams
- (iv) ER diagrams
- (v) Use case diagram
- (vi) Use cases
- (vii) SRS as per IEEE std. 830-1993

These seven documents may provide holistic view of the system to be developed. The SRS will act as contract document between developers (software company) and client (University).

3.5.1 Problem Statement

The problem statement is the first document which is normally prepared by the client. It only, gives superficial view of the system as per client's perspective and expectations. it is the input to the requirement engineering process where final product is the SRS.

The problem statement of student result management system of M. Tech. (Information Technology) Programme of a University is given below:

"A University conducts a 4-semester M. Tech. (IT) program. The students are offered four theory papers and two Lab papers (practicals) during Ist, IIInd and IIIrd semesters. The theory papers offered in these semesters are categorized as either 'Core' or 'Elective'. Core papers do not have an alternative subject, whereas elective papers have two other alternative subjects. Thus, a student can study any subject out of the 3 choices available for an elective paper.

In Ist, IIInd and IIIrd semesters, 2 core papers and 2 elective papers are offered to each student. The students are also required to submit a term paper/minor project in IIInd and IIIrd semesters each. In IVth semester the students have to give a seminar and submit a dissertation on a topic/subject area of their interest.

The evaluation of each subject is done out of 100 marks. During the semester, minor exams are conducted for each semester. Students are also required to submit assignments as directed by the corresponding faculty and maintain Lab records for practicals. Based on the students' performance in minor exams, assignments, Lab records and their attendance, marks out of 40 are given in each subject and practical paper. These marks out of 40 account for internal evaluation of the students. At the end of each semester major exams are conducted in each subject (theory as well as practical). These exams are evaluated out of 60 marks and account for external evaluation of the students. Thus, the total marks of a student in a subject are obtained by adding the marks obtained in internal and external evaluation.

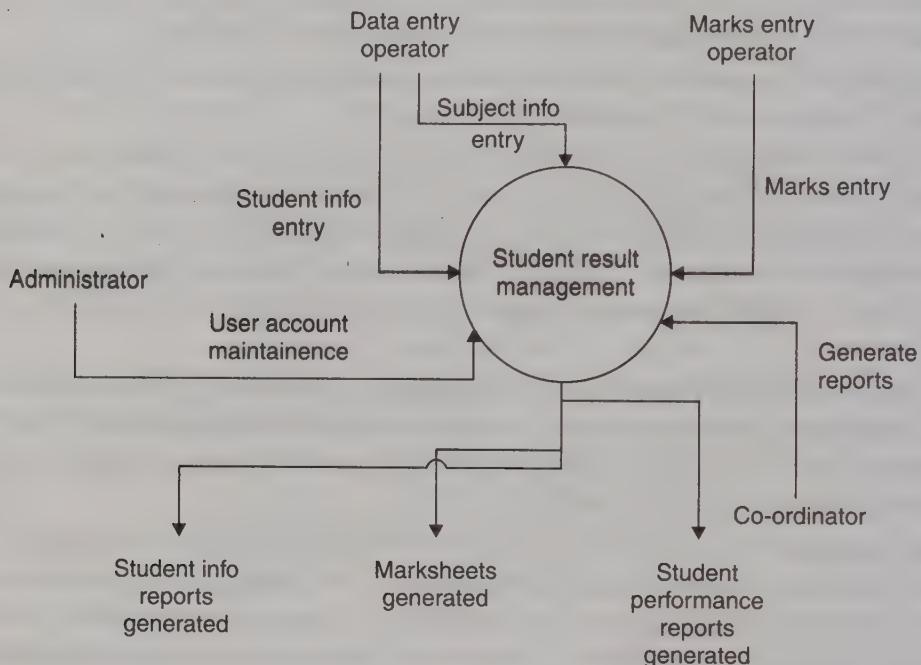
Every subject has some credit points assigned to it. If the total marks of a student are $>= 50$ in a subject, he/she is considered 'Pass' in that subject otherwise the student is considered 'Fail' in that subject. If a student passes in a subject, he/she earns all the credit points assigned to that subject, but if the student fails in a subject he/she does not earn any credit point in that subject. At any time, the latest information about subjects being offered in various semesters and their credit points can be obtained from university's website.

It is required to develop a system that will manage information about subjects offered in various semesters, students enrolled in various semesters, elective(s) opted by various students in different semesters, marks and credit points obtained by students in different semesters.

The system should also have the ability to generate printable mark sheets for each student. Semester-wise detailed mark lists and student performance reports also need to be generated."

3.5.2 Context Diagram

The context diagram is given below:



The following persons are interacting with the "student result management system"

- (i) Administrator
- (ii) Marks entry operator
- (iii) Data entry operator
- (iv) Co-ordinator

3.5.3 Level-n DFD

Level-1 DFD

The Level-1 DFD is given below:

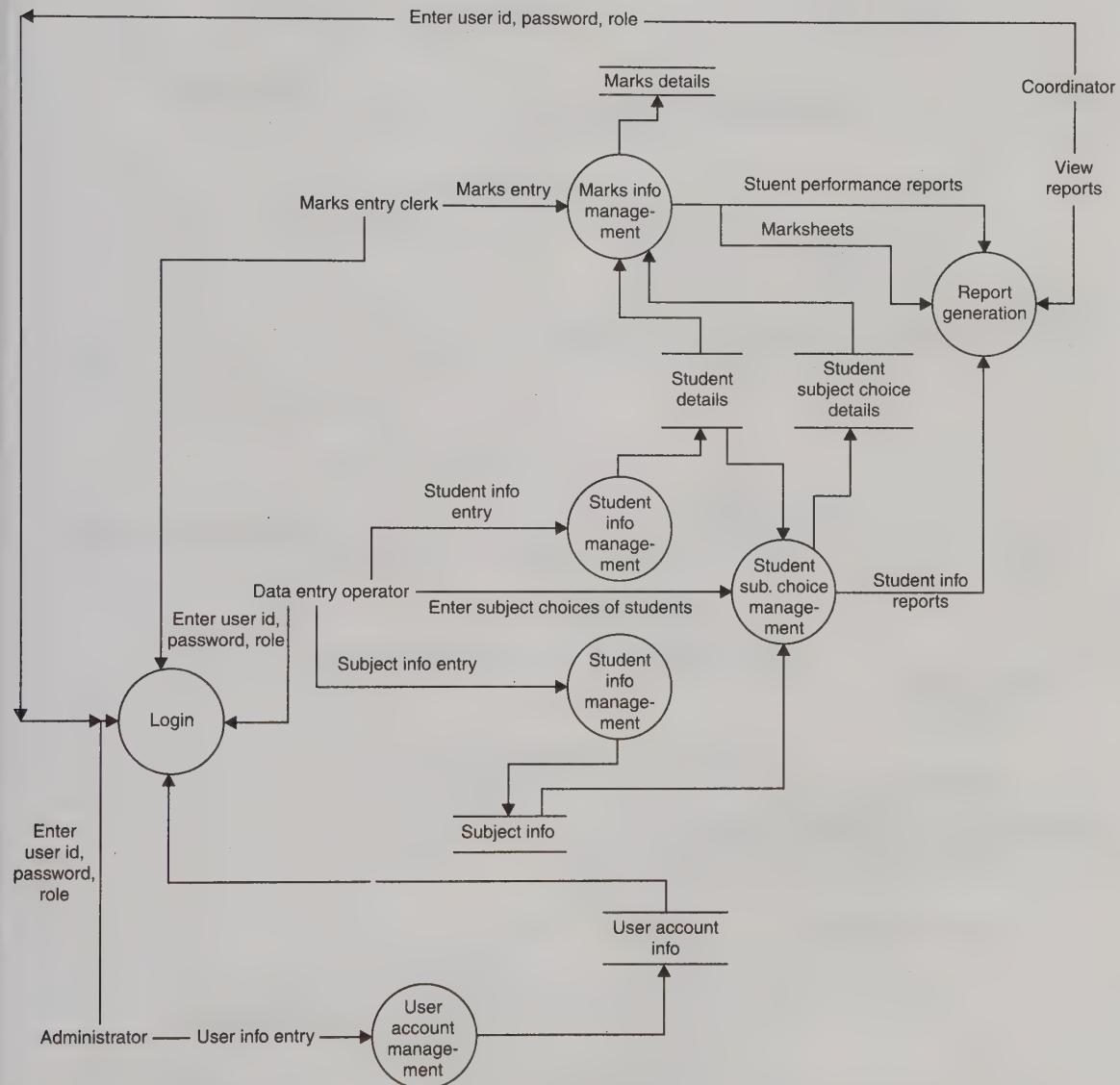
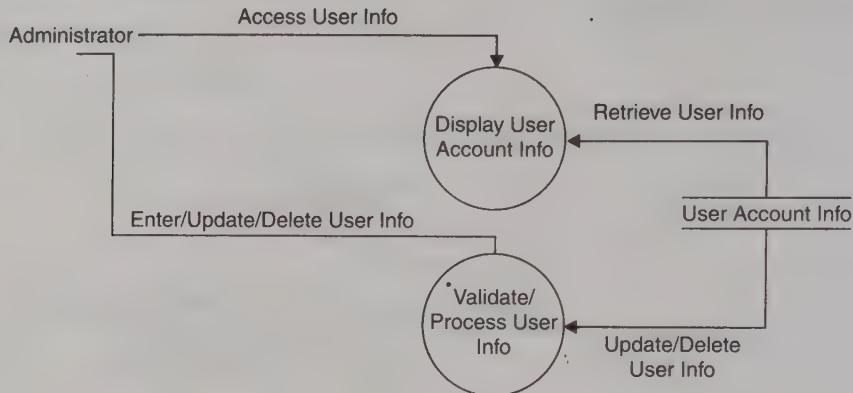


Fig. 3.8: Level 1 DFD of result management system.

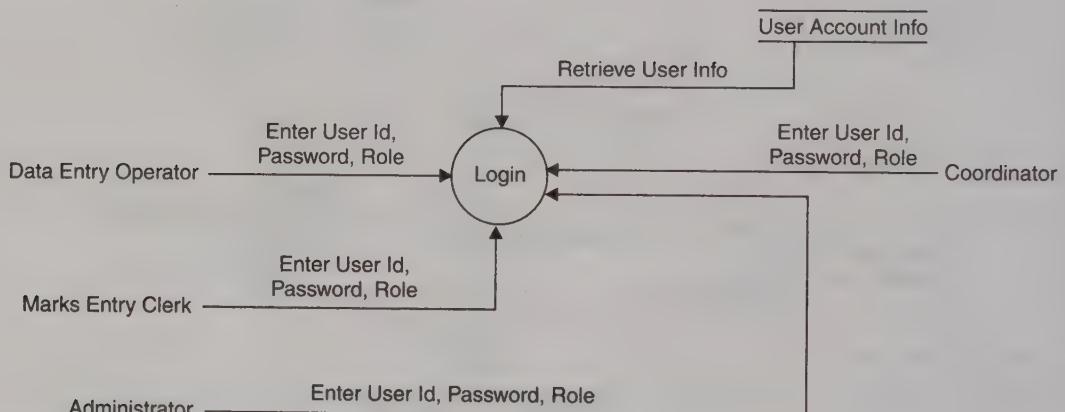
Level-2 DFDs

1. User Account Maintenance



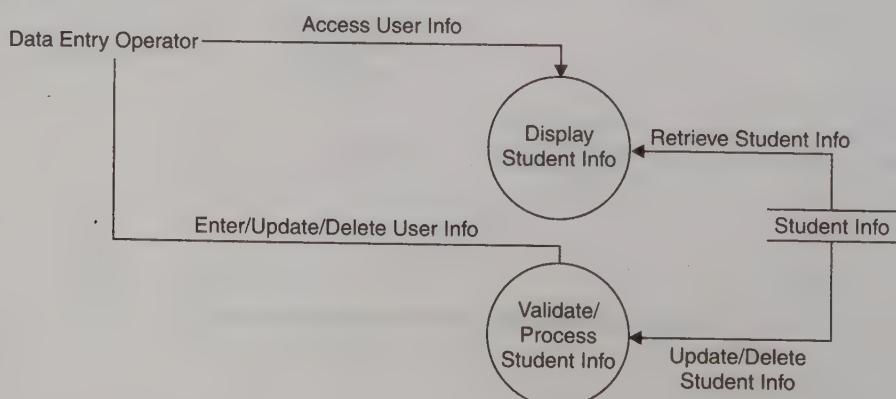
2. Login

The Level 2 DFD of this process is given below:



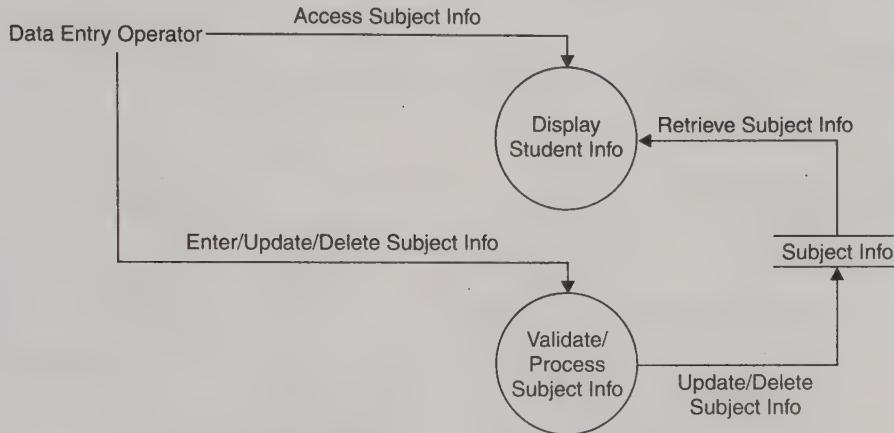
3. Student Information Management

The Level 2 DFD of this process is given below:



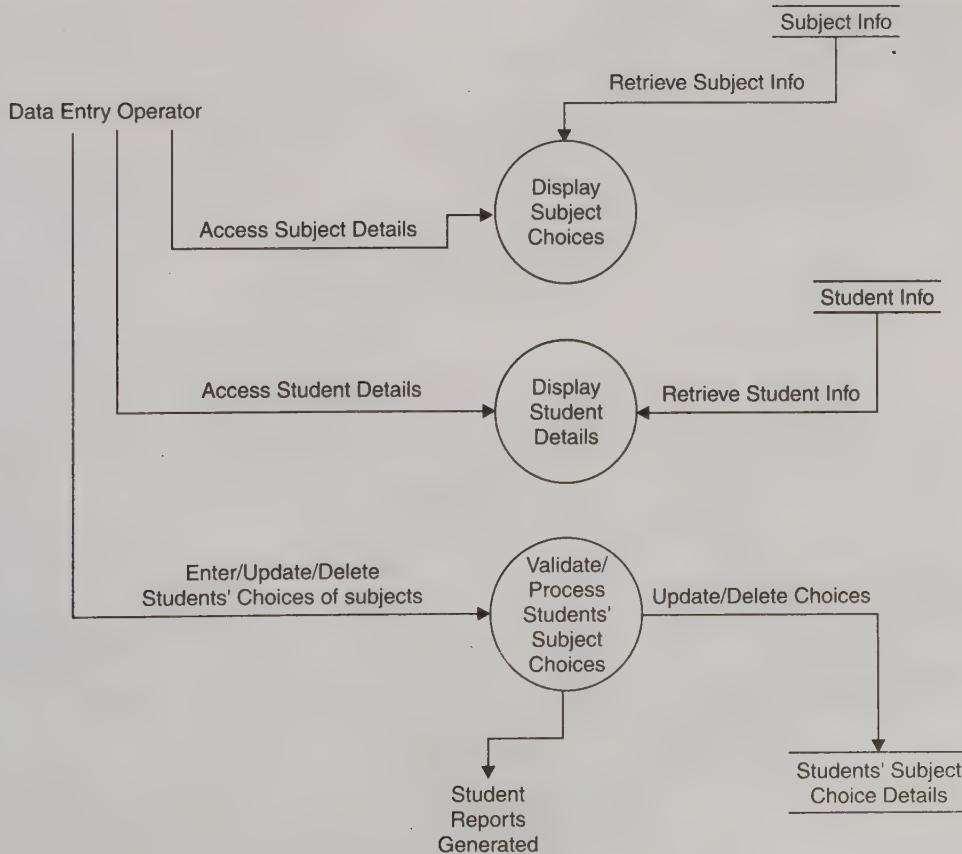
4. Subject Information Management

The Level 2 DFD of this process is given below:



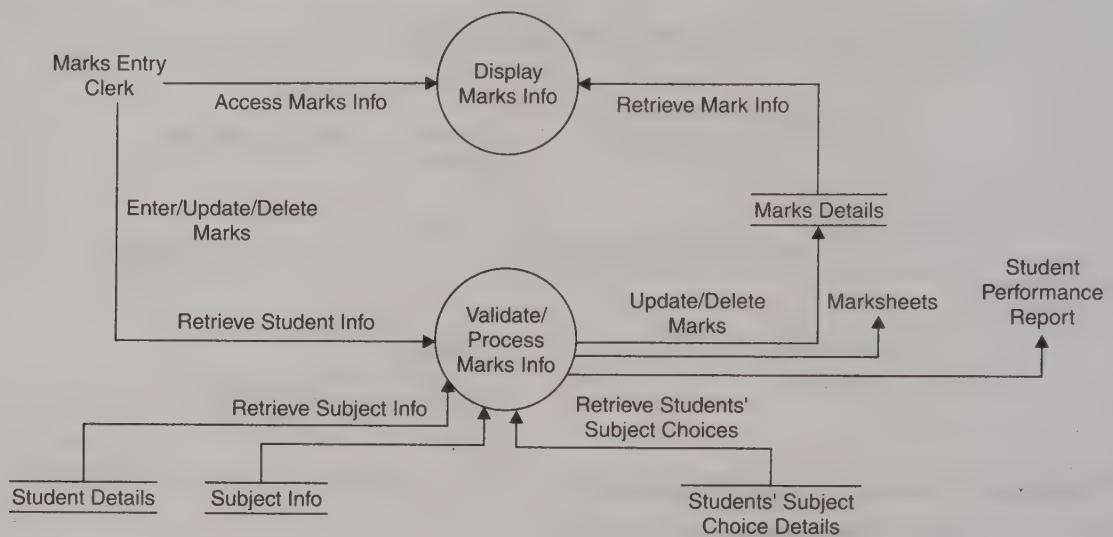
5. Students' Subject Choice Management

The Level 2 DFD of this process is given below:



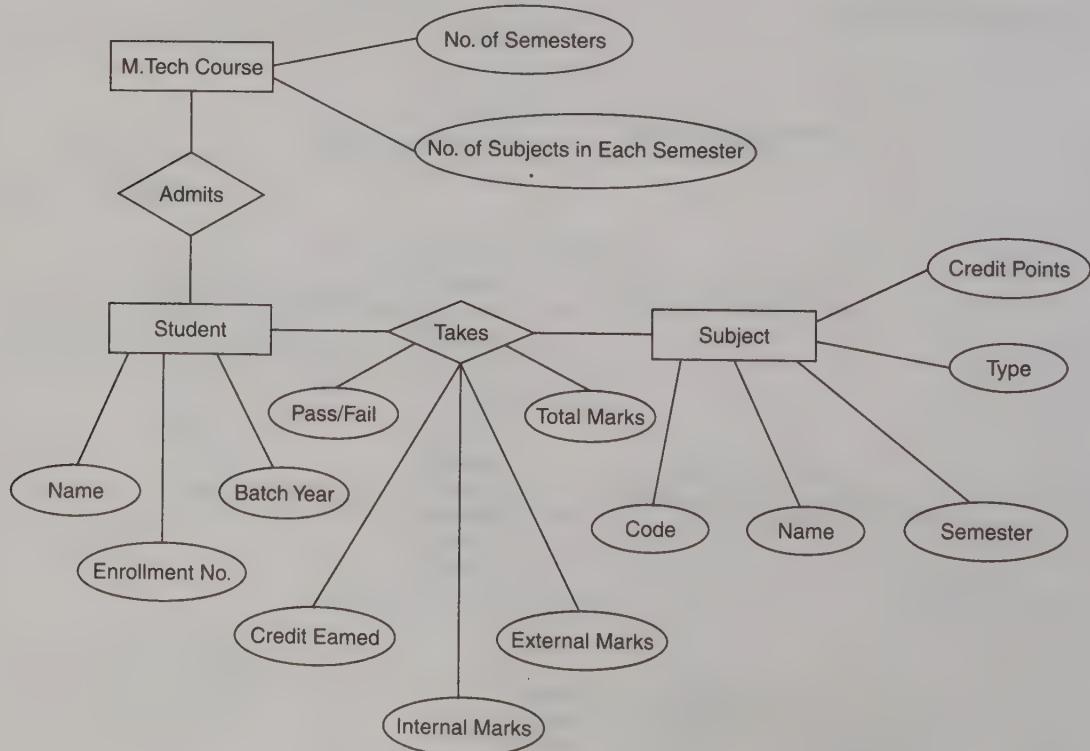
6. Marks Information Management

The Level 2 DFD of this process is given below:

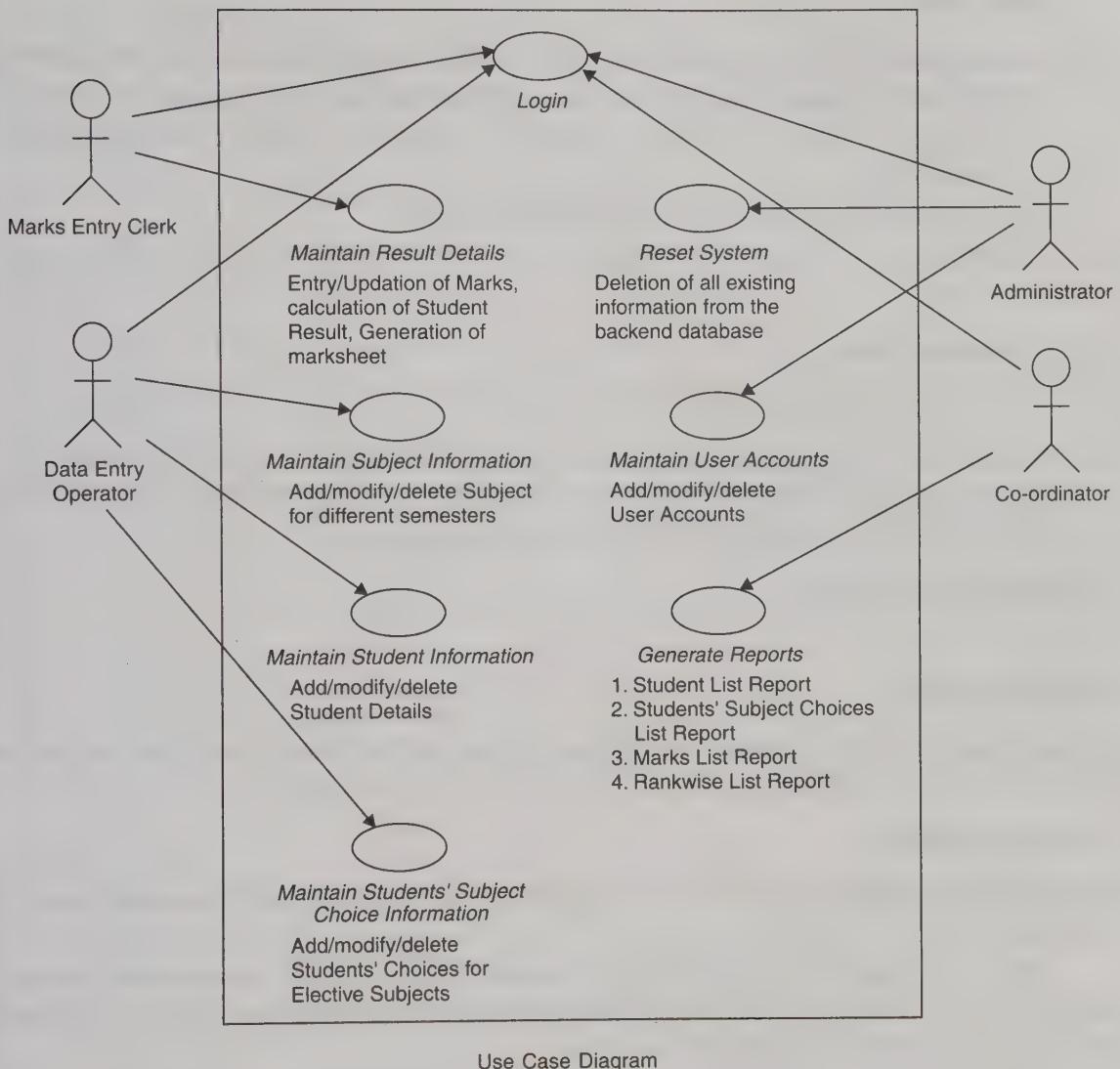


3.5.4 Entity Relationship Diagram

The ER diagram of the system is given below:



3.5.5 Use Case Diagram



Use Case Diagram

3.5.6 Use Cases

1 Login

1.1 Brief Description

This use case describes how a user logs into the Student Result Management System.

1.2 Actors

The following actor(s) interact and participate in this use case:

Data Entry Operator, Marks Entry Clerk, Administrator, Coordinator.

1.3 Flow of Events

1.3.1 Basic Flow

This use case starts when the actor wishes to Login to the Student Result Management System.

1. The system requests that the actor enter his/her name, password and role. The role can be any one of Data Entry Operator, Marks Entry Clerk, Coordinator, and Administrator.
2. The actor enters his/her name, password and role.
3. The system validates the entered name, password, role and logs the actor into the system.

1.3.2 Alternative Flows

1.3.2.1 Invalid Name/Password/Role

If in the Basic Flow, the actor enters an invalid name, password and/or role, the system displays an error message. The actor can choose to either return to the begining of the Basic Flow or cancel the login, at which point the use case ends.

1.4 Special Requirements

None

1.5 Pre-Conditions

All users must have a User Account (*i.e.*, User ID, Password and Role) created for them in the system (through the Administrator), prior to executing the use cases.

1.6 Post-Conditions

If the use case was successful, the actor is logged into the system. If not, the system state is unchanged.

If the actor has the role ‘Data Entry Operator’ he/she will have access to only screens corresponding to the Subject Info Maintenance, Student Info Maintenance and Students’ Subject Choice Info Maintenance modules of the system.

If the actor has the role ‘Marks Entrey Clerk’, he/she will have access to only screens corresponding to the Marks Info Maintenance module of the system. If the actor has the role ‘Coordinator’, he/she will only be able to view/print the various reports generated by the system.

If the actor has the role ‘Administrator’ he/she will have access to only screens corresponding to User Account maintenance module and Reset System feature of the system.

1.7 Extension Points

None

2 Maintain Student Information

2.1 Brief Description

This use case allows the actor with role ‘Data Entry Operator’ to maintain student information. This includes adding, changing and deleting student information from the system.

2.2 Actors

The following actor(s) interact and participate in this use case:

Data Entry Operator.

2.3 Flow of Events

2.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to add, change, and/or delete student information from the system.

1. The system requests that the Data Entry Operator specify the function he/she would like to perform (either Add a Student, Update a Student, or Delete a Student).
2. Once the Data Entry Operator provides the requested information, one of the sub-flows is executed.
 - If the Data Entry Operator selected “Add a Student”, the **Add a Student** sub-flow is executed.
 - If the Data Entry Operator selected “Update a Student”, the **Update a Student** sub-flow is executed.
 - If the Data Entry Operator selected “Delete a Student”, the **Delete a Student** sub-flow is executed.

2.3.1.1 Add a Student

1. The system requests that the Data Entry Operator enter the student information. This includes:
 - (a) Name
 - (b) Enrollment Number—should be unique for every student
 - (c) Year of Enrollment
2. Once the Data Entry Operator provides the requested information, the student is added to the system and an appropriate message is displayed.

2.3.1.2 Update a Student

1. The system requests that the Data Entry Operator enters the student enrollment number.
2. The Data Entry Operator enters the student enrollment number. The system retrieves and displays the student information.
3. The Data Entry Operator makes the desired changes to the student information. This includes any of the information specified in the **Add a Student** sub-flow.
4. Once the Data Entry Operator updates the necessary information, the system updates the student record with the updated information.

2.3.1.3 Delete a Student

1. The system requests that the Data Entry Operator enters the student enrollment number.
2. The Data Entry Operator enters the student enrollment number. The system retrieves and displays the student information.
3. The system prompts the Data Entry Operator to confirm the deletion of the student.
4. The Data Entry Operator confirms the deletion.
5. The system deletes the student record.

2.3.2 Alternative Flows

2.3.2.1 Student Not Found

If in the **Update a Student** or **Delete a Student** sub-flows, a student with the specified enrollment number does not exist, the system displays an error message. The Data Entry Operator can then enter a different enrollment number or cancel the operation, at which point the use case ends.

2.3.2.2 Update Cancelled

If in the **Update a Student** sub-flow, the Data Entry Operator decides not to update the student information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

2.3.2.3 Delete Cancelled

If in the **Delete a Student** sub-flow, the Data Entry Operator decides not to delete the student information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

2.4 Special Requirements

None

2.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

2.6 Post-Conditions

If the use case was successful, the student information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

2.7 Extension Points

None

3 Maintain Subject Information

3.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain subject information. This includes adding, changing and deleting subject information from the system.

3.2 Actors

The following actor(s) interact and participate in this use case:

Data Entry Operator

3.3 Flow of Events

3.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to add, change, and/or delete subject information from the system.

1. The system requests that the Data Entry Operator specify the function he/she would like to perform (either Add a Subject, Update a Subject, or Delete a Subject).
2. Once the Data Entry Operator provides the requested information, one of the sub-flows is executed.
 - If the Data Entry Operator selected “Add a Subject”, the **Add a Subject** sub-flow is executed.
 - If the Data Entry Operator selected “Update a Subject”, the **Update a Subject** sub-flow is executed.
 - If the Data Entry Operator selected “Delete a Subject”, the **Delete a Subject** sub-flow is executed.

3.2.1.1 Add a Subject

1. The system requests that the Data Entry Operator enters the subject information. This includes:
 - (a) Name of the subject
 - (b) Subject Code—should be unique for every subject
 - (c) Semester
 - (d) Subject Type—can be Core 1/Core 2/Dissertation/Elective 1/Elective 2/Lab 1/Lab 2/Minor Project/Seminar/Term Paper.
 - (e) Credits.
2. Once the Data Entry Operator provides the requested information, the subject is added to the system and an appropriate message is displayed.

3.3.1.2 Update a Subject

1. The system requests that the Data Entry Operator enters the subject code.
2. The Data Entry Operator enters the subject code. The system retrieves and displays the subject information.
3. The Data Entry Operator makes the desired changes to the subject information. This includes any of the information specified in the **Add a Subject** sub-flow.
4. Once the Data Entry Operator updates the necessary information, the system updates the subject record with the updated information.

3.3.1.3 Delete a Subject

1. The system requests that the Data Entry Operator enter the subject code.
2. The Data Entry Operator enters the subjects code. The system retrieves and displays the subject information.
3. The system prompts the Data Entry operator to confirm the deletion of the subject.
4. The Date Entry Operator confirms the deletion.
5. The system deletes the subject record.

3.3.2 Alternative Flows

3.3.2.1 Subject Not Found

If in the **Update a Subject** or **Delete a Subject** sub-flows, a subject with the specified subject code does not exist, the system displays an error message. The Data Entry Operator can then enter a different subject code or cancel the operation, at which point the use case ends.

3.3.2.2 Update Cancelled

If in the **Update a Subject** sub-flow, the Data Entry Operator decides not to update the subject information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

3.3.2.3 Delete Cancelled

If in the **Delete a Subject** sub-flow, the Data Entry Operator decides not to delete the subject information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

3.4 Special Requirements

None

3.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

3.6 Post-Conditions

If the use case was successful, the student information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

3.7 Extension Points

None

4 Maintain Students' Subject Choice Information

4.1 Brief Description

This use case allows the actor with role 'Data Entry Operator' to maintain information about the choice of different Elective subjects opted by various students. This includes displaying the various available choices of Elective subjects available during a particular semester and updating the information about the choice of Elective Subject(s) opted by different students of that semester.

4.2 Actors

The following actor(s) interact and participate in this use case:

Data Entry Operator

4.3 Flow of Events

4.3.1 Basic Flow

This use case starts when the Data Entry Operator wishes to update students' Subject Choice information from the system.

1. The system requests that the Data Entry Operator specify the semester and enrollment year of students, for which the Students' Subject Choices have to be updated.
2. Once the Data Entry Operator provides the requested information, the system displays the list of available choices for Elective I and Elective II subjects for that semester and the list of students enrolled in the given enrollment year (along with their existing subject choices, if any).
3. The system requests that the Data Entry Operator specify the information regarding Students' Subject Choices. this includes
 - (a) Student's Enrollment Number
 - (b) Student's Choice for Elective I subject (the corresponding subject code)
 - (c) Student's Choice for Elective II subject (the corresponding subject code).
4. Once the Data Entry Operator provides the requested information, the information regarding Student's Subject Choices is added/updated in the system and an appropriate message is displayed.

4.3.2 Alternative Flows

4.3.2.1 Subject Information Does Not Exist

If no or incomplete subject information exists in the system for the semester specified by the Data Entry Operator, the system displays an error message. The Data Entry Operator can then enter a different semester or cancel the operation, at which point the use case ends.

4.3.2.2 Student Information Does Not Exist

If no student information exists in the system for the enrollment year specified by the Data entry Operator, the system displays an error message. The Data Entry Operator can then enter a different enrollment year or cancel the operation, at which point the use case ends.

4.3.2.3 Incorrect Choice Entered for Elective I/Elective II Subjects

If the subject code entered by the Data Entry Operator for Elective I/Elective II subject does not exist in the system, the system displays an error message.

The Data Entry Operator can then enter the correct subject code or cancel the operation, at which point the use case ends.

4.3.2.4 Update Cancelled

If in the **Basic Flow**, the Data Entry Operator decides not to update the subject information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

4.4 Special Requirements

None

4.5 Pre-Conditions

The Data Entry Operator must be logged onto the system before this use case begins.

4.6 Post-Conditions

If the use case was successful, information about students' choices for opting different Elective Subjects is added/updated in the system. Otherwise, the system state is unchanged.

4.7 Extension Points

None

5 Maintain Result Details

5.1 Brief Description

This use case allows the actor with role 'Marks Entry Clerk' to maintain subject-wise marks information of each student, in different semesters. This includes adding, changing and deleting marks information from the system.

5.2 Actors

The following actor(s) interact and participate in this use case:

Marks Entry Clerk.

5.3 Flow of Events

5.3.1 Basic Flow

This use case starts when the Marks Entry Clerk wishes to add, change, and/or delete marks information from the system.

1. The system requests that the Marks Entry Clerk specify the function he/she would like to perform (either Add Marks, Update Marks, Delete Marks, or Generate Mark sheet).
2. Once the Marks Entry Clerk provides the requested information, one of the sub-flows is executed.
 - If the Marks Entry Clerk selected "Add Marks", the **Add Marks** sub-flow is executed.
 - If the Marks Entry Clerk selected "Update Marks", the **Update Marks** sub-flow is executed.
 - If the Marks Entry Clerk selected "Delete Marks", the **Delete Marks** sub-flow is executed.
 - If the Marks Entry Clerk selected "Generate Mark sheet", the **Generate Mark sheet** sub-flow is executed.

5.3.1.1 Add Marks Record

1. The system requests that the Marks Entry Clerk enters the marks information. This includes:

- (a) Selecting a semester
 - (b) Selecting a Subject Code
 - (c) Selecting the student enrollment number
 - (d) Entering the internal/external marks for that semester, subject code and enrollment number.
2. Once the Marks Entry Clerk provides the requested information, the system saves the marks and an appropriate message is displayed.
- 5.3.1.2 Update Marks Record**
1. The system requests the Marks Entry Clerk to make following entries:
 - (a) Selecting the semester
 - (b) Selecting the subject code for which marks have to be updated
 - (c) Selecting the student enrollment number.
 2. Once the Marks Entry Clerk provides the requested information, the system retrieves and displays the corresponding marks details.
 3. The Marks Entry Clerk makes the desired changes to the internal/external marks details.
 4. The system updates the marks record with the changed information.

5.3.1.3 Delete Marks Record

1. The system requests the Marks Entry Clerk to make following entries:
 - (a) Selecting the semester
 - (b) Selecting the subject code for which marks have to be updated
 - (c) Selecting the student enrollment number.
2. Once the Marks Entry Clerk provides the requested information, the system retrieves and displays the corresponding marks record from the database.
3. The system verifies if the Marks Entry Clerk wishes to proceed with the deletion of the record. Upon confirmation, the record is deleted from the system.

5.3.1.4 Compute Result

1. Once all the marks are added to the database, the result is computed for each student.
2. If the student has scored more than 50% in a subject, the associated credit points are allotted to that student.
3. The average percentage marks are calculated for the student and his/her division is also derived based on the percentage.

5.3.1.5 Generate Mark Sheet

1. The system requests that the Marks Entry Clerk specify the Enrollment Number of the student and the semester for which mark sheet is to be generated.
2. Once the Marks Entry Clerk provides the requested information, the system generates a printable mark sheet for the specified student and displays it.
3. The Marks Entry Clerk can then issue a print request for the mark sheet to be printed.

5.3.2 Alternative Flows

5.3.2.1 Record Not Found

If in the **Update Marks**, **Delete Marks** or **Generate Mark sheet** sub-flows, a record with the specified information does not exist, the system displays an error message. The Marks Entry Clerk can then enter different information for retrieving the record or cancel the operation, at which point the use case ends.

5.3.2.2 Update Cancelled

If in the **Update Marks** sub-flow, the Marks Entry Clerk decides not to update the marks, the update is cancelled and the **Basic Flow** is re-started at the beginning.

5.3.2.3 Delete Cancelled

If in the **Delete Marks** sub-flow, the Marks Entry Clerk decides not to delete the marks, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

5.4 Special Requirements

None

5.5 Pre-Conditions

The Marks Entry Clerk must be logged onto the system before this use case begins.

5.6 Post-Conditions

If the use case was successful, the marks information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

5.7 Extension Points

None

6 Generate Reports

6.1 Brief Description

This use case allows the actor with role ‘Coordinator’ to generate various reports. The following reports can be generated:

- (a) Student List Report
- (b) Students’ Subject Choices List Report
- (c) Marks List Report
- (d) Rank-wise List Report

6.2 Actors

The following actor(s) interact and participate in this use case:

Coordinator

6.3 Flow of Events

6.3.1 Basic Flow

This use case starts when the Coordinator wishes to generate reports.

1. The system requests the Coordinator specify the report he/she would like to generate.
2. Once the Coordinator provides the requested information, one of the sub-flows is executed:
 - If the Coordinator selected “Student List Report”, the **Generate Student List Report** sub-flow is executed.
 - If the Coordinator selected “Students’ Subject Choices List Report”, the **Generate Students’ Subject Choices List Report** sub-flow is executed.
 - If the Coordinator selected “Marks List Report”, the **Generate Marks List Report** sub-flow is executed.
 - If the Coordinator selected “Rank-wise List Report”, the **Generate Rank-wise List Report** sub-flow is executed.

6.3.1.1 Generate Student List Report

1. The system requests that the Coordinator provide the enrollment year for which the Student List report is to be generated.
2. Once the Coordinator provides the requested information, the system generates the Student List report, containing the list of students enrolled in the given year.
3. The Coordinator can then issue a print request for the report to be printed.

6.3.1.2 Generate Student’s Subject Choices List Report

1. The system requests that the Coordinator provides the enrollment year and the semester for which the Students’ Subject Choices List report is to be generated.
2. Once the Coordinator provides the requested information, the system generates the Students’ subject Choices List report, containing the choices for Elective I and Elective II subjects, opted by the students of the given enrollment year and semester.
3. The Coordinator can then issue a print request for the report to be printed.

6.3.1.3 Generate Marks List Report

1. The system requests that the Coordinator provides the enrollment year and the semester for which the Marks List report is to be generated.
2. Once the Coordinator provides the requested information, the system generates the Marks List report, containing the marks details of various students in all the subjects for the given enrollment year and semester.
3. The Coordinator can then issue a print request for the report to be printed.

6.3.1.4 Generate Rank-wise List Report

1. The system requests that the Coordinator provide the enrollment year and the semester for which the Rank-wise List report is to be generated.

2. Once the Coordinator provides the requested information, the system generates the Rank-wise List report, containing the percentage wise and rank-wise list of all students (along with their total marks and division) for the given enrollment year and semester.
3. The Coordinator can then issue a print request for the report to be printed.

6.3.2 Alternative Flows

6.3.2.1 Student Not Found

If no student information exists in the system for the enrollment year specified by the Coordinator, the system displays an error message. The Coordinator can then enter a different enrollment year or cancel the operation, at which point the use case ends.

6.4 Special Requirements

None

6.5 Pre-Conditions

The Coordinator must be logged onto the system before this use case begins.

6.6 Post-Conditions

If the use case was successful, the desired report is generated. Otherwise, the system state is unchanged.

6.7 Extension Points

None.

7 Maintain User Accounts

7.1 Brief Description

This use case allows the actor with role ‘Administrator’ to maintain User Account. This includes adding, changing and deleting user account information from the system.

7.2 Actors

The following actor(s) interact and participate in this use case:

Administrator.

7.3 Flow of Events

7.3.1 Basic Flow

This use case starts when the Administrator wishes to add, change, and/or delete use account information from the system.

1. The system requests that the Administrator specify the function he/she would like to perform (either Add a User Account, Update a User Account, or Delete a User Account).
2. Once the Administrator provides the requested information, one of the sub-flows is executed.

- If the Administrator selected “Add a User Account”, the **Add a User Account** sub-flow is executed.
- If the Administrator selected “Update a User Account”, the **Update a User Account** sub-flow is executed.
- If the Administrator selected “Delete a User Account”, the **Delete a User Account** sub-flow is executed.

7.3.1.1 Add a User Account

1. The system requests that the Administrator enters the user information. This includes:
 - (a) User Name
 - (b) User ID-should be unique for each user account
 - (c) Password
 - (d) Role
2. Once the Administrator provides the requested information, the user account information is added to the system and an appropriate message is displayed.

7.3.1.2 Update a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The Administrator makes the desired changes to the user account information. This includes any of the information specified in the **Add a User Account** sub-flow.
4. Once the Administrator updates the necessary information, the system updates the user account record with the updated information.

7.3.1.3 Delete a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The system prompts the Administrator to confirm the deletion of the user account.
4. The Administrator confirms the deletion.
5. The system deletes the user account record.

7.3.2 Alternative Flows

7.3.2.1 User Not Found

If in the **Update a User Account** or **Delete a User Account** sub-flows, a user account with the specified User ID does not exist, the system displays an error message. The Administrator can then enter a different User ID or cancel the operation, at which point the use case ends.

7.3.2.2 Update Cancelled

If in the **Update a User Account** sub-flow, the Administrator decides not to update the user account information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

7.3.2.3 Delete Cancelled

If in the **Delete a User Account** sub-flow, the Administrator decides not to delete the user account information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

7.4 Special Requirements

None .

7.5 Pre-Conditions

The Administrator must be logged onto the system before this use case begins.

7.6 Post-Conditions

If the use case was successful, the user account information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

7.7 Extension Points

None

8. Reset System

8.1 Brief Description

This Use case allows the actor with role 'Administrator' to reset the system by deleting all existing information from the system.

8.2 Actors

The following actor(s) interact and participate in this use case:

Administrator

8.3 Flow of Events

8.3.1 Basic Flow

This use case starts when the Administrator wishes to reset the system.

1. The system requests the Administrator to confirm if he/she wants to delete all the existing information from the system.
2. Once the Administrator provides confirmation, the system deletes all the existing information from the backend database and displays an appropriate message.

8.3.2 Alternative Flows

8.3.2.1 Reset Cancelled

If in the **Basic Flow**, the Administrator decides not to delete the entire existing information, the reset is cancelled and the use case ends.

8.4 Special Requirements

None

8.5 Pre-Conditions

The Administrator must be logged onto the system before this use case begins.

8.6 Post-Conditions

If the use case was successful, all the existing information is deleted from the backend database of the system. Otherwise, the system state is unchanged.

8.7 Extension Points

None

3.5.7 SRS Document

- 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 Interface Requirements
 - 3.1.1 User Interfaces
 - Login Screen:
 - Subject Info Parameters Screen:
 - Subject Information Screen:
 - Student Info Parameters Screen:
 - Student Information Screen:
 - Students' Subject Choice Parameters Screen:
 - Students' Subject Choice Information Screen:

(Contd.)...

- Marks Entry Parameters Screen:
- Marks Entry Screen:
- Mark-sheet Parameters Screen:
- Students List Report Parameters Screen:
- Marks List Report Parameters Screen:
- Rank-wise List Report Parameters Screen:
- Students' Subject Choices List Report Parameters Screen:
 - 3.1.2 Hardware Interfaces
 - 3.1.3 Software Interfaces
 - 3.1.4 Communications Interfaces
- 3.2 Software Product Features
 - 3.2.1 Subject Information Maintenance
 - Validity Checks
 - Sequencing Information
 - Error Handling/ Response to Abnormal situations
 - 3.2.2 Student Information Maintenance
 - Validity Checks
 - Sequencing Information
 - Error Handling / Response to Abnormal situations
 - 3.2.3 Marks Info Maintenance
 - Validity Checks
 - Sequencing Information
 - Error Handling/ Response to Abnormal situations
 - 3.2.4 Mark sheet Generation
 - 3.2.5 Report Generation
 - Student List Reports
 - Students' Subject Choices List Reports
 - Semester-wise Mark lists
 - Rank-wise List Report
- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Software System Attributes
 - 3.5.1 Security
 - 3.5.2 Maintainability
 - 3.5.3 Portability
- 3.6 Logical Database Requirements
- 3.7 Other Requirements

1 Introduction

This document aims at defining the overall software requirements for 'Student Result Management System'. Efforts have been made to define the requirements exhaustively and

accurately. The final product will be having only features/functionalities mentioned in this document and assumptions for any additional functionality/feature should not be made by any of the parties involved in developing/testing/implementing/using this product. In case it is required to have some additional features, a formal change request will need to be raised and subsequently a new release of this document and/or product will be produced.

1.1 Purpose

This specification document describes the capabilities that will be provided by the software application ‘Student Result Management System’. It also states the various required constraints by which the system will abide. The intended audience for this document are the development team, testing team and end users of the product.

1.2 Scope

The software product ‘Student Result Management System’ will be an MIS and Reporting application that will be used for result preparation and management of M. Tech. Program of a University. The application will manage the information about various students enrolled in this course in different years, the subjects offered during different semesters of the course, the students’ choices for opting different subjects, and the marks obtained by various students in various subjects in different semesters. Printable reports regarding list of students, marks obtained by all students in a particular semester and performance of students (rank-wise, percentage-wise, pass/fail, division-wise.) will be generated. The system will also generate printable mark-sheets for individual students.

The application will greatly simplify and speed up the result preparation and management process.

1.3 Definitions, Acronyms, and Abbreviations

Following abbreviations have been used throughout this document:

M.Tech: Master of Technology

IT: Information Technology

DBA: Database Administrator

1.4 References

- (i) *University website*: For information about course structure of M.Tech. Program
- (ii) IEEE Recommended Practice for Software Requirements Specifications—IEEE Std 830-1993

1.5 Overview

The rest of this SRS document describes the various system requirements, interfaces, features and functionalities in detail.

2 Overall Description

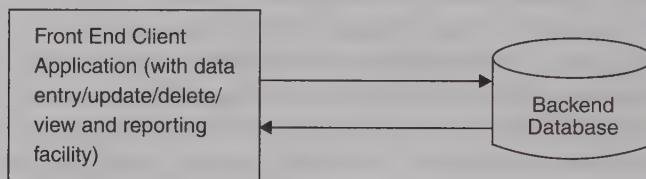
M.Tech. Program is a 4-semester course. The students are offered 4 subjects (theory) and 2 Labs (practical) during first, second and third semesters. Students also have to submit a term paper/minor project in 2nd and 3rd semesters. The fourth semester consists of a seminar and

disseratation. Each subject/lab/term paper/seminar/dissertation has credits associated with it. When a student secures pass marks in a paper he/she also earns all the credit (s) assigned to that paper.

The ‘Student Result Management System’ will have capability to maintain information about students enrolled in the course, the subjects offered to students during different semesters, the students’ choices for opting different Elective subjects (out of the available ones) and the marks obtained by students in different subjects in various semesters. The software will also generate summary reports regarding student information, semester-wise mark lists and performance reports. Printable mark-sheets of individual students will also be generated by the application.

2.1 Product Perspective

The application will be a windows-based, self-contained and independent software product.



2.1.1 System Interfaces

None

2.1.2 User Interfaces

The application will have a user-friendly and menu based interface. Following screens will be provided:

- (i) A Login screen for entering the username, password and role (Administrator, Data Entry Operator, Marks Entry Clerk, Coordinator) will be provided. Access to different screens will be based upon the role of the user.
- (ii) There will be a screen for capturing and displaying information regarding what all subjects are offered during which semester, how many credit points are assigned to that subject and whether the subject is an elective, a core paper, a lab paper, a term paper or a dissertation.
- (iii) There will be a screen for capturing and displaying information regarding various students enrolled for the course in different years.
- (iv) There will be a screen for capturing and displaying information regarding which student is currently enrolled in which semester and what all elective subjects he/she has opted.
- (v) There will be a screen that will capture information regarding which student has scored how many marks (internal + external evaluation) in each subject (in a particular semester). Credits in each subject will be calculated depending upon the marks obtained in that subject.

(vi) There will be a screen for capturing and displaying information regarding which all user accounts exist in the system, thus showing who all can access the system.

The following reports will be generated:

- (i) *Students' List Report*: Printable reports will be generated to show the list of students enrolled in a particular batch year.
- (ii) *Students' Subject Choices List Report*: For Ist, IInd and IIIrd semester, there will be printable reports showing the different elective subjects opted by various students (enrolled in a particular batch year) of the corresponding semester.
- (iii) *Marks List Report*: For each semester there will be a printable report showing the subject-wise marks details for all students of that semester.
- (iv) *Rank-wise List Report*: For each semester there will be a printable report showing the percentage-wise and rank-wise list of students along with the division secured.
- (v) *Mark-sheet*: For each student of each semester, a printable mark sheet will be generated, showing the subject-wise marks details, Total marks, total credits, Percentage, Pass/Fail status for that student.

2.1.3 Hardware Interfaces

- (i) Screen resolution of at least 800×600 —required for proper and complete viewing of screens. Higher resolution would not be a problem.
- (ii) Support for printer (dot-matrix/deskJet/inkjet etc.—any will do)—that is, appropriate drivers are installed and printer connected Printer will be required for printing of reports and mark-sheets.
- (iii) Standalone system or network based—not a concern, as it will be possible to run the application on any of these.

2.1.4 Software Interfaces

- (i) Any windows-based operating system (Windows 95/98/2000/XP/NT)
- (ii) MS Access 2000 as the DBMS—for database. Future release of the application will aim at upgrading to Oracle 8i as the DBMS.
- (iii) Crystal Reports 8—for generating and viewing reports.
- (iv) Visual Basic 6—for coding/developing the software.

Software mentioned in pts. (iii) and (iv) above, will be required only for development of the application. The final application will be packaged as an independent setup program that will be delivered to the client (University in this case).

2.1.5 Communications Interfaces

None

2.1.6 Memory Constraints

At least 64 MB RAM and 2 GB space on hard disk will be required for running the application.

2.1.7 Operations

This product release will not cover any automated housekeeping aspects of the database. The DBA at the client site (*i.e.*, University) will be responsible for manually deleting old/non-required data. Database backup and recovery will also have to be handled by the DBA.

However, the system will provide a ‘RESET SYSTEM’ function that will delete (upon confirmation from the Administrator) all the existing information from the database.

2.1.8 Site Adaptation Requirements

The terminals at client site will have to support the hardware and software interfaces specified in above sections.

2.2 Product Functions

The system will allow access only to authorized users with specific roles (Administrator, Data Entry Operator, Marks Entry Clerk and Coordinator). Depending upon the user’s role, he/she will be able to access only specific modules of the system.

A summary of the major functions that the software will perform:

- (i) A Login facility for enabling only authorized access to the system.
- (ii) User (with role Data Entry Operator) will be able to add/modify/delete information about different students that are enrolled for the course in different years.
- (iii) User (with role Data Entry Operator) will be able to add/modify/delete information about different subjects that are offered in a particular semester. The semester-wise list of subjects along with their credit points and type (*i.e.*, elective/core/lab/term paper/dissertation) will also be displayed.
- (iv) User (with role Data Entry Operator) will be able to add/modify/delete information about the Elective subjects opted by different students in different semesters.
- (v) User (with role Marks Entry Clerk) will be able to add/modify/delete information regarding marks obtained by different students in different semesters.
- (vi) User (with role Marks Entry Clerk) will also be able to print mark-sheets of students.
- (vii) User (with role Coordinator) will be able to generate Printable reports (as mentioned in section 2.1.2 above).
- (viii) User (with role Administrator) will be able to ‘Reset’ the system-leading to deletion of all existing information from the backend database.
- (ix) User (with role Administrator) will be able to create/modify/delete new/existing user accounts.

2.3 User Characteristics

- *Educational level:* At least graduate should be comfortable with English language.
- *Experience:* Should be well versed/informed about the course structure of M. Tech. program of University. Entry of marks or their modification can be done only by user who is authorized for this job by the result preparation committee of University.

- *Technical expertise:* Should be comfortable using general-purpose applications on a computer.

2.4 Constraints

- (i) Since the DBMS being used is MS Access 2000, which is not a very powerful DBMS, it will not be able to store a very huge number of records.
- (ii) Due to limited features of DBMS being used performance tuning features will not be applied to the queries and thus the system may become slow with the increase in number of records being stored.
- (iii) Due to limited features of DBMS being used, database auditing will also not be provided.
- (iv) Users at University will have to implement a security policy to safeguard the marks-related information from being modified by unauthorized users (by means of gaining access to the backend database)

2.5 Assumptions and Dependencies

- (i) The number of subjects to be taken up by a student in each semester does not change.
- (ii) The subject types (*i.e.*, elective, core, lab, term paper and dissertation) do not change.
- (iii) The number of semesters in the M. Tech. Program does not change.

2.6 Apportioning of Requirements

Not Required.

3 Specific Requirements

This section contains the software requirements to a level of detail sufficient to enable designers to design the system, and testers to test that system.

3.1 External Interface Requirements

3.1.1 User Interfaces

The following screens will be provided:

Login Screen:

This will be the first screen that will be displayed. It will allow user to access different screens based upon the user's role. Various fields available on this screen will be

- (i) *User ID:* Alphanumeric of length upto 10 characters
- (ii) *Password:* Alphanumeric of length upto 8 characters
- (iii) *Role:* Will have the following values:

Administrator, Marks Entry Clerk, Coordinator, Data Entry Operator

Subject Info Parameters Screen:

This screen will be accessible only to user with role Administrator. It will allow the user to enter the semester number for which the user wants to access the subject information.

Subject Information Screen:

This screen will be accessible only to user with role Administrator. It will allow user to add/modify/delete information about new/existing subject (s) for the semester that was selected in the 'Subject Info Parameters' screen. The list of available subjects for that semester, will also be displayed. Various fields available on this screen will be:

- (i) *Subject Code*: of the format IT-### (# represents a digit)
- (ii) *Subject Name*: Alphanumeric, of length upto 50 characters
- (iii) *Category/Type*: Will have any of the following values:-

Elective 1/Elective 2/Core/Lab/Term paper/Seminar/Dissertation

- (iv) *Credits*: Numeric, will have any value from 0 to 20.

Student Info Parameters Screen:

This screen will be accessible only to user with role Administrator. It will allow the user to enter the Batch Year for which the user wants to access the student information.

Student Information Screen:

This screen will be accessible only to user role Administrator. It will allow the user to add/modify/delete information about new/existing student(s) for a particular Batch Year. Batch Year-wise list of students will also be displayed. Various fields available on these screens will be:

- (i) *Student Enrollment No*: of the format ##/M.Tech. (IT)/YYYY (# represents a digit and YYYY represents the batch year)
- (ii) *Student Name*: will have only alphabetic letters and length upto 40 characters
- (iii) *Batch Year*: of the format YYYY (representing the year in which the student enrolled for the course)

Students' Subject Choice Parameters Screen:

This screen will be accessible only to user with role Administrator. It will allow the user to enter the Batch Year and the semester number for which the user wants to access the students' subject choice information.

Students' Subject Choice Information Screen:

This screen will be accessible only to user with role Administrator. It will allow user to add/modify/delete students' choices for elective subjects of the semester and batch year selected in "Students' Subject Choice Parameters" screen. For the selected semester it will display the list of available choices for Elective I and for Elective II. The screen will display the list of students enrolled during the selected batch year and currently studying in the selected semester and the user will be able to view/add/modify/delete the subject choices for each student in the list.

Marks Entry Parameters Screen:

This screen will be accessible only to user with role Marks Entry Clerk. It will allow the user to enter the Batch Year, the semester number and the Subject for which the user wants to access the marks information.

Marks Entry Screen:

This screen will be accessible only to user with role Marks Entry Clerk. It will allow user to add/modify/delete information about marks obtained in the selected subjects by different students of that semester who were enrolled in the Batch Year selected in the 'Marks Entry Parameters' Screen. The screen will display the list of students enrolled during the selected batch year and currently studying the selected subject in the selected semester and the user will be able to view/add/modify/delete the marks for each student in the list.

Various fields available on this screens will be:

- (i) *Student Enrollment No*: will display the enrollment numbers of all students of the selected Batch Year studying the selected subject in the selected semester.
- (ii) *Student Name*: will display the name of the student
- (iii) *Internal Marks*: between 0 and 40
- (iv) *External Marks*: between 0 and 60
- (v) *Total Marks*: sum of Internal Marks and External Marks

Mark-sheet Parameters Screen:

This screen will be accessible only to user with role Marks Entry Clerk. It will allow the user to enter the Enrollment Number and the semester number of the student for whom the user wants to view/print the mark-sheet.

Students List Report Parameters Screen:

This screen will be accessible only to user with role Coordinator. It will allow the user to enter the Batch Year for which the user wants to view/print the students list report.

Marks List Report Parameters Screen:

This screen will be accessible only to user with role Coordinator. It will allow the user to enter the Batch Year and the semester for which the user wants to view/print the marks list report.

Rank-wise List Report Parameters Screen:

This screen will be accessible only to user with role Coordinator. It will allow the user to enter the Batch Year and the semester for which the user wants to view/print the rank-wise list report.

Students' Subject Choices List Report Parameters Screen:

This screen will be accessible only to user with role Coordinator. It will allow the user to enter the Batch Year and the semester for which the user wants to view/print the students' subject choices list report.

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 Communications Interfaces

None

3.2 System Features

3.2.1 Subject Information Maintenance

Description

The system will maintain information about various subjects being offered during different semesters of the course. The following information would be maintained for each subject:

Subject code, Subject name, Subject Type (Core/Elective 1/Elective 2/Lab 1/Lab 2/Term Paper/Minor Project/Dissertation/Seminar), Semester, Credits.

The system will allow creation/modification/deletion of new/existing subjects and also have the ability to list all the available subjects for a particular semester.

Validity Checks

- (i) Only user with role Data Entry Operator will be authorized to access the Subject Information Maintenance module.
- (ii) Ist, IIInd and IIIrd semesters will have 2 core papers, 2 Elective papers, 2 Lab papers and 1 term paper/Minor Project.
- (iii) Ist, IIInd and IIIrd semesters will have 3 choices (subjects) each of type Elective 1 and of type Elective 2.
- (iv) IVth semester will have only 1 dissertation and 1 seminar.
- (v) No two semesters will have the same subject *i.e.*, A subject will be offered only in a particular semester.
- (vi) Subject code will be unique for every subject.
- (vii) subject code cannot be blank.
- (viii) Subject name cannot be blank.
- (ix) Credits cannot be blank.
- (x) Credits can have value only between 0 and 20.
- (xi) Subject Type cannot be blank.
- (xii) Semester cannot be blank.

Sequencing Information

Subject info for a particular semester will have to be entered in the system before any student/marks information for that semester can be entered.

Error Handling/Response to Abnormal Situations

If any of the above validations/sequencing flow does not hold true, appropriate error messages will be prompted to the user for doing the needful.

3.2.2 Student Information Maintenance

Description

The system will maintain information about various students enrolled in the M.Tech. (ITW) course in different years. The following information would be maintained for each student :

Student Enrollment Number, Student Name, Year of Enrollment.

The system will allow creation/modification/deletion of new/existing students and also have the ability to list all the students enrolled in a particular year.

Validity Checks

- (i) Only user with role Data Entry Operator will be authorized to access the Student Information Maintenance module.
- (ii) Every student will have a unique Enrollment Number.
- (iii) Enrollment Number cannot be blank.
- (iv) Student name cannot be blank.
- (v) Enrollment Year cannot be blank.

Sequencing Information

Student Info for a particular student will have to be entered in the system before any marks info can be entered for that student.

Error Handling/Response to Abnormal Situations

If any of the above validations/sequencing flow does not hold true, appropriate error messages will be prompted to the user for doing the needful.

3.2.3 Students' Subject Choices Information Maintenance

Description

The system will maintain information about choice of different Elective subjects opted by various students of different enrollment years in different semesters. The following information would be maintained:

Student Enrollment number, Semester, Student's Choice for Elective 1 subject, Student's Choice for Elective 2 subject.

The system will allow creation/modification/deletion of students' subject choices and also have the ability to list all the available students' subject choices for a particular semester.

Validity Checks

- (i) Only user with role Data Entry Operator will be authorized to access the Students' Subject Choices Information Maintenance module.
- (ii) The subject choice for Elective 1 and Elective 2 can be made only from the list of available choices for that semester.

Sequencing Information

Students' Subject Choices Info for a particular student can be entered in the system only after Subject Info has been entered in the system for the given semester and the Student Info for that student has been entered in the system.

Students' Subject Choices Info for a particular student will have to be entered in the system before any marks info can be entered for that student in the given semester.

Error Handling/Response to Abnormal Situations

If any of the above validations/sequencing flow does not hold true appropriate error messages will be prompted to the user for doing the needful.

3.2.4 Marks Information Maintenance

Description

The system will maintain information about marks obtained by various students of different enrollment years in different semesters. The following information would be maintained:

Student Enrollment Number, Semester, Subject Code, Internal Marks, External Marks, Total Marks, and Credits.

The system will allow creation/modification/deletion of marks information and also have the ability to list all the available marks information for all students for a particular subject in the given semester.

Validity Checks

- (i) Only user with role Marks Entry Clerk will be authorized to access the Marks Information Maintenance module.
- (ii) Internal Marks for any subject cannot be less than 0 and greater than 40.
- (iii) External marks for any subject cannot be less than 0 and greater than 60.
- (iv) Total marks in any subject will be calculated as: Internal Marks in that subject + External Marks in that subject.
- (v) If the total Marks in a subject are $>= 50$, all the credit points associated with that subject will be given to the student, else the credit points earned by the student will be 0 for that subject.

Sequencing Information

Marks Info for a particular student can be entered in the system only after Subject Info has been entered in the system for the given semester, the Student Info for that student has been entered in the system, and the Students' Subject Choice Info has been entered in the system for that student in the given semester.

Marks info for a particular student will have to be entered in the system before that student's mark-sheet can be generated.

Error handling/Response to Abnormal Situations

If any of the above validations/sequencing flow does not hold true appropriate error messages will be prompted to the user for doing the needful.

3.2.5 Mark-sheet Generation

Description

The system will generate mark-sheet for every student in different semesters.

Mark-sheet will have the following format:

Name of the University

Name of Program

Semester <no.>

Mark-sheet

Student Enrollment No. _____ Student Name: _____

<i>S.No.</i>	<i>Subject</i>	<i>Internal Marks (Out of 40)</i>	<i>External Marks (Out of 60)</i>	<i>Total Marks (Int. + Ext.)</i>	<i>Pass/Fail</i>	<i>Credits Earned</i>
1.						
2.						
3.						
4.						
5.						
6.						

Marks Grand Total: _____ /600 Total Credits: _____

Result: (Pass/Fail)

› Date:

Signature of Controller of Examination

There will be a 'Print' icon at the top of mark sheet for printing the mark-sheet.

Validity Checks

- (i) Only user with role Marks Entry Clerk will be authorized to access the Mark-sheet Generation module.

Sequencing Information

Marks-sheet for a particular student can be generated by the system only after Subject info has been entered in the system for the given semester, the Student Info for that student has been entered in the system, the Students' Subject Choice Info has been entered in the system for that student in the given semester, and the Marks Info has been entered for that student for the given semester.

Error Handling/Response to Abnormal Situations

If any of the above validations/sequencing flow does not hold true appropriate error messages will be prompted to the user for doing the needful.

3.2.6 Report Generation

Student List Reports

For each year a report will be generated containing the list of students enrolled in that batch year.

Report Format:

*Name of University
Name of the Program
List of students enrolled in year xxxx*

S.No.	Student Enrollment Number	Student Name
1.		
2.		

Students' Subject Choices List Reports

For each batch year a report will be generated containing the list of students and their choices for Elective subjects in the selected semester. For Ist, IInd and IIIrd semesters the list will contain the names of elective subjects opted by each student. For IVth semester the list will contain the topic/subject area of dissertation for each student.

Report Format (for Ist, IInd and IIIrd Semesters):

*Name of University
Name of the Program*

S.No.	Student Enrollment Number	Student Name	Elective 1		Elective 2	
			Code	Name	Code	Name
1.						
2.						

Report Format (for IVth Semester):

*Name of University
Name of the Program*

S.No.	Student Enrollment Number	Student Name	Topic/Subject Area of Dissertation
1.			
2.			

Semester-wise Mark Lists

For each semester a mark list will be generated that will have the total marks (internal + external) of all students (enrolled in the selected Batch Year) of that semester in all subjects.

Report Format:

*Name of University
Name of the Program*

S.No.	Enrollment Number	Subject 1 Total Marks	Subject 2 Total Marks	Subject 3 Total Marks	Subject 4 Total Marks	Subject 5 Total Marks	Subject 6 Total Marks
1.							
2.							

Rank-wise List Report

This report will be generated for each semester of every Batch Year. It will show the Grand Total marks, Percentage, Rank and Division secured of all students of that semester. The report will be sorted in increasing order of Percentage/Rank.

Report Format:

*Name of University
Name of the Program*

S.No.	Enroll. No.	Name	Grand Total Marks	%age	Rank	Division
1.						
2.						

3.2.7 User Accounts Information Maintenance

Description

The system will maintain information about various users who will be able to access the system. The following information would be maintained:

User Name, User ID, Password, and Role.

Validity Checks

- (i) Only user with role Administrator will be authorized to access the User Accounts Information Maintenance module.

- (ii) User Name cannot be blank.
- (iii) User ID cannot be blank.
- (iv) User ID should be unique for every user.
- (v) Password cannot be blank.
- (vi) Role cannot be blank.

Sequencing Information

User Account for a particular user has to be created in order for the system to be accessible to that user. At system startup, only a default user account for 'Administrator' would be present in the system.

Error Handling/Response to Abnormal Situations

If any of the above validations/sequencing flow does not hold true, appropriate error messages will be prompted to the user for doing the needful.

3.3 Performance Requirements

None

3.4 Design Constraints

None

3.5 Software System Attributes

3.5.1 Security

The application will be password protected. Users will have to enter correct username, password and role in order to access the application.

3.5.2 Maintainability

The application will be designed in a maintainable manner. It will be easy to incorporate new requirements in the individual modules (*i.e.*, subject info, student info, students' subject choices info, marks info, report generation and user accounts info).

3.5.3 Portability

The application will be easily portable on any windows-based system that has MS-Access 2000 installed.

3.6 Logical Database Requirements

The following information will be placed in a database:

- (i) **Subject Info:** Subject Name, Code, Credit points, Type and Semester
- (ii) **Student Info:** Student enrollment number, Student Name, enrollment year
- (iii) **Students' Subject Choice Info:** Student Enrollment No, Semester, Choice of Elective 1 subject, Choice of Elective 2 subject

- (iv) **Marks Info:** Student Enrollment No., Semester, internal Marks in each subject, External Marks in each subject
- (v) **User Account Info:** User Name, User ID, Password, Role.

3.7 Other Requirements

None

REFERENCES

- [BERR98] Berry D.M. & B. Lawrence, "Requirements Engineering", IEEE Software, Mar/April, 26–29, 1998.
- [BITT03] Bittner K. and I spence, "Use Case Modelling", Pearson Education, 2003.
- [BROO95] Brooks F.P., "No Silver Bullet", The Mythical Man Month: Essay on Software Engineering (Second Edition), Addison Wesley Longman, Reading, Mass, 179–209, 1995.
- [CAIN75] Caine S. & E. Gordon, "A Tool for Software Design", Proceeding of the AFIPS, National Computer Conference, Vol. 47, AFIPS Press, Montvale, NJ, 271–276, 1975.
- [CARE90] Carey J. M., "Prototyping: Alternative Systems Development Methodology", Information and Software Technology, Vol. 32, No. 2, March 1990, 119–126.
- [CERI88] Ceri S. et al., "Software Prototyping by Relational Techniques: Expenses with Program Construction Systems", IEEE Transaction on Software Engineering, Vol. 14, No. 11, Nov. 1988, 1597–1609.
- [CHEN76] Chen P., "The Entity–Relationship Model—Towards the Unified View of Data", ACM Trans on database systems, March, 9–36, 1976.
- [COAD89] Coad P. & E. Yourdon, "OOA—Object Oriented Analysis", Englewood Cliffs, N.J. Prentice Hall, 1989.
- [COUN99] Councill B., "Third–Party Testing and Stirrings of the New Software Engineering", IEEE Software, Nov./Dec., 76–88, 1999.
- [DAV194] Davis A. & P. Hsia, "Giving Voice to Requirements Engineering", IEEE Software, March, 12–16, 1994.
- [DAVI90] Davis A.M., "Software Requirements Analysis and Specification", PH, Englewood Cliffs, NJ, 1990.
- [DEMA79] DeMacro T., "Structured Analysis and System Specification", Englewood Cliffs, N.J., PH, 1979.
- [HEKM87] HeKmatpour S., "Experience Evolutionary Prototyping in Large Software Project", ACM Software Engineering Notes, Vol. 12, No. 1, January, 1987, 38–41.
- [HEME82] Hemenway K. & L. X. McCusker, "Prototyping & Evaluating a User Interface", Proc. of the 6th International Computer Software & Applications Conference, Chicago, Illinois, Nov. 1982, 175–180.
- [HICK03] Hickey A.M. and A.M. Davis, "Requirements Elicitation and Elicitation Technique Selection: A Model for Two Knowledge Intensive Software Development Processes", Proc. of 36th Hawaii Int. conf. on system science, 2003.
- [HOFF99] Hoffer J.A. et al., "Modern Systems Analysis and Design", Addison–Wesley, 1999.
- [HOOP89] Hooper W.J., "Languages Features for Prototyping and Simulation Support of the Software Life Cycle", Computer Languages, Vol. 14, No. 2, Feb. 1989, 83–92.

- [HSIA93] Hsia P., A. Davis & D. Kung, "Status Report: Requirements Engineering", IEEE Software, 75–79, Nov. 1993.
- [IEEE87] IEEE Standard for Project Management Plans (ANSI), IEEE Std., 1058.1, 1987.
- [IEEE94] IEEE Standards for Requirements Engineering, 1994.
- [IEEE97] IEEE Standard Taxonomy for Software Engineering Standards (ANSI), 1997.
- [IEEE93] IEEE guide to software requirements specifications (std. 830–1993). QS/IRM/Private/Initial/QA/QA Plan/Sig Plan, DOC, 1993.
- [MACA96] Macaulay L., "Requirements Engineering", Springer, 1996.
- [MANT88] Mantel M. & T. Teorey, "Cost/Benefit for Incorporating Human Factors in the Software Life Cycle", Communications of the ACM 31, 4, April, 428–439, 1988.
- [MARC88] Marca D. & C. McGowan, "Structured Analysis and Design Techniques", New York, MH., 1988.
- [MARC88] Marco D. & C. McGowan, "Structured Analysis & Design Techniques", NY, McGraw Hill, 1988.
- [MARC88] Marco D. & McGowan, "Structured Analysis & Design Techniques", NY, McGraw Hill, 1988.
- [MUSA87] Musa J. et al., "Software Reliability", New York, McGraw Hill, 1987.
- [PRES2K] Pressman R.S., "Software Engineering", McGraw Hill, 2000.
- [RATC88] Ratcliff B., "Early and Not So Early Prototyping—Rationale and Tool Support", Proc. of the 12th Annual Int. Computer Software & Application Conference, Chicago, Illinois, IEEE Computer Society Press, Oct, 1988, 127–134.
- [ROBE02] Robert T. Futrell et. al. "Quality Software Project Management", Pearson Education Asia, 2002.
- [ROSS77] Ross D., "Structured Analysis: A Language for Communicating Ideas", IEEE Trans on Software Engineering, 3,1, Jan, 6–15, 1977.
- [SAGE90] Sage A.P. & J.D. Palma, "Software System Engineering", John Wiley & Sons, 1990.
- [SAWY99] Sawyer P. et al., "Capturing the benefits of Requirement Engineering", IEEE Software, 78–85, March/April, 1999.
- [SKEL86] Skelton S., "Measurements of Migratability and Transportability", ACM Software Engineering Notes, 11, 1, 29–34, 1986.
- [SOMM96] Sommerville I., "Software Engineering", Addison Wesley, 1996.
- [SOMM01] Sommerville I., "Software Engineering", Pearson Education, 2001.
- [THAY97] Thayer R.H. & M. Dorfman, "Software Requirements Engineering", IEEE Computer Society, LA, 1997.
- [TOZE87] Tozer J. E., "Prototyping as a System Development Methodology: Opportunities and Pitfalls", Information and Software Technology, Vol. 29, No. 5, June 1987, 265–269.
- [WIEG99] Wiegers K.E., "Software Requirements", Microsoft Press, Washington, USA, 1999.
- [YOUR79] Yourdon E. & L. Constantine, "Structured Design", Englewood Cliffs, NJ, Prentice-Hall, 1979.
- [ZULT92] Zultner R., "Quality Function Deployment for Software: Satisfying Customers" American Programmer, February 28–41' 1992.

MULTIPLE CHOICE QUESTIONS

Note. Choose the most appropriate answer of the following questions.

- 3.13.** Context diagram explains
 (a) The overview of the system
 (c) The entities of the system
3.14. DFD stands for
 (a) Data flow design
 (c) Data flow diagram
3.15. Level-O DFD is similar to
 (a) Use case diagram
 (c) System diagram
3.16. ERD stands for
 (a) Entity relationship diagram
 (c) Entity relationship design
3.17. Which is not a characteristic of a good SRS ?
 (a) Correct
 (c) Consistent
3.18. Outcome of requirements specification phase is
 (a) Design document
 (c) Test document
3.19. The basic concepts of ER model are:
 (a) Entity and relationship
 (c) Entity, effects and relationship
3.20. The DFD depicts
 (a) Flow of data
 (c) Both (a) and (b)
- (b) The internal view of the system
 (d) None of the above.
 (b) Descriptive functional design
 (d) None of the above.
 (b) Context diagram
 (d) None of the above.
 (b) Exit related diagram
 (d) Exit related design.
 (b) Complete
 (d) Brief.
 (b) Software requirements specification
 (d) None of the above.
 (b) Relationships and keys
 (d) Entity, relationship and attribute.
 (b) Flow of control
 (d) None of the above.

EXERCISES

- 3.1.** Discuss the significance and use of requirement engineering. What are the problems in the formulation of requirements ?
- 3.2.** Requirements analysis is unquestionably the most communication intensive step in the software engineering process. Why does the communication path frequently break down ?
- 3.3.** What are crucial process steps of requirement engineering ? Discuss with the help of a diagram.
- 3.4.** Discuss the present state of practices in requirement engineering. Suggest few steps to improve the present state of practice.
- 3.5.** Explain the importance of requirements. How many types of requirements are possible and why ?
- 3.6.** Describe the various steps of requirements engineering. Is it essential to follow these steps ?
- 3.7.** What do you understand with the term “requirements elicitation” ? Discuss any two techniques in detail.
- 3.8.** List out requirements elicitation techniques. Which one is most popular and why ?
- 3.9.** Describe facilitated application specification technique (FAST) and compare this with brainstorming sessions.

- 3.10.** Discuss quality function deployment technique of requirements elicitation. Why an importance or value factor is associated with every requirement ?
- 3.11.** Explain the use case approach of requirements elicitation. What are use-case guidelines ?
- 3.12.** What are components of a use case diagram. Explain their usage with the help of an example.
- 3.13.** Consider the problem of library management system and design the following:
- (i) Problem statement
 - (ii) Use case diagram
 - (iii) Use cases.
- 3.14.** Consider the problem of railway reservation system and design the following:
- (i) Problem statement
 - (ii) Use case diagram
 - (iii) Use cases.
- 3.15.** Explain why a many to many relationship is to be modeled as an associative entity ?
- 3.16.** What are the linkages between data flow and E-R diagrams ?
- 3.17.** What is the degree of a relationship ? Give an example of each of the relationship degree.
- 3.18.** Explain the relationship between minimum cardinality and optional and mandatory participation.
- 3.19.** An airline reservation is an association between a passenger, a flight, and a seat. Select a few pertinent attributes for each of these entity types and represent a reservation in an E-R diagram.
- 3.20.** A department of computer science has usual resources and usual users for these resources. A software is to be developed so that resources are assigned without conflict. Draw a DFD specifying the above system.
- 3.21.** Draw a DFD for result preparation automation system of B. Tech. courses (or MCA program) of any university. Clearly describe the working of the system. Also mention all assumptions made by you.
- 3.22.** Write short notes on
- (i) Data flow diagram
 - (ii) Data dictionary.
- 3.23.** Draw a DFD for borrowing a book in a library which is explained below: "A borrower can borrow a book if it is available else he/she can reserve for the book if he/she so wishes. He/she can borrow a maximum of three books".
- 3.24.** Draw the E-R diagram for a hotel reception desk management.
- 3.25.** Explain why, for large software systems development, is it recommended that prototypes should be "throw-away" prototype ?
- 3.26.** Discuss the significance of using prototyping for reusable components and explain the problems, which may arise in this situation.
- 3.27.** Suppose a user is satisfied with the performance of a prototype. If he/she is interested to buy this for actual work, what should be the response of a developer ?
- 3.28.** Comment on the statement: "The term throw-away prototype is inappropriate in that these prototypes expand and enhance the knowledge base that is retained and incorporated in the final prototype; therefore they are not disposed of or thrown away at all."
- 3.29.** Which of the following statements are ambiguous ? Explain why.
- (a) The system shall exhibit good response time.
 - (b) The system shall be menu driven.
 - (c) There shall exist twenty-five buttons on the control panel, numbered PF1 to PF25.
 - (d) The software size shall not exceed 128K of RAM.

- 3.30. Are there other characteristics of an SRS (besides listed in section 3.4.2) that are desirable ? List a few and describe why ?
- 3.31. What is software requirements specification (SRS) ? List out the advantages of SRS standards. Why is SRS known as the black box specification of a system ?
- 3.32. State the model of a data dictionary and its contents. What are its advantages ?
- 3.33. 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.
- 3.34. Construct an example of an inconsistent (incomplete) SRS.
- 3.35. Discuss the organisation of a SRS. List out some important issues of this organisation.

4

Software Project Planning

Contents

4.1 Size Estimation

4.1.1 Lines of Code

4.1.2 Function Count

4.2 Cost Estimation

4.3 Models

4.3.1 Static Single Variable Models

4.3.2 Static Multivariable Models

4.4 Constructive Cost Model (COCOMO)

4.4.1 Basic Model

4.4.2 Intermediate Model

4.4.3 Detailed COCOMO Model

4.5 COCOMO II

4.5.1 Application Composition Estimation Model

4.5.2 The Early Design Model

4.5.3 Post Architecture Model

4.6 Putnam Resource Allocation Model

4.6.1 The Norden/Rayleigh Curve

4.6.2 Difficulty Metric

4.6.3 Productivity Versus Difficulty

4.6.4 The Trade-off between Time Versus Cost

4.6.5 Development Sub-cycle

4.7 Software Risk Management

4.7.1 What is Risk ?

4.7.2 Typical Software Risks

4.7.3 Risk Management Activities

4

Software Project Planning

After the finalisation of SRS, we would like to estimate size, cost and development time of the project. Also, in many cases, customer may like to know the cost and development time even prior to finalisation of the SRS. Hence, whether we estimate before SRS or after SRS, it would always be very critical and crucial decision for the project. Estimation of cost and development time are the key issues during project planning. The correlation between cost, development time, and the planning process can best be illustrated by an example.

Suppose we want to put an addition on our home. After deciding what we want and getting several quotations, most of which are around 2.5 lacs, we pick up a builder who offers to do the job in two months for 2.0 lacs. We sign an agreement and the builder starts the work. After about a month into the job, the builder comes and explains that because of the problems the job will take an extra month and cost an additional 0.5 lacs. This creates several problems. First, we badly need space and another month of delay is a real inconvenience. Second, we have already arranged for a loan and do not know from where we can get this additional amount of Rs. 0.5 lac. Third, if we get a lawyer and decide to fight the builder in court, all work on the job will stop for many months while the case is decided. Fourth, it would take a great deal of time and probably cost even more to switch to a new builder in the middle of the job.

On exploration, we conclude that the real problem is that the builder did a sloppy job of planning. Builder might have forgot to include some major costs like the labour or materials to do the woodwork or final plastering and painting. Because other quotations were close to Rs. 2.5 lacs, we know that this is pretty fair price. At this point, we have no option but to try to negotiate a lower price but will continue with the current builder. However, we would neither use this builder again, nor would probably recommend the builder to anyone else [HUMP95].

This is the essential issue in the planning process; being able to make plans that accurately represent what we can do. Business operates on commitments, and commitments require plans. The failure of many large software projects in the 1960s and early 1970s highlighted this problem of poor planning. The delivered software was late, unreliable, costed several times the original estimates and often exhibited poor performance characteristics. These projects did not fail because managers or developers were incompetent. The fault was in the approach of planning that was used. Planning techniques derived from small-scale projects did not scale up to large systems development.

Software managers are responsible for planning and scheduling project development. They supervise the work to ensure that it is carried out to the required standards. They monitor progress to check that the development is on time and within budget. Good managers cannot guarantee project success. However, bad managers usually result in project failure. Usually, the software is delivered late, costs more than originally estimated and fails to meets its re-

quirements [SOMM95]. The project planning must incorporate the major issues like size and cost estimation, scheduling, project monitoring and reviews, personnel selection and evaluation, and risk management.

In order to conduct a successful software project, we must understand [PRES2K]

- scope of work to be done
- the risk to be incurred
- the resources required
- the task to be accomplished
- the cost to be expended
- the schedule to be followed

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired. The various steps of planning activities are illustrated in Fig. 4.1. As shown, first activity is to estimate the size of the project. The size is the key parameter for the estimation of other activities. It is an input to all costing models for the estimation of cost, development time and schedule for the project. If size estimation is not reasonable, it may have serious impact on the other estimation activities.

Resources requirements are estimated on the basis of cost and development time. Project scheduling may prove to be very useful for controlling and monitoring the progress of the project. This is dependent on the resources and development time.

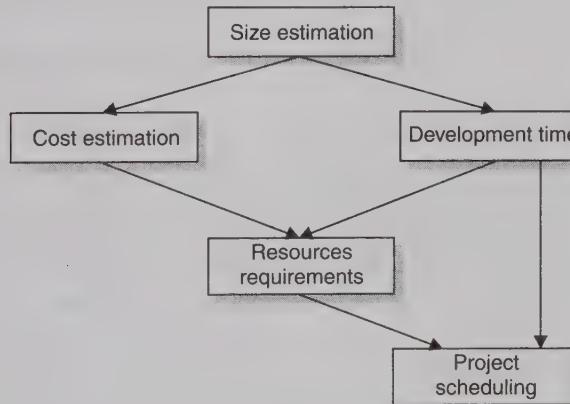


Fig. 4.1: Activities during software project planning.

4.1 SIZE ESTIMATION

Some programs are written in C, some in PASCAL, others in FORTRAN, and still others in assembly language. Some programs are for GUI applications and some are for batch processing. Some are written using the latest software engineering techniques, while others are developed without adequate planning. Some programs are well documented with carefully crafted internal comments, other programs are written in a quick and dirty fashion with no comments at all. But, there is one characteristic that all programs share—they all have size [CONT86].

The estimation of size is very critical and difficult area of the project planning. It has been recognised as a crucial step from the very beginning. The difficulties in establishing units for measuring size lie in the fact that the software is essentially abstract: it is difficult to identify the size of a system. Other engineering disciplines have the advantage that a bridge or a building or a road can be seen and touched, they are (sometimes literally) concrete. Many attempts have been made at establishing a unit of measure for size. The more widely known are given below.

4.1.1 Lines of Code (LOC)

This was the first measurement attempted. It has the advantage of being easily recognizable, seen and therefore counted. Although this may seem to be a simple metric that can be counted algorithmically, there is no general agreement about what constitutes a line of code. Early users of lines of code did not include data declarations, comments, or any other lines that did not result in object code. Later users decided to include declarations and other unexecutable statements but still excluded comments and blank lines. The reason for this shift is the recognition that contemporary code can have 50% or more data statements and that bugs occur as often in such statements as in real code. For example, in the function shown in Fig. 4.2, if LOC is simply a count of the number of lines then Fig. 4.2 contains 18 LOC [CONT86].

1.	int. sort (int x[], int n)
2.	{
3.	int i, j, save, im1;
4.	/* This function sorts array x in ascending order */
5.	If (n < 2) return 1;
6.	for (i = 2; i < = n; i++)
7.	{
8.	im1 = i - 1;
9.	for (j = 1; j < = im; j++)
10.	if (x[i] < x[j])
11.	{
12.	Save = x[i];
13.	x[i] = x[j];
14.	x[j] = save;
15.	}
16.	}
17.	return 0;
18.	}

Fig. 4.2: A function for sorting an array in ascending order.

But most researchers agree that the LOC metric should not include comments or blank lines. Since these are really internal documentation and their presence or absence does not affect the functions of the program. Moreover, comments and blank lines are not as difficult to

construct as program lines. The inclusion of comments and blank lines in the count may encourage developers to introduce artificially many such lines in project development in order to create the illusion of high productivity, which is normally measured in LOC/PM (lines of code/person-month). When comments and blank lines are ignored, the program in Figs. 4.2 contains 17 LOC.

However, there is a fundamental reason for including comments in the program. The quality of comments materially affects maintenance costs because maintenance person will depend on the comments more than anything else to do the job. Conversely, too many blank lines and comments with poor readability and understandability will increase maintenance effort. The problem with including comments is that we must be able to distinguish between useful and useless comments, and there is no rigorous way to do that. Therefore, it is always advisable not to consider comments and blank lines while counting for LOC.

Furthermore, if the main interest is the size of the program for specific functionality, it may be reasonable to include executable statements. The only executable statements in Fig. 4.2 are in lines 5–17 leading to a count of 13. The differences in the counts are 18 to 17 to 13. One can easily see the potential for major discrepancies for large programs with many comments or programs written in languages that allow a large number of descriptive but non-executable statements. Conte [CONT86] has defined lines of code as:

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declarations, and executable and non-executable statements".

This is the predominant definition for lines of code used by researchers. By this definition, Fig. 4.2 has 17 LOC.

There are some disadvantages of this simple method of counting. LOC is language dependent. A line of assembler is not the same as a line of COBOL. They also reflect what the system is rather than what it does. Measuring systems by the number of lines of code is rather like measuring a building by the number of bricks involved in construction; useful when deciding on the type and number of brick layers but useless in describing the building as a whole. Buildings are normally described in terms of facilities, the number and size of rooms, and their total areas in square feet or meters.

While lines of code have their uses (e.g., in estimating programming time for a program during the build phase of a project), their usefulness is limited for other tasks like functionality, complexity, efficiency, etc. If counting LOC is similar to counting bricks in a building, then what is needed is some way of expressing the system in a way that is analogous to counting the number and size of rooms and their total area in square feet or meters.

4.1.2 Function Count

Measuring software size in terms of lines of code is analogous to measuring a car stereo by the number of resistors, capacitors and integrated circuits involved in its production. The number of components is useful in predicting the number of assembly line staff needed, but it does not say anything about the functions available in the finished stereo. When dealing with customers, the manufacturer talks in terms of functions available (e.g., digital tuning) and not in terms of components (e.g., integrated circuits).

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem [ALBR79, ALBR83]. It measures functionality from the users point of view, that is, on the basis of what the user requests and receives in return. Therefore, it deals with the functionality being delivered, and not with the lines of code, source modules, files, etc. Measuring size in this way has the advantage that size measure is independent of the technology used to deliver the functions. In other words, two identical counting systems, one written in 4 GL and the other in assembler, would have the same function count. This makes sense to the user, because the object is to buy an accounting system, not lines of assembler and it makes sense to the IT department, because they can measure the performance differences between the assembler and 4GL environments [STEP95].

Function point measures functionality from the users point of view, that is, on the basis of what the user requests and receives in return from the system. The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units

- Inputs : information entering the system.
- Outputs : information leaving the system.
- Enquiries : requests for instant access to information.
- Internal logical files : information held within the system.
- External interface files : Information held by other systems that is used by the system being analyzed.

The FPA functional units are shown in Fig. 4.3.

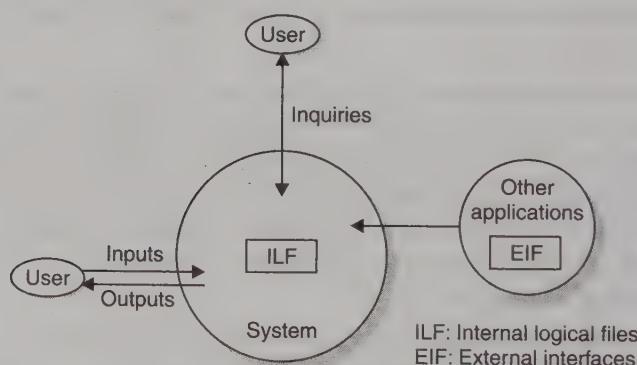


Fig. 4.3: FPAs functional units.

The five functional units are divided in two categories:

(i) Data function types

- *Internal Logical files (ILF)*: A user identifiable group of logically related data or control information maintained within the system.
- *External Interface files (EIF)*: A user identifiable group of logically related data or control information referenced by the system, but maintained within another system. This means that EIF counted for one system, may be an ILF in another system.

(ii) Transactional Function Types

- **External Input (EI):** An EI processes data or control information that comes from outside the system. The EI is an elementary process, which is the smallest unit of activity that is meaningful to the end user in the business.
- **External Output (EO):** An EO is an elementary process that generates data or control information to be sent outside the system.
- **External Inquiry (EQ):** An EQ is an elementary process that is made up of an input-output combination that results in data retrieval.

Special features

- Function point approach is independent of the language, tools, or methodologies used for implementation; *i.e.*, they do not take into consideration programming languages, data base management systems, processing hardware or any other data base technology.
- Function points can be estimated from requirement specification or design specifications, thus making it possible to estimate development effort in early phases of development.
- Function points are directly linked to the statement of requirements; any change of requirements can easily be followed by a re-estimate [INCE89].
- Function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

This method resolves many of the inconsistencies that arise when using lines of code as a software size measure [MATS94].

Counting function points

The five functional units are ranked according to their complexity *i.e.*, Low, Average, or High, using a set of prescriptive standards. Organisations that use FP methods develop criteria for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of Complexity is somewhat subjective.

After classifying each of the five function types, the Unadjusted Function Points (UFP) are calculated using predefined weights for each function type as given in Table 4.1.

Table 4.1: Functional units with weighting factors

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
Internal logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 4.2: UFP calculation table

<i>Functional Units</i>	<i>Count Complexity</i>	<i>Complexity Totals</i>	<i>Functional Unit Totals</i>
External Inputs (EIs)	<input type="text"/> Low × 3 = <input type="text"/> <input type="text"/> Average × 4 = <input type="text"/> <input type="text"/> High × 6 = <input type="text"/>		<input type="text"/>
External Outputs (EOs)	<input type="text"/> Low × 4 = <input type="text"/> <input type="text"/> Average × 5 = <input type="text"/> <input type="text"/> High × 7 = <input type="text"/>		<input type="text"/>
External Inquiries (EQs)	<input type="text"/> Low × 3 = <input type="text"/> <input type="text"/> Average × 4 = <input type="text"/> <input type="text"/> High × 6 = <input type="text"/>		<input type="text"/>
Internal logical files (ILFs)	<input type="text"/> Low × 7 = <input type="text"/> <input type="text"/> Average × 10 = <input type="text"/> <input type="text"/> High × 15 = <input type="text"/>		<input type="text"/>
External Interface files (EIFs)	<input type="text"/> Low × 5 = <input type="text"/> <input type="text"/> Average × 7 = <input type="text"/> <input type="text"/> High × 10 = <input type="text"/>		<input type="text"/>
Total Unadjusted Function Point Count			<input type="text"/>

The weighting factors are identified (as per Table 4.1) for all functional units and multiplied with the functional units accordingly. The procedure for the calculation of Unadjusted Function Point (UFP) is given in Table 4.2.

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^5 \sum_{j=1}^3 Z_{ij} w_{ij}$$

where i indicates the row and j indicates the column of Table 4.1.

w_{ij} : It is the entry of the i^{th} row and j^{th} column of the Table 4.1.

Z_{ij} : It is the count of the number of functional units of Type i that have been classified as having the complexity corresponding to column j .

Organisations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of complexity is somewhat subjective.

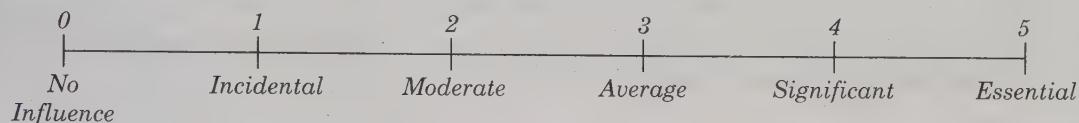
The final number of function points is arrived at by multiplying the UFP by an adjustment factor that is determined by considering 14 aspects of processing complexity which are given in Table 4.3. This adjustment factor allows the UFP count to be modified by at most $\pm 35\%$. The final adjusted FP count is obtained by using the following relationship

$$FP = UFP * CAF$$

Where CAF is complexity adjustment factor and is equal to $[0.65 + 0.01 \times \sum F_i]$. The F_i ($i = 1$ to 14) are the degrees of influence and are based on responses to questions noted in Table 4.3.

Table 4.3: Computing function points.

Rate each factor on a scale of 0 to 5.



Number of factors considered (F_i)

1. Does the system require reliable backup and recovery?
2. Is data communication required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing heavily utilized operational environment?
6. Does the system require on line data entry?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on line?
9. Is the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Uses of function points

The collection of function point data has two primary motivations. One is the desire by managers to monitor levels of productivity, for example, number of function points achieved per work hour expended. From this perspective, the manager is not concerned with when the function point counts are made, but only that the function points accurately describe the size of the final software project. In this instance, function points have an advantage over LOC in that they provide a more objective measure of software size by which to assess productivity.

Another use of function points is in the estimation of software development cost. There are only a few studies that address this issue, though it is arguably the most important potential use of function point data.

Functions points may compute the following important metrics:

Productivity	=	FP/persons-months
Quality	=	Defects/FP
Cost	=	Rupees /FP
Documentation	=	Pages of documentation per FP

These metrics are controversial and are not universally acceptable. There are standards issued by the International Function Point User Group (IFPUG, covering the Albrecht method) and the United Kingdom Function Point User Group (UFPGU, covering the MK11 method). An ISO standard for function point methods is also being developed.

The function point method continues to be refined. So if we intend to use it we should obtain copies of the latest international function point user group (IFPUG) guidelines and standards. IFPUG is the fastest growing non profit software metrics user group in the world with an annual growth rate of 30%. Today it has grown to over 800 corporate members and over 1500 individual members in more than 50 countries.

Example 4.1

Consider a project with the following functional units:

Number of user inputs	= 50
Number of user outputs	= 40
Number of user enquiries	= 35
Number of user files	= 06
Number of external interfaces	= 04

Assume all complexity adjustment factors and weighting factors are average.

Compute the function points for the project.

Solution

We know

$$\text{UFP} = \sum_{i=1}^5 \sum_{j=1}^3 Z_{ij} w_{ij}$$

$$\begin{aligned} \text{UFP} &= 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7 \\ &= 200 + 200 + 140 + 60 + 28 = 628 \\ \text{CAF} &= (0.65 + 0.01 \sum F_i) \\ &= (0.65 + 0.01(14 \times 3)) = 0.65 + 0.42 = 1.07 \\ \text{FP} &= \text{UFP} \times \text{CAF} \\ &= 628 \times 1.07 = 672. \end{aligned}$$

Example 4.2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average external inquiries, and a value of complexity adjustment factor of 1.10.

What are the unadjusted and adjusted function point counts ?

Solution

Unadjusted function point counts may be calculated using as:

$$\begin{aligned}
 \text{UFP} &= \sum_{i=1}^5 \sum_{j=1}^3 Z_{ij} w_{ij} \\
 &= 10 \times 3 + 12 \times 7 + 20 \times 7 + 15 + 10 + 12 \times 4 \\
 &= 30 + 84 + 140 + 150 + 48 \\
 &= 452 \\
 \text{FP} &= \text{UFP} \times \text{CAF} \\
 &= 452 \times 1.10 = 497.2.
 \end{aligned}$$

Example 4.3

Consider a project with the following parameters.

(i) External Inputs:

- (a) 10 with low complexity
- (b) 15 with average complexity
- (c) 17 with high complexity.

(ii) External Outputs:

- (a) 6 with low complexity
- (b) 13 with high complexity.

(iii) External Inquiries:

- (a) 3 with low complexity
- (b) 4 with average complexity
- (c) 2 with high complexity.

(iv) Internal logical files:

- (a) 2 with average complexity
- (b) 1 with high complexity.

(v) External Interface files:

- (a) 9 with low complexity.

In addition to above, system requires

- (i) Significant data communication
- (ii) Performance is very critical
- (iii) Designed code may be moderately reusable
- (iv) System is not designed for multiple installations in different organisations.

Other complexity adjustment factors are treated as average. Compute the function points for the project.

Solution

Unadjusted function points may be counted using Table 4.2.

<i>Functional Units</i>	<i>Count</i>	<i>Complexity</i>			<i>Complexity Totals</i>	<i>Functional Unit totals</i>
External	10	Low	× 3	=	30	
Inputs (EIs)	15	Average	× 4	=	60	
	17	High	× 6	=	102	192
External	6	Low	× 4	=	24	
Outputs (EOs)	0	Average	× 5	=	0	
	13	High	× 7	=	91	115
External	3	Low	× 3	=	9	
Inquiries (EIs)	4	Average	× 4	=	16	
	2	High	× 6	=	12	37
Internal	0	Low	× 7	=	0	
Logical	2	Average	× 10	=	20	
Files (ILFs)	1	High	× 15	=	15	35
External	9	Low	× 5	=	45	
Interface	0	Average	× 7	=	0	
Files (EIFs)	0	High	× 10	=	0	45
Total unadjusted function point count =					424	

The factors given in Table 4.3 may be calculated as:

$$\sum_{i=1}^{14} F_i = 3 + 4 + 3 + 5 + 3 + 3 + 3 + 3 + 3 + 3 + 2 + 3 + 0 + 3 = 41$$

$$\begin{aligned} CAF &= (0.65 + 0.01 \times \Sigma F_i) \\ &= (0.65 \times 0.01 \times 41) \\ &= 1.06 \end{aligned}$$

$$\begin{aligned} FP &= UFP \times CAF \\ &= 424 \times 1.06 \\ &= 449.44 \end{aligned}$$

Hence

$$FP = 449$$

4.2 COST ESTIMATION

For any new software project, it is necessary to know how much will it cost to develop and how much development time will it take. These estimates are needed before development is initiated. But how is this done? In many cases estimates are made using past experience as the only guide. However, in most of the cases projects are different and hence past experience alone may not be enough. A number of estimation techniques have been developed and are having following attributes in common.

- Project scope must be established in advance
- Software metrics are used as a basis from which estimates are made
- The project is broken into small pieces which are estimated individually

To achieve reliable cost and schedule estimates, a number of options arise:

- Delay estimation until late in project (obviously we can achieve 100% accurate estimates after project is complete!)
- Use simple decomposition techniques to generate project cost and schedule estimates
- Develop empirical models for estimation
- Acquire one or more automated estimation tools

Unfortunately, the first option, however, attractive, is not practical. Cost estimates must be provided up front. However, we should recognize that the longer we wait, more we know, and more we know, the less likely, are we to make serious errors in our estimates [PRES2K]

4.3 MODELS

The model is concerned with the representation of the process to be estimated. A model may be static or dynamic. In a static model, a unique variable (say, size) is taken as a key element for calculating all others (say, cost, time). The form of equation used is the same for all calculations. In a dynamic model, all variables are interdependent and there is no basic variable as in the static model.

When a model makes use of a single basic variable to calculate all others it is said to be a *single-variable* model. In some models, several variables are needed to describe the software development process, and selected equations combine these variables to give the estimate of time and cost. These models are called *multivariable*. The variables, single or multiple, that are input to the model to predict the behaviour of a software development are called *predictors*. The choice and handling of these predictors are most crucial activity in estimating methodology [LOND87].

4.3.1 Static, Single Variable Models

Methods using this model use an equation to estimate the desired values such as cost, time, effort, etc. They all depend on the same variable used as predictor (say, size). An example of the most common equation is

$$C = a L^b \quad (4.1)$$

where C is the cost (effort expressed in any unit of manpower, for example, person-months) and L is the size generally given in the number of lines of code. The constants, a & b are

derived from the historical data of the organization. Since a and b depend on the local development environment, these models are not transportable to different organizations.

The Software Engineering Laboratory of the University of Maryland has established a model, the SEL model, for estimating its own software productions. This model [BASL80] is a typical example of a static single-variable model.

$$E = 1.4 L^{0.93} \quad (4.2)$$

$$DOC = 30.4 L^{0.90} \quad (4.3)$$

$$D = 4.6 L^{0.26} \quad (4.4)$$

Effort (E in Person-months), documentation (DOC, in number of pages) and duration (D, in months) are calculated from the number of lines of code (L, in thousands of lines) used as a predictor.

4.3.2 Static, Multivariable Models

Although these models are often based on equation (4.1), they actually depend on several variables representing various aspects of the software development environment, for example, methods used, user participation, customer oriented changes, memory constraints, etc. The model developed by Walston and Felix at IBM [WALS77] provides a relationship between delivered lines of source code (L in thousands of lines) and effort E (E in person-months) and is given by the following equation:

$$E = 5.2 L^{0.91} \quad (4.5)$$

In the same fashion, the duration of the development (D in months) is given by

$$D = 4.1 L^{0.36} \quad (4.6)$$

Data collected on 60 software projects, representing a wide variety of applications and size (ranging from 4000 to 467000 lines of code), shows a relationship between productivity (expressed in number of lines of source code per person months) and a productivity index I.

The productivity index uses 29 variables which are found to be highly correlated to productivity as follows:

$$I = \sum_{i=1}^{29} W_i X_i \quad (4.7)$$

where W_i is a factor weight for the i^{th} variable and $X_i = \{-1, 0, +1\}$. The estimator gives X_i one of the values -1, 0 or +1 depending on whether the variable decreases, has no effect, or increases the productivity respectively. The terms of equation (4.7) are then added up to give the productivity index. A productivity range can be obtained for the project by using a productivity versus index chart [LOND87].

Example 4.4

Compare the Walston-Felix model [equation (4.5) and equation (4.6)] with the SEL model [equation (4.2) and equation (4.4)] on a software development expected to involve 8 person-years of effort [LOND87].

- (a) Calculate the number of lines of source code that can be produced.
- (b) Calculate the duration of the development.

- (c) Calculate the productivity in LOC/PY.
- (d) Calculate the average manning.

Solution

The amount of manpower involved = 8 PY = 96 person-months

- (a) Number of lines of source code can be obtained by reversing equation (4.2) and equation (4.5) to give:

$$L = (E/a)^{1/b}$$

Then

$$L(SEL) = (96/1.4)^{1/0.93} = 94264 \text{ LOC}$$

$$L(W-F) = (96/5.2)^{1/0.91} = 24632 \text{ LOC}$$

- (b) Duration in months can be calculated by means of equation (4.4) and equation (4.6)

$$\begin{aligned} D(SEL) &= 4.6 (94.264)^{0.26} \\ &= 4.6(94.264)^{0.26} = 15 \text{ months} \end{aligned}$$

$$\begin{aligned} D(W-F) &= 4.1 L^{0.36} \\ &= 4.1 (24.632)^{0.36} = 13 \text{ months} \end{aligned}$$

- (c) Productivity is the lines of code produced per person/month (year).

$$P(SEL) = \frac{94264}{8} = 11783 \text{ LOC/Person-Years}$$

$$P(W-F) = \frac{24632}{8} = 3079 \text{ LOC/Person-Years}$$

- (d) Average manning is the average number of persons required per month in the project.

$$M(SEL) = \frac{96 \text{ P-M}}{15 \text{ M}} = 6.4 \text{ Persons}$$

$$M(W-F) = \frac{96 \text{ P-M}}{13 \text{ M}} = 7.4 \text{ Persons}$$

If we look at the value of "L", it seems that SEL can produce four times as much software as IBM for the same manpower and time scale.

4.4 THE CONSTRUCTIVE COST MODEL (COCOMO)

This model gained rapid popularity following the publication of B.W. Boehm's excellent book *Software Engineering Economics* in 1981 [BOEH81]. COCOMO is a hierarchy of software cost estimation models, which include basic, intermediate and detailed sub models.

4.4.1 Basic Model

The basic model aims at estimating, in a quick and rough fashion, most of the small to medium sized software projects. Three modes of software development are considered in this model: organic, semi-detached and embedded.

In the organic mode, a small team of experienced developers develops software in a very familiar environment. The size of the software development in this mode ranges from small

(a few KLOC) to medium (a few tens of KLOC), while in other two modes the size ranges from small to very large (a few hundreds of KLOC).

In the embedded mode of software development, the project has tight constraints, which might be related to the target processor and its interface with the associated hardware. The problem to be solved is unique and so it is often hard to find experienced persons, as the same does not usually exist.

The *semi detached* mode is an intermediate mode between the organic mode and embedded mode. The comparison of all three modes is given in Table 4.4.

Table 4.4: The comparison of three COCOMO modes

Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2 – 50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar projects. For Example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For Example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

Depending on the problem at hand, the team might include a mixture of experienced and less experienced people with only a recent history of working together. The basic COCOMO equations take the form

$$E = a_b (\text{KLOC})^{b_b} \quad (4.8)$$

$$D = c_b (E)^{d_b} \quad (4.9)$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b , b_b , c_b and d_b are given in Table 4.4(a).

Table 4.4(a): Basic COCOMO coefficients

Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size (SS)} = \frac{E}{D} \text{ Persons.}$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity (P)} = \frac{\text{KLOC}}{E} \text{ KLOC/PM.}$$

With the basic model, the software estimator has a useful tool for estimating quickly, by two runs on a pocket calculator, the cost and development time of a software project, once the size is estimated. The software estimator will have to assess by himself/herself which mode is the most appropriate.

Example 4.5

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes *i.e.*, organic, semidetached and embedded.

Solution

The basic COCOMO equations take the form:

$$E = a_b (\text{KLOC})^{b_b}$$

$$D = c_b (\text{KLOC})^{d_b}$$

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ M}$$

(ii) Semidetached mode

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ M}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ M}$$

As we have seen, effort calculated for embedded mode is approximately 4 times, the effort for organic mode. However, the effort calculated for semidetached mode is 2 times the effort of organic mode. There is a large difference in these values. But, surprisingly, the development time is approximately the same for all three modes. It is clear from here that the

selection of mode is very important. Since development time is approximately the same, the only varying parameter is the requirement of persons. Every mode will have different manpower requirement. If we look to the Table 4.4, it is mentioned that for over 300 KLOC projects, embedded mode is the right choice. The selection of a mode is not only dependent on project size, but also on other parameters as mentioned in Table 4.4. We should be utmost careful about the selection of mode for the project.

Example 4.6

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

Solution

The semi-detached mode is the most appropriate mode; keeping in view the size, schedules and experience of the development team.

$$\text{Hence } E = 3.0(200)^{1.12} = 1133.12 \text{ PM}$$

$$D = 2.5(1133.12)^{.35} = 29.3 \text{ M}$$

$$\begin{aligned} \text{Average staff size (SS)} &= \frac{E}{D} \text{ Persons} \\ &= \frac{1133.12}{29.3} = 38.67 \text{ Persons.} \end{aligned}$$

$$\begin{aligned} \text{Productivity} &= \frac{\text{KLOC}}{E} = \frac{200}{1133.12} = 0.1765 \text{ KLOC/PM} \\ P &= 176 \text{ LOC/PM.} \end{aligned}$$

4.4.2 Intermediate Model

The basic model allowed for a quick and rough estimate, but it resulted in a lack of accuracy. Boehm introduced an additional set of 15 predictors called cost drivers in the intermediate model to take account of the software development environment. Cost drivers are used to adjust the nominal cost of a project to the actual project environment, hence increasing the accuracy of the estimate.

The cost drivers are grouped into four categories:

1. Product attributes
 - (a) Required software reliability (RELY)
 - (b) Database size (DATA)
 - (c) Product complexity (CPLX)
2. Computer attributes
 - (a) Execution time constraint (TIME)
 - (b) Main storage constraint (STOR)
 - (c) Virtual machine volatility (VIRT)
 - (d) Computer turnaround time (TURN)

3. Personnel attributes

- (a) Analyst capability (ACAP)
- (b) Application experience (AEXP)
- (c) Programmer capability (PCAP)
- (d) Virtual machine experience (VEXP)
- (e) Programming language experience (LEXP)

4. Project attributes

- (a) Modern programming practices (MODP)
- (b) Use of software tools (TOOL)
- (c) Required development schedule (SCED)

Each cost driver is rated for a given project environment. The rating uses a scale very low, low, nominal, high, very high, extra high which describes to what extent the cost driver applies to the project being estimated. Table 4.5 gives the multiplier values for the 15 cost drivers and their rating as provided by Boehm [BOEH81].

Table 4.5: Multiplier values for effort calculations

<i>Cost Drivers</i>	<i>RATINGS</i>					
	<i>Very low</i>	<i>Low</i>	<i>Nominal</i>	<i>High</i>	<i>Very high</i>	<i>Extra high</i>
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	—
DATA	—	0.94	1.00	1.08	1.16	—
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	—	—	1.00	1.11	1.30	1.66
STOR	—	—	1.00	1.06	1.21	1.56
VIRT	—	0.87	1.00	1.15	1.30	—
TURN	—	0.87	1.00	1.07	1.15	—
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	—
AEXP	1.29	1.13	1.00	0.91	0.82	—
PCAP	1.42	1.17	1.00	0.86	0.70	—
VEXP	1.21	1.10	1.00	0.90	—	—
LEXP	1.14	1.07	1.00	0.95	—	—
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	—
TOOL	1.24	1.10	1.00	0.91	0.83	—
SCED	1.23	1.08	1.00	1.04	1.10	—

The multiplying factors for all 15 cost drivers are multiplied to get the effort adjustment factor (EAF). Typical values for EAF range from 0.9 to 1.4.

The intermediate COCOMO equations take the form:

$$E = a_i(\text{KLOC})^{b_i} \quad (3.10)$$

$$D = c_i(E)^{d_i} \quad (3.11)$$

The coefficients a_i , b_i , c_i and d_i are given in Table 4.6.

Table 4.6: Coefficients for intermediate COCOMO

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

4.4.3 Detailed COCOMO Model

A large amount of work has been done by Boehm to capture all significant aspects of a software development. It offers a means for processing all the project characteristics to construct a software estimate. The detailed model introduces two more capabilities:

1. Phase-sensitive effort multipliers:

Some phases (design, programming, integration/test) are more affected than others by factors defined by the cost drivers. The detailed model provides a set of phase sensitive effort multipliers for each cost driver. This helps in determining the manpower allocation for each phase of the project.

2. Three-level product hierarchy:

Three product levels are defined. These are module, subsystem and system levels. The ratings of the cost drivers are done at appropriate level; that is, the level at which it is most susceptible to variation.

Development phases

A software development is carried out in four successive phases: plans/requirements, product design, programming and integration/test.

1. **Plan/requirements:** This is the first phase of the development cycle. The requirement is analyzed, the product plan is set up and a full product specification is generated. This phase consumes from 6% to 8% of the effort and 10% to 40% of the development time. These percentages depend not only on mode (organic, semi-detached or embedded), but also on the size.
2. **Product design:** The second phase of the COCOMO development cycle is concerned with the determination of the product architecture and the specification of the subsystem. This phase requires from 16% to 18% the nominal effort and can last from 19% to 38% of the development time.

3. **Programming:** The third phase of the COCOMO development cycle is divided into two sub phases: detailed design and code/unit test. This phase requires from 48% to 68% of the effort and lasts from 24% to 64% of the development time.
4. **Integration/Test:** This phase of the COCOMO development cycle occurs before delivery. This mainly consists of putting the tested parts together and then testing the final product. This phase requires from 16% to 34% of the nominal effort and can last from 18% to 34% of the development time.

Principle of the effort estimate

Size equivalent: As the software might be partly developed from software already existing (that is, re-usable code), a full development is not always required. In such cases, the parts of design document (DD%), code (C%) and integration (I%) to be modified are estimated. Then, an adjustment factor, A, is calculated by means of the following equation.

$$A = 0.4 \text{ DD} + 0.3 \text{ C} + 0.3 \text{ I} \quad (4.12)$$

The size equivalent is obtained by

$$S(\text{equivalent}) = (S \times A)/100 \quad (4.13)$$

where S represents the thousands of lines of code (KLOC) of the module. Multipliers have been developed that can be applied to the total project effort, E, and total project development time, D in order to allocate effort and schedule components to each phase in the life cycle of a software development program. There are assumed to be five distinct life cycle phases, and the effort and schedule for each phase are assumed to be given in terms of the overall effort and schedule by

$$E_p = \mu_p E \quad (4.14)$$

$$D_p = \tau_p D \quad (4.15)$$

where μ_p and τ_p are given in Table 4.7. There exist more sophisticated versions of this development that result in multipliers μ_p and τ_p that not only depend on the particular phase of the life cycle and mode of operation of the software but also contain the correction terms for the 15 attributes [BOEH81].

The COCOMO model is certainly the most thoroughly documented model currently available. It is very easy to use. And by doing so, the software manager can learn a lot about productivity, particularly from the very clear presentation of the cost drivers. The size and cost drivers can be progressively adjusted to realistic values to some extent. However, mode choice offers some difficulties, since it is not always possible to be sure which of the three modes is appropriate for a given software development—it might be a mixed mode.

Table 4.7: Effort and schedule fractions occurring in each phase of the lifecycle [SAGE90]

Mode & Code Size	Plan & Requirement	System Design	Detail Design	Module Code & Test	Integration and Test
Lifecycle Phase Value of μ_p					
Organic Small S≈2	0.06	0.16	0.26	0.42	0.16
Organic Medium S≈32	0.06	0.16	0.24	0.38	0.22
Semidetached Medium S≈32	0.07	0.17	0.25	0.33	0.25
Semidetached Large S≈128	0.07	0.17	0.24	0.31	0.28
Embedded Large S≈128	0.08	0.18	0.25	0.26	0.31
Embedded Extra Large S≈320	0.08	0.18	0.24	0.24	0.34
Lifecycle Phase Value of τ_p					
Organic Small S≈2	0.10	0.19	0.24	0.39	0.18
Organic Medium S≈32	0.12	0.19	0.21	0.34	0.26
Semidetached Medium S≈32	0.20	0.26	0.21	0.27	0.26
Semidetached Large S≈128	0.22	0.27	0.19	0.25	0.29
Embedded Large S≈128	0.36	0.36	0.18	0.18	0.28
Embedded Extra Large S≈320	0.40	0.38	0.16	0.16	0.30

There are five phases of software life cycle in Table 4.7. However, plan and requirement phase has been combined with system design and known as requirement and product design. Both include the most conceptual part of the life cycle. So the effort and time shown in Table 4.7 for plan and requirements phase are over and above the estimated effort and time. The actual distribution may start from system design phase. Hence, four phases are:

1. Requirement and product design
 - (a) Plans and requirements
 - (b) System design
2. Detailed Design
 - (a) Detailed design
3. Code & Unit test
 - (a) Module code & test
4. Integrate and Test
 - (a) Integrate & Test.

COCOMO is highly calibrated model, based on previous experience. It is easy to use and documented properly. Actual data gathered from previous projects may help to determine the values of the constants of the model (like a , b , c and d). These values may vary from organisation to organisation. However, this model ignores software safety & security issues. It also ignores many hardware and customer related issues. It is silent about the involvement and responsiveness of customer.

It does not give proper importance to software requirements and specification phase which has identified as the most sensitive phase of software development life cycle.

Example 4.7

A new project with estimated 400 KLOC embedded system has to be developed. Project manager has a choice of hiring from two pools of developers: Very highly capable with very little experience in the programming language being used or developers of low quality but a lot of experience with the programming language. What is the impact of hiring all developers from one or the other pool ?

Solution

This is the case of embedded mode and model is intermediate COCOMO.

$$\begin{aligned}\text{Hence } E &= \alpha_i (\text{KLOC})^{d_i} \\ &= 2.8(400)^{1.20} = 3712 \text{ PM}\end{aligned}$$

Case I: Developers are very highly capable with very little experience in the programming being used.

$$\begin{aligned}EAF &= 0.82 \times 1.14 = 0.9348 \\ E &= 3712 \times 0.9348 = 3470 \text{ PM} \\ D &= 2.5(3470)^{0.32} = 33.9 \text{ PM.}\end{aligned}$$

Case II: Developers are of low quality but lot of experience with the programming language being used.

$$\begin{aligned}EAF &= 1.29 \times 0.95 = 1.22 \\ E &= 3712 \times 1.22 = 4528 \text{ PM} \\ D &= 2.5(4528)^{0.32} = 36.9 \text{ PM.}\end{aligned}$$

Case II requires more effort and time. Hence, low quality developers with lot of programming language experience could not match with the performance of very highly capable developers with very little experience.

Example 4.8

Consider a project to develop a full screen editor. The major components identified are (1) Screen Edit (2) Command language Interpreter (3) File input and output, (4) Cursor movement and (5) Screen movement. The sizes for these are estimated to be 4K, 2K, 1K, 2K and 3K delivered source code lines. Use COCOMO model to determine:

- (a) Overall cost and schedule estimates (assume values for different cost drivers, with at least three of them being different from 1.0).
- (b) Cost and Schedule estimates for different phases.

Solution

Size of five modules are:

Screen edit	= 4 KLOC
Command language interpreter	= 2 KLOC
File input and output	= 1 KLOC
Cursor movement	= 2 KLOC

Screen movement	= 3 KLOC
Total	= 12 KLOC

Let us assume that significant cost drivers are

- (i) Required software reliability is high, i.e., 1.15
- (ii) Product complexity is high, i.e., 1.15
- (iii) Analyst capability is high, i.e., 0.86
- (iv) Programming language experience is low, i.e., 1.07
- (v) All other drivers are nominal.

$$\text{EAF} = 1.15 \times 1.15 \times 0.86 \times 1.07 = 1.2169$$

(a) The initial effort estimate for the project is obtained from the following equation

$$\begin{aligned} E &= a_i(\text{KLOC})^{b_i} \times \text{EAF} \\ &= 3.2(12)^{1.05} \times 1.2169 = 52.91 \text{ PM} \end{aligned}$$

$$\begin{aligned} \text{Development time } D &= C_i(E)^{d_i} \\ &= 2.5 (52.91)^{0.38} = 11.29 \text{ M} \end{aligned}$$

(b) Using the following equations and referring Table 4.7, phase wise cost and schedule estimates can be calculated.

$$\begin{aligned} E_p &= \mu_p E \\ D_p &= \tau_p D \end{aligned}$$

Since size is only 12 KLOC, it is an organic small model. Phase wise effort distribution is given below:

System Design	= 0.16 × 52.91 = 8.465 PM
Detailed Design	= 0.26 × 52.91 = 13.756 PM
Module Code & Test	= 0.42 × 52.91 = 22.222 PM
Integration & Test	= 0.16 × 52.91 = 8.465 PM

Now Phase wise development time duration is

System Design	= 0.19 × 11.29 = 2.145 M
Detailed Design	= 0.24 × 11.29 = 2.709 M
Module Code & Test	= 0.39 × 11.29 = 4.403 M
Integration & Test	= 0.18 × 11.29 = 2.032 M

4.5 COCOMO-II

COCOMO-II is the revised version of the original COCOMO (discussed in article 4.4) and is developed at University of Southern California under the leadership of Dr. Barry Boehm. The model is tuned to the life cycle practices of the 21st century. It also provides a quantitative analytic framework, and set of tools and techniques for evaluating the effects of software technology improvements on software life cycle costs and schedules. The following categories of applications/projects are identified by COCOMO-II for the estimation [UCSD01] and are shown in Fig. 4.4.

(i) **End User Programming:** This category is applicable to small systems, developed by end user using application generators. Some application generators are spreadsheets, extended query system, report generators etc. End user may write small programs using application generators. The end users may not have sufficient knowledge about computers and software engineering practices. However, they may have in-depth knowledge about their business needs and practices. Hence, this excellent domain knowledge may motivate them to develop an application using user friendly tools like MS-Excel, MS-Access, MS-Studio etc.

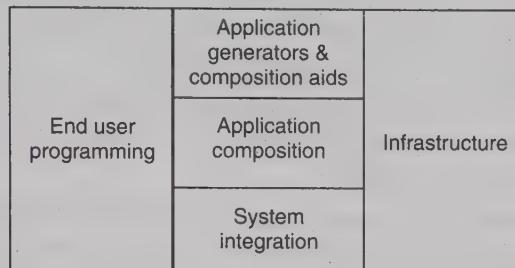


Fig. 4.4: Categories of applications/projects.

(ii) **Infrastructure Sector:** This category is applicable to the infrastructure development i.e., the software that provides infrastructure like operating systems, database management systems, user interface management system, networking system etc. Some commercial examples are Microsoft products, Oracle, DB2, MAYA, LINUX, 3D-STUDIO.

Infrastructure developers generally have good knowledge of software development and software engineering practices and relatively little knowledge about applications. The product lines will have many reusable components, but the pace of technology (new processor, memory, communications, display, and multimedia technology) will require them to build many components and capabilities from scratch.

(iii) **Intermediate Sectors:** This category is partitioned in three sub categories as shown in Fig. 4.4. Software developers will need to have good knowledge of software development and software engineering practices, experience and expertise of infrastructure software and indepth domain knowledge of one or more applications. Creating this talent pool is a major challenge.

Application generators and composition aids : This subcategory will create largely prepackaged capabilities for user programming. Typical firms operating in this sector are Microsoft, Lotus, Novell, Borland, Alias Wavefront, Oracle, IBM. Their product lines will have many reusable components, but also will require a good deal of new capability development from the scratch. Application composition aids will be developed both by the firms above and by software product line investments of firms in the application composition sector.

Application composition sector : This subcategory deals with applications which are too diversified to be handled by prepackaged solutions, but which are sufficiently simple to be rapidly composable from interoperable components. Typical components will be graphic user interface (GUI) builders, databases or object managers, domain specific components such as financial, medical, or industrial process control packages etc.

These applications are complex, large, versatile, diversified and require specialised developers with sound knowledge of development and software engineering practices.

However, they are developed using application generator environment like CASE tools, DBMS (DB2, oracle etc.), and 4GL programming tools (Developer 2000, Visual basic, Powerbuilder, ASP, JSP, PHP etc).

System integration : This subcategory deals with large scale, highly embedded, or unprecedented systems. Portions of these systems can be developed with application composition capabilities, but their demands generally require a significant amount of upfront systems engineering and customised software development activities.

Stages of COCOMO-II : The end user programming sector does not need a COCOMO-II model. Its applications are normally developed in hours to days. Hence a simple activity based estimate will generally be sufficient.

COCOMO-II includes three stages. Stage I supports estimation of prototyping or application composition types of projects. Stage II supports estimation in the early design stage of a project, when less is known about the project's cost drivers. Stage III supports estimation in the Post-Architecture stage of a project. The details are given in Table 4.8.

Table 4.8: Stages of COCOMO-II

Stage No.	Model Name	Applicable for the types of projects	Applications
Stage I	Application composition estimation model	Application composition	In addition to application composition type of projects, this model is also used for prototyping (if any) stage of application generators, infrastructure & system integration.
Stage II	Early design estimation model	Application generators, infrastructure & system integration.	Used in early design stage of a project, when less is known about the project.
Stage III	Post architecture estimation model	Application generators, infrastructure & system integration	Used after the completion of the detailed architecture of the project

4.5.1 Application Composition Estimation Model

The model is designed for quickly developed applications using interoperable components. Example of these components based systems are Graphic User Interface (GUI) builders, database or object managers, hypermedia handlers, smart data finders, and domain specific components such as financial, medical, or industrial process control packages. The model can also be used for the prototyping phase of application generator development, infrastructure sector and system integration projects.

In this model, size is first estimated using object points. The object points are easy to identify and count. The object in object points defines screens, reports, and 3GL modules as objects. This may or may not have any relationship to other definitions of “objects”, such as those processing features like class affiliation, inheritance, encapsulation, message passing, and so forth [UCSDOI].

Object point estimation is a relatively new size estimation technique, but it is well suited in application composition sector. It is also a good match to associated prototyping efforts, based on the use of a rapid composition Integrated Computer Aided Software Engineering (ICASE) Environment providing graphic user interface builders, software development tools, and large, composable infrastructure and applications components. The steps required for the estimation of effort in Person-months are given in Fig. 4.5.

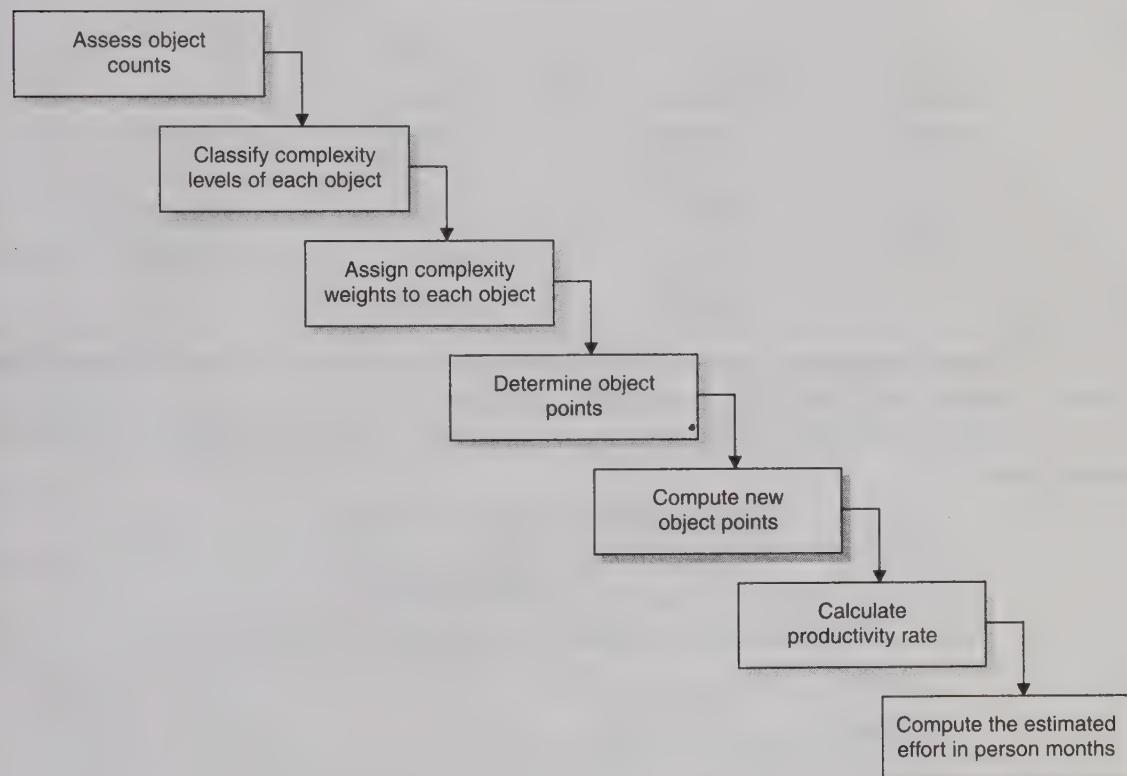


Fig. 4.5: Steps for the estimation of effort in person months

(i) **Assess object counts:** Estimate the number of screens, reports, and 3GL components that will comprise this application.

(ii) **Classification of complexity levels:** We have to classify each object instance into simple, medium and difficult complexity levels depending on values of its characteristics.

The screens are classified on the basis of number of views and sources and reports are on the basis of number of sections and sources. The details are given in Table 4.9.

Table 4.9(a): For screens

Number of views contained	# and sources of data tables		
	Total < 4 (< 2 server < 3 client)	Total < 8 (2 – 3 server 3 – 5 client)	Total 8 + (> 3 server, > 5 client)
< 3	Simple	Simple	Medium
3 – 7	Simple	Medium	Difficult
> 8	Medium	Difficult	Difficult

Table 4.9(b): For reports

Number of sections contained	# and sources of data tables		
	Total < 4 (< 2 server < 3 client)	Total < 8 (2 – 3 server 3 – 5 client)	Total 8 + (> 3 server, > 5 client)
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 +	Medium	Difficult	Difficult

(iii) Assign complexity weight to each object: The weights are used for three object types i.e., screen, report and 3GL components using the Table 4.10.

The weights reflect the relative effort required to implement an instance of that complexity level.

Table 4.10: Complexity weights for each level

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Component	—	—	10

(iv) Determine object points: Add all the weighted object instances to get one number and this number is known as object-point count.

(v) Compute new object points: We have to estimate the percentage of reuse to be achieved in a project. Depending on the percentage reuse, the new object points (NOP) are computed.

$$\text{NOP} = \frac{(\text{object points}) * (100 - \% \text{ reuse})}{100}$$

NOPs are the object points that will need to be developed and differ from the object point count because there may be reuse.

(vi) **Calculation of productivity rate:** The productivity rate can be calculated as:

$$\text{Productivity rate (PROD)} = \text{NOP}/\text{Person month}.$$

PROD values Calculation requires NOP and total person-months of past projects in similar environments. COCOMO-II application composition model gives the following Table 4.11 containing the values of PROD based on their data and experience.

Table 4.11: Productivity values

<i>Developer's experience & capability; ICASE maturity & capability</i>	<i>PROD (NOP/PM)</i>
Very low	4
Low	7
Nominal	13
High	25
Very high	50

(vii) **Compute the effort in Person-Months:** When PROD is known, we may estimate effort in Person-Months as:

$$\text{Effort in PM} = \frac{\text{NOP}}{\text{PROD}}$$

Example 4.9

Consider a database application project with the following characteristics:

- (i) The application has 4 screens with 4 views each and 7 data tables for 3 servers and 4 clients.
- (ii) The application may generate two report of 6 sections each from 07 data tables for two server and 3 clients. There is 10% reuse of object points.

The developer's experience and capability in the similar environment is low. The maturity of organisation in terms of capability is also low. Calculate the object point count, New object points and effort to develop such a project.

Solution

This project comes under the category of application composition estimation model.

Number of screens = 4 with 4 views each

Number of reports = 2 with 6 sections each.

From Table 4.9, we know that each screen will be of medium complexity and each report will be of difficult complexity.

Using Table 4.10 of complexity weights, we may calculate object point count

$$= 4 \times 2 + 2 \times 8 = 24$$

$$\text{NOP} = \frac{24 * (100 - 10)}{100} = 21.6$$

Table 4.11 gives the low value of productivity (PROD) i.e., 7.

$$\text{Efforts in PM} = \frac{\text{NOP}}{\text{PROD}}$$

$$\text{Effort} = \frac{21.6}{7} = 3.086 \text{ PM}$$

4.5.2 The Early Design Model

The COCOMO-II models use the base equation of the form

$$\text{PM}_{\text{nominal}} = A * (\text{size})^B$$

where $\text{PM}_{\text{nominal}}$ = Effort of the project in person months.

A = Constant representing the nominal productivity, provisionally set to 2.5

B = Scale factor

Size = Software size

The early design model uses Unadjusted Function Points (UFP) as the measure of size. This model is used in the early stages of a software project when very little may be known about the size of the product to be developed, the nature of the target platform, the nature of the personnel to be involved in the project or the detailed specifics of the process to be used. This model can be used in either Application Generator, System Integration, or Infrastructure Development Sector.

If $B = 1.0$, there is a linear relationship of effort and size. If value of B is not 1, there will be a non linear relationship between size and effort. If $B < 1.0$, the rate of increase of effort decreases as the size of the product increases. If the product's size is doubled, the project effort is less than doubled.

If $B > 1.0$, the rate of increase of effort increases as the size of the product increases. This is due to the growth of interpersonal communications overheads and growth of large system integration overhead. Application composition model assumes the value of B to be 1. But the early design model assumes the value of B to be greater than 1. Thus, the basic assumption is that the effort spent in a project usually increases faster than the size of the project. The value of B is computed on the basis of scaling factors (or drivers) that may cause drop in productivity with increase in size.

Table 4.12: Scaling factors required for the calculation of the value of B

Scale factor	Explanation	Remarks
Precedentness	Reflects the previous experience on similar projects. This is applicable to individuals & organisation both in terms of expertise & experience.	Very low means no previous experiences, Extra high means that organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process.	Very low means a well defined process is used. Extra high means that the client gives only general goals.
Architecture/Risk resolution	Reflects the degree of risk analysis carried out.	Very low means very little analysis and Extra high means complete and thorough risk analysis.

(Contd...)

Team cohesion	Reflects the team management skills.	Very low means very little interaction & hardly any relationship among team members; Extra high means an integrated & effective team.
Process maturity	Reflects the process maturity of the organisation. Thus it is dependent on SEI-CMM level of the organisation.	Very low means organisation has no level at all and extra high means organisation is rated as highest level of SEI-CMM.

The scaling factors that COCOMO-II uses for the calculation of B are Precedentness, Development Flexibility, Architecture/Risk Resolution, Team Cohesion and Process Maturity. The details are given in Table 4.12. These factors are rated on a six point scale i.e., very low, low, nominal, high, very high and extra high and are given in Table 4.13.

Table 4.13: Data for the Computation of B

Scaling factors	Very low	Low	Nominal	High	Very high	Extra high
Precedentness	6.20	4.96	3.72	2.48	1.24	0.00
Development flexibility	5.07	4.05	3.04	2.03	1.01	0.00
Architecture/Risk resolution	7.07	5.65	4.24	2.83	1.41	0.00
Team cohesion	5.48	4.38	3.29	2.19	1.10	0.00
Process maturity	7.80	6.24	4.68	3.12	1.56	0.00

The value of B can be calculated as:

$$B = 0.91 + 0.01 * (\text{Sum of rating on scaling factors for the project})$$

When all the scaling factors of a project are rated as extra high, the best value of B is obtained and is equal to 0.91. When all the scaling factors are very low, the worst value of B is obtained and is equal to 1.23. Hence value of B may vary from 0.91 to 1.23.

Early design cost drivers

There are seven early design cost drivers and are given below:

- (i) Product Reliability and Complexity (RCPX)
- (ii) Required Reuse (RUSE)
- (iii) Platform Difficulty (PDIF)
- (iv) Personnel Capability (PERS)
- (v) Personnel Experience (PREX)
- (vi) Facilities (FCIL)
- (vii) Schedule (SCED)

Post architecture cost drivers

There are 17 cost drivers in the Post Architecture model. These are rated on a scale of 1 to 6 as given below:

<i>Very Low</i>	<i>Low</i>	<i>Nominal</i>	<i>High</i>	<i>Very High</i>	<i>Extra High</i>
1	2	3	4	5	6

The list of seventeen cost drivers is given below:

1. Reliability Required (RELY)
2. Database Size (DATA)
3. Product Complexity (CPLX)
4. Required Reusability (RUSE)
5. Documentation (DOCU)
6. Execution Time Constraint (TIME)
7. Main Storage Constraint (STOR)
8. Platform Volatility (PVOL)
9. Analyst Capability (ACAP)
10. Programmers Capability (PCAP)
11. Personnel Continuity (PCON)
12. Analyst Experience (AEXP)
13. Programmer Experience (PEXP)
14. Language & Tool Experience (LTEX)
15. Use of Software Tools (TOOL)
16. Site Locations & Communication Technology between Sites (SITE)
17. Schedule (SCED)

Mapping of early design cost drivers and post architecture cost drivers

The 17 Post Architecture Cost Drivers are mapped to 7 Early Design Cost Drivers and are given in Table 4.14. This mapping is essential because many parameters will not be known correctly in early design phase. The mapping combines estimated parameters in order to have reasonable view of cost drivers. In Post Architecture, all 17 drivers will be known with reasonable accuracy, hence no mapping is required.

Table 4.14: Mapping table

<i>Early Design Cost drivers</i>	<i>Counter part Combined Post Architecture Cost drivers</i>
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PERS	ACAP, PCAP, PCON
PREX	AEXP, PEXP, LTEX
FCIL	TOOL, SITE
SCED	SCED

Product of cost drivers for early design model

The combined early design cost drivers may be obtained by summing the numerical values of the contributing Post Architecture cost drivers. The resulting totals are allocated to an expanded early design model rating scale from Extra Low to Extra High. The early design model rating scales always have a Nominal total equal to the sum of the Nominal ratings of its contributing Post-Architecture Cost drivers.

(i) **Product Reliability and Complexity (RCPX):** The cost driver combines four Post-Architecture cost drivers which are RELY, DATA, CPLX and DOCU. Here RELY & DOCU range from Very Low to Very High. DATA ranges from Low to Very High; and CPLX ranges from Very Low to Extra High. The numerical sum of their ratings thus ranges from 5(VL, L, VL, VL) to 21(VH, VH, EH, VH). For details please refer to Table 4.16. The RCPX rating levels are given below:

<i>RCPX</i>	<i>Extra Low</i>	<i>Very Low</i>	<i>Low</i>	<i>Nominal</i>	<i>High</i>	<i>Very High</i>	<i>Extra High</i>
Sum of RELY, DATA, CPLX, DOCU ratings	5, 6	7, 8	9-11	12	13-15	16-18	19-21
Emphasis on reliability, documentation	Very Little	Little	Some	Basic	Strong	Very Strong	Extreme
Product complexity	Very Simple	Simple	Some	Moderate	Complex	Very Complex	Extremely Complex
Database size	Small	Small	Small	Moderate	Large	Very Large	Very Large

(ii) **Required Reuse (RUSE):** This early design model cost driver is same as its Post-architecture Counterpart. The RUSE rating levels are (As per Table 4.16):

	Vary Low	Low	Nominal	High	Very High	Extra High
RUSE	1	2	3	4	5	6
		None	Across project	Across program	Across product line	Across multiple product line

(iii) **Platform Difficulty (PDIF):** This cost driver combines TIME, STOR, and PVOL of Post-Architecture cost drivers. From the Table 4.16 it is clear that TIME and STOR range from Nominal to Extra High; PVOL ranges from Low to Very High. The numerical sum of ratings thus ranges from 8(N, N, L) to 17 (EH, EH, VH). Hence PDIF rating levels are:

PDIF	Low	Nominal	High	Very High	Extra High
Sum of Time, STOR & PVOL ratings	8	9	10-12	13-15	16-17
Time & storage constraint	≤ 50%	≤ 50%	65%	80%	90%
Platform Volatility	Very stable	Stable	Somewhat stable	Volatile	Highly Volatile

(iv) **Personnel Capability (PERS):** This cost driver combines three Post-Architecture Cost drivers. These drivers are analyst capability (ACAP), Programmers Capability (PCAP) and Personnel Continuity (PCON). Each of these has a rating scale from Very Low to very High as per Table 4.16. Adding up their numerical ratings produces values ranging from 3 to 15. PERS rating levels are calculated with the help of Table 4.16 and are given below:

PERS	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
Sum of ACAP, PCAP, PCON ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Combined ACAP & PCAP Percentile	20%	39%	45%	55%	65%	75%	85%
Annual Personnel Turnover	45%	30%	20%	12%	9%	5%	4%

(v) **Personnel Experience (PREX):** This early design cost driver combines three Post Architecture Cost drivers, which are: application experience (AEXP), platform experience (PEXP) and language and Tool experience (LTEX). Each of these range from Very Low to Very High and numerical sum of their ratings ranges from 3 to 15. The PREX rating levels are obtained using Table 4.16 and are given below:

PREX	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
Sum of AEXP, PEXP and LTEX ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Applications, Platform, Language & Tool Experience	≤ 3 months	5 months	9 months	1 year	2 year	4 year	6 year

(vi) **Facilities (FCIL):** This depends on two Post Architecture Cost drivers: Use of Software Tools (TOOL) and multisite development (SITE). TOOL ranges from Very Low to Very High; SITE ranges from Very Low to Extra High. Thus the numeric sum of their rating ranges from 2(VL, VL) to 11(VH, EH). FCIL rating levels are obtained using Table 4.16.

FCIL	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
Sum of TOOL & SITE ratings	2	3	4, 5	6	7, 8	9, 10	11
Tool support	Minimal	Some	Simple CASE tools	Basic life cycle tools	Good support of tools	Very strong use of tools	Very strong & well integrated tools
Multisite conditions development support	Weak support of complex multisite development	Some support	Moderate support	Basic support	Strong support	Very strong support	Very strong support

(vii) **Schedule (SCED):** The early design cost driver is the same as Post Architecture Counterpart and rating levels are given below using Table 4.16.

SCED	Very Low	Low	Nominal	High	Very High
Schedule	75% of Nominal	85%	100%	130%	160%

The seven early design cost drivers have been converted into numeric values with a Nominal value 1.0. These values are used for the calculation of a factor called “Effort multiplier” which is the product of all seven early design cost drivers. The numeric values are given in Table 4.15.

Table 4.15: Early design Parameters

Early design Cost drivers	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
RCPX	.73	.81	.98	1.0	1.30	1.74	2.38
RUSE	—	—	0.95	1.0	1.07	1.15	1.24
PDIF	—	—	0.87	1.0	1.29	1.81	2.61
PERS	2.12	1.62	1.26	1.0	0.83	0.63	0.50
PREX	1.59	1.33	1.12	1.0	0.87	0.71	0.62
FCIL	1.43	1.30	1.10	1.0	0.87	0.73	0.62
SCED	—	1.43	1.14	1.0	1.0	1.0	—

The early design model adjusts the nominal effort using 7 effort multipliers (EMs). Each effort multiplier (also called cost drivers) has 7 possible weights as given in Table 4.15. These factors are used for the calculation of adjusted effort as given below:

$$PM_{adjusted} = PM_{nominal} \times \left[\prod_{i=1}^7 EM_i \right]$$

$PM_{adjusted}$ effort may very even up to 400% from $PM_{nominal}$.

Hence $PM_{adjusted}$ is the fine tuned value of effort in the early design phase.

Example 4.10

A software project of application generator category with estimated 50 KLOC has to be developed. The scale factor (B) has low precedentness, high development flexibility and low team cohesion. Other factors are nominal. The early design cost drivers like platform difficult (PDIF) and Personnel Capability (PERS) are high and others are nominal. Calculate the effort in person months for the development of the project.

Solution

Here

$$\begin{aligned} B &= 0.91 + 0.01 * (\text{Sum of rating on scaling factors for the project}) \\ &= 0.91 + 0.01 * (4.96 + 2.03 + 4.24 + 4.38 + 4.68) \\ &= 0.91 + 0.01(20.29) = 1.1129 \end{aligned}$$

$$\begin{aligned} PM_{nominal} &= A * (\text{size})^B \\ &= 2.5 * (50)^{1.1129} = 194.41 \text{ Person months.} \end{aligned}$$

The 7 cost drivers are

PDIF = high (1.29)

PERS = high (0.83)

RCPX = nominal (1.0)

RUSE = nominal (1.0)

PREX = nominal (1.0)

FCIL = nominal (1.0)

SCEO = nominal (1.0)

$$\begin{aligned} PM_{adjusted} &= PM_{nominal} * \left[\prod_{i=1}^7 EM_i \right] \\ &= 194.41 * [1.29 \times 0.83] = 194.41 \times 1.07 \\ &= 208.155 \text{ Person-months.} \end{aligned}$$

4.5.3 Post Architecture Model

The Post architecture Model is the most detailed estimation model and is intended to be used when a software life cycle architecture has been completed. This model is used in the

development and maintenance of software products in the application generators, system integration or infrastructure sectors.

The Post Architecture model adjusts nominal effort using 17 efforts multipliers. The large number of multipliers takes advantage of the greater knowledge available later in the development stage.

$$PM_{adjusted} = PM_{nominal} \times \left[\prod_{i=1}^7 EM_i \right]$$

EM : Effort multiplier which is the product of 17 cost drivers.

The 17 cost drivers of the Post Architecture model are described in the Table 4.16.

Table 4.16: Post Architecture Cost Driver rating level summary

Cost driver	Purpose	Very low	Low	Nominal	High	Very High	Extra High
RELY (Reliability required)	Measure of the extent to which the software must perform its intended function over a period of time	Only slight inconvenience	Low, easily recoverable losses	Moderate, easily recoverable losses	High financial loss	Risk to human life	—
DATA (Data base size)	Measure the affect of large data requirements on product development	—	$\frac{\text{Database size(D)}}{\text{Prog. size (P) < 10}}$	$10 \leq \frac{D}{P} < 100$	$100 \leq \frac{D}{P} < 1000$	$\frac{D}{P} \geq 1000$	—
CPLX (Product complexity)	Complexity is divided into five areas: Control operations, computational operations, device dependent operations, data management operations & User Interface management operations.					See Table 4.17	
DOCU Documentation	Suitability of the project's documentation to its life cycle needs	Many life cycle needs uncovered	Some needs uncovered	Adequate	Excessive for life cycle needs	Very Excessive	—

(Contd...)

TIME (Execution Time constraint)	Measure of execution time constraint on software	—	—	≤ 50% use of available execution time	70%	85%	95%
STOR (Main storage constraint)	Measure of main storage constraint on software	—	—	≤ 50% use of available storage	70%	85%	95%
PVOL (Platform Volatility)	Measure of changes to the OS, compilers, editors, DBMS etc.	—	Major changes every 12 months & minor changes every 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 week Minor: 2 days	—
ACAP (Analyst capability)	Should include analysis and design ability, efficiency & thoroughness, and communication skills.	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—
PCAP (Programmers capability)	Capability of Programmers as a team. It includes ability, efficiency, thoroughness & communication skills	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—
PCON (Personnel Continuity)	Rating is in terms of Project's annual personnel turnover	48%/year	24%/year	12%/year	6%/year	3%/year	—
AEXP (Applications Experience)	Rating is dependent on level of applications experience.	≤ 2 months	6 months	1 year	3 year	6 year	—
PEXP (Platform experience)	Measure of Platform experience	≤ 2 months	6 months	1 year	3 year	6 year	—

(Contd...)

LTEX (Language & Tool experience)	Rating is for Language & tool experience	≤ 2 months	6 months	1 year	3 year	6 year	—
TOOL (Use of software tools)	It is the indicator of usage of software tools	No use	Beginning to use	Some use	Good use	Routine & habitual use	—
SITE (Multisite development)	Site location & Communication technology between sites	International with some phone & mail facility	Multiplicity & multi company with individual phones, FAX	Multiplicity & multi company with Narrow band mail	Same city or Metro with wideband electronic communication	Same building or complex with wideband electronic communication & Video conferencing	Fully co-located with interactive multimedia
SCED (Required Development Schedule)	Measure of Schedule constraint. Ratings are defined in terms of percentage of schedule stretch-out or acceleration with respect to nominal schedule	75% of nominal	85%	100%	130%	160%	—

Product complexity is based on control operations, computational operations, device dependent operations, data management operations and user interface management operations. Module complexity ratings are given in Table 4.17.

The numeric values of these 17 cost drivers are given in Table 4.18 for the calculation of the product of efforts *i.e.*, effort multiplier (EM). Hence PM adjusted is calculated which will be a better and fine tuned value of effort in person months.

Table 4.17: Module complexity[”] ratings

	<i>Control Operations</i>	<i>Computational Operations</i>	<i>Device-dependent Operations</i>	<i>Data Management Operations</i>	<i>User Interface Management Operations</i>
Very Low	Straight-line code with a few non-nested structured programming operators: DOs, CASEs, IF THEN ELSEs. Simple module composition via procedure calls or simple scripts.	Evaluation of simple expressions : e.g., $A = B + C * (D - E)$	Simple read, write statements with simple formats.	Simple arrays in main memory. Simple COTS-DB queries, updates.	Simple input forms, report generators.
Low	Straight forward nesting of structured programming operators. Mostly simple predicates	Evaluation of moderate-level expressions: e.g., $D = \text{SQRT}(B^{**2} - 4*A*C)$	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level.	Single file subsetting with no data structure changes, no edits, no intermediate files. Moderately complex COTS-DB queries, updates.	Use of simple graphic user interface (GUI) builders.
Nominal	Mostly simple nesting. Some inter module control and statistical routines. Decision tables. Simple callbacks or message passing, including middleware-supported distributed processing.	Use of standard maths and statistical routines. Basic matrix/vector operations.	I/O processing includes device selection, status checking and error processing.	Multi-file input and single file output. Simple structural changes, simple edits. Complex COTS-DB queries, updates.	Simple use of widget set.

(Contd.)...

High	Highly nested structured programming operators with many compound predicates. Queue and stack control. Homogeneous, distributed processing. Single processor soft real-time control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns.	Operations at physical I/O level (physical storage address translations; Complex data seeks, read, etc.) Optimized I/O overlap.	Simple triggers activated by data stream contents. Complex data restructuring. Simple voice I/O, multimedia.	Widget set development and extension. Simple voice I/O, multimedia.
Very High	Reentrant and recursive coding. Fixed priority interrupt handling. Task synchronization, complex callbacks, heterogeneous distributed processing. Single-processor hard real-time control.	Difficult but structured numerical analysis: near-singular matrix equations, partial differential equations. Simple parallelization.	Routines for interrupt diagnosis, servicing, masking. Communication line handling. Performance-intensive embedded systems.	Distributed database coordination. Complex triggers. Search optimization.	Moderately complex 2D/3D, dynamic graphics, multimedia.
Extra High	Multiple resource scheduling with dynamically changing priorities. Microcode-level control. Distributed hard real-time control.	Difficult and unstructured numerical analysis: highly accurate analysis of noisy, stochastic data. Complex parallelization.	Device timing-dependent coding, micro-programmed operations. Performance-critical embedded systems.	Highly coupled, dynamic relational and object structures. Natural language data management.	Complex multi-media, virtual reality.

Table 4.18: 17 Cost drivers

Cost Driver	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
RELY	0.75	0.88	1.00	1.15	1.39	
DATA		0.93	1.00	1.09	1.19	
CPLX	0.75	0.88	1.00	1.15	1.30	1.66
RUSE		0.91	1.00	1.14	1.29	1.49
DOCU	0.89	0.95	1.00	1.06	1.13	
TIME			1.00	1.11	1.31	1.67
STOR			1.00	1.06	1.21	1.57
PVOL		0.87	1.00	1.15	1.30	
ACAP	1.50	1.22	1.00	0.83	0.67	
PCAP	1.37	1.16	1.00	0.87	0.74	
PCON	1.24	1.10	1.00	0.92	0.84	
AEXP	1.22	1.10	1.00	0.89	0.81	
PEXP	1.25	1.12	1.00	0.88	0.81	
LTEX	1.22	1.10	1.00	0.91	0.84	
TOOL	1.24	1.12	1.00	0.86	0.72	
SITE	1.25	1.10	1.00	0.92	0.84	0.78
SCED	1.29	1.10	1.00	1.00	1.00	

Schedule estimation

Development time can be calculated using $PM_{adjusted}$ as a key factor and the desired equation is:

$$TDEV_{nominal} = [\phi \times (PM_{adjusted})^{(0.28 + 0.2(B - 0.091))}] * \frac{SCED \%}{100}$$

where ϕ = constant, provisionally set to 3.67

$TDEV_{nominal}$ = calendar time in months with a scheduled constraint

B = Scaling factor

$PM_{adjusted}$ = Estimated effort in Person months (after adjustment)

Size measurement

Size can be measured in any unit and the model can be calibrated accordingly. However, COCOMO II details are:

- (i) Application composition model uses the size in object points.
- (ii) The other two models use size in KLOC.

Early design model uses unadjusted function points. These function points are converted into KLOC using Table 4.19. Post architecture model may compute KLOC after defining LOC counting rules. If function points are used, then use unadjusted function points and convert it into KLOC using Table 4.19 [JONE 91].

Table 4.19: Converting function points to lines of code

Language	SLOC/UFP
Ada	71
AI Shell	49
APL	32
Assembly	320
Assembly (Macro)	213
ANSI/Quick/Turbo Basic	64
Basic-Compiled	91
Basic-Interpreted	128
C	128
C++	29
ANSI Cobol 85	91
Fortan 77	105
Forth	64
Jovial	105
Lisp	64
Modula 2	80
Pascal	91
Prolog	64
Report Generator	80
Spreadsheet	6

COCOMO II reflects the experience and data collection of developers. It is a complex model and there are many attributes with too much scope for uncertainty in estimating their values. Each organisation should calibrate the model and attribute values according to its own historical data which may reflect local circumstances that affect the model.

Example 4.11

Consider the software project given in example 4.10. Size and scale factor (B) are the same. The identified 17 Cost drivers are high reliability (RELY), very high database size (DATA), high execution time constraint (TIME), very high analyst capability (ACAP), high programmers capability (PCAP). The other cost drivers are nominal. Calculate the effort in Person-Months for the development of the project.

Solution

Here

$$B = 1.1129$$

$$PM_{nominal} = 194.41 \text{ Person-months}$$

$$\begin{aligned} PM_{adjusted} &= PM_{nominal} \times \left[\prod_{i=1}^{17} EM_i \right] \\ &= 194.41 \times (1.15 \times 1.19 \times 1.11 \times 0.67 \times 0.87) \\ &= 194.41 \times 0.885 \\ &= 172.05 \text{ Person-months.} \end{aligned}$$

If analyst capability is very high alongwith high programmers capability, the effort will be reduced significantly. If such human resource factors are low, effort will be increased drastically. Therefore, human resource factors should be considered very carefully and are very important for the success of the project. If we consider estimated adjusted effort ($PM_{adjusted}$) of both the cases, the values are 208.155 Person-months and 172.05 Person-months. Hence difference is of 36.105 Person-months. More we know, better is the estimate, hence, effort estimated in Post Architecture model is more realistic and reasonable.

4.6 THE PUTNAM RESOURCE ALLOCATION MODEL

Norden [NORD58] of IBM observed that the Rayleigh curve can be used as an approximate model for a range of hardware development projects. This approach was later extended by Putnam to apply to software projects. Putnam observed that the Rayleigh curve (Fig. 4.6) was a close representation, not only at the project level but also for software subsystem development. As many as 150 projects were studied by Norden [NORD77] and subsequently by Putnam, and apparently both researchers observed the same tendency for the manpower curve to rise, peak, and then exponentially trail off as a function of time.

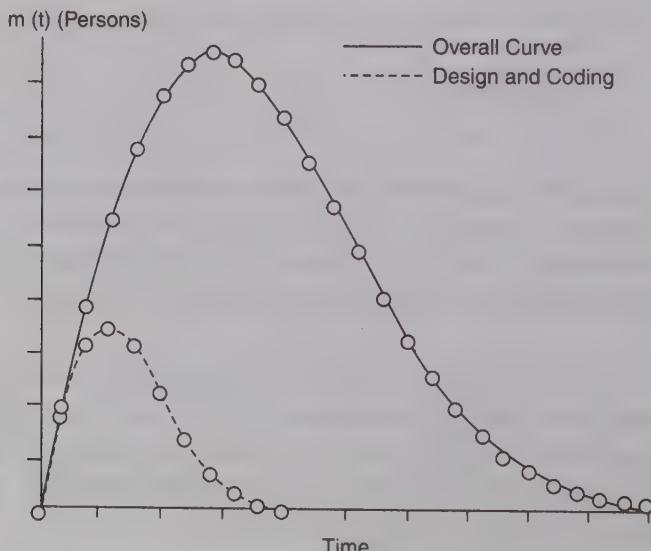


Fig. 4.6: The Rayleigh manpower loading curve.

4.6.1 The Norden/Rayleigh Curve

The Norden/Rayleigh equation represents manpower, measured in persons per unit time as a function of time. It is usually expressed in person-year/year (PY/YR). The Rayleigh curve is modeled by the differential equation

$$m(t) = \frac{dy}{dt} = 2Kae^{-at^2} \quad (4.16)$$

where dy/dt is the manpower utilization rate per unit time, “ t ” is elapsed time, “ a ” is a parameter that affects the shape of the curve, and “ K ” is the area under the curve in the interval $[0, \infty]$. Integrating equation (4.16), on interval $[0, t]$, we obtain

$$y(t) = K[1 - e^{-at^2}] \quad (4.17)$$

where $y(t)$ is the cumulative manpower used upto time t .

$$y(0) = 0$$

$$y(\infty) = K$$

The cumulative manpower is null at the start of the project, and grows monotonically towards the total effort K (area under the curve).

It can be seen from the Fig. 4.7 that the parameter “ a ”, which has the dimensions of $1/\text{time}^2$, plays an important role in the determination of the peak manpower. The larger the value of “ a ”, earlier the peak time occurs and steeper is the person profile. By deriving the manpower function relative to time and finding the zero value of this derivative, the relationship between the peak time, “ t_d ”, and “ a ” can be found to be:

$$\begin{aligned} \frac{d^2y}{dt^2} &= 2Kae^{-at^2}[1 - 2at^2] = 0 \\ \therefore t_d^2 &= \frac{1}{2a} \end{aligned} \quad (4.18)$$

“ t_d ” denotes the time where maximum effort rate occurs. Thus, the point “ t_d ” on the time scale should correspond very closely to the total project development time. If we substitute “ t_d ” for t in equation (4.17), we can obtain an estimate for development time

$$E = y(t) = K \left(1 - e^{-\frac{t_d^2}{2t_d^2}} \right) = K(1 - e^{-0.5})$$

$$E = y(t) = 0.3935 K \quad (4.19)$$

Actually if we divide the life cycle of a project into phases, each phase can be modeled by a curve of the form given in Fig. 4.6.

As shown in equation (4.18), the peak manning time is related to “ a ”. Therefore, “ a ” can be obtained from the peak time as follows:

$$a = \frac{1}{2t_d^2}$$

The number of people involved in the project at the peak time then becomes easy to determine by replacing “ a ” with $1/2t_d^2$ in the Norden/ Rayleigh model. By making this substitution in equation (4.16), we have

$$\begin{aligned} m(t) &= \frac{2K}{2t_d^2} te^{-\frac{t^2}{2t_d^2}} \\ &= \frac{K}{t_d^2} te^{-\frac{t^2}{2t_d^2}} \end{aligned}$$

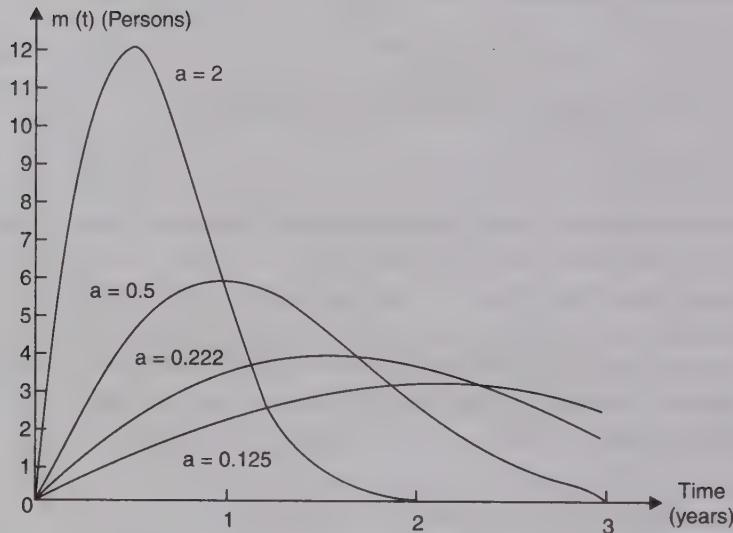


Fig. 4.7 Influence of parameter ‘ a ’ on the manpower distribution

At time $t = t_d$, the peak Manning, $m(t_d)$ is obtained, which is denoted by m_o . Thus, expression for the peak Manning of the project is

$$m_o = \frac{K}{t_d \sqrt{e}}$$

Where “ K ” is the total project cost (or effort) in person-years, “ t_d ” is the delivery time in years, “ m_o ” is the number of persons employed at the peak.

The average rate of software team build-up can also be calculated by dividing m_o by t_d .

Example 4.12

A software project is planned to cost 95 PY in a period of 1 year and 9 months. Calculate the peak Manning and average rate of software team build up.

Solution

$$\text{Software project cost} \quad K = 95 \text{ PY}$$

$$\text{Peak development time} \quad t_d = 1.75 \text{ years}$$

$$\text{Peak Manning} \quad = m_o = \frac{K}{t_d \sqrt{e}}$$

$$\frac{95}{1.75 \times 1.648} = 32.94 \approx 33 \text{ persons}$$

Average rate of software team build-up

$$= m_o/t_d = 33/1.75 = 18.8 \text{ person/year or } 1.56 \text{ person/month.}$$

Example 4.13

Consider a large-scale project for which the manpower requirement is $K = 600$ PY and the development time is 3 years 6 months.

- (a) Calculate the peak manning and peak time.
- (b) What is the manpower cost after 1 year and 2 months?

Solution

- (a) We know $t_d = 3$ years and 6 months = 3.5 years

$$\text{Now } m_o = \frac{K}{t_d \sqrt{e}}$$

$$\therefore m_o = 600/(3.5 \times 1.648) \approx 104 \text{ persons}$$

- (b) We know

$$y(t) = K [1 - e^{-at^2}]$$

$$\begin{aligned} t &= 1 \text{ year and 2 months} \\ &= 1.17 \text{ years} \end{aligned}$$

$$a = \frac{1}{2t_d^2} = \frac{1}{2 \times (3.5)^2} = 0.041$$

$$\begin{aligned} y(1.17) &= 600[1 - e^{-0.041(1.17)^2}] \\ &= 32.6 \text{ PY} \end{aligned}$$

4.6.2 Difficulty Metric

The slope of the manpower distribution at start time ($t = 0$) also has some useful properties. By differentiating the Norden/Rayleigh function with respect to time, the following equation is obtained.

$$m'(t) = \frac{d^2y}{dt^2} = 2Ka e^{-at^2} (1 - 2at^2)$$

Then, for $t = 0$,

$$m'(0) = 2Ka = \frac{2K}{2t_d^2} = \frac{K}{t_d^2} \quad (4.20)$$

The ratio $\frac{K}{t_d^2}$ is called difficulty and is denoted by D , which is measured in person/year:

$$D = \frac{K}{t_d^2} \quad (4.21)$$

This relationship shows that a project is more difficult to develop when the manpower demand is high or when the time schedule is short (small t_d). It is also interesting to note that difficult projects will tend to have a steeper demand for manpower at the beginning for the same time scale. After studying a large number (about 50) of Army developed software projects, Putnam observed that for systems that were relatively easy to develop, D tended to be small, while for systems that were relatively hard to develop, D tended to be large.

Peak manning is defined as

$$m_0 = \frac{K}{t_d \sqrt{e}}$$

We notice that the difficulty, D, is also related to the peak manning, " m_0 " and the development time " t_d " by

$$D = \frac{K}{t_d^2} = \frac{m_0 \sqrt{e}}{t_d}$$

Thus, difficult projects tend to have a higher peak manning for a given development time, which is in line with Norden's observations relative to the parameter "a" [LOND87].

Manpower buildup

D is dependent upon "K" and " t_d ". The derivative of D relative to "K" and " t_d " are:

$$D'(t_d) = \frac{-2K}{t_d^3} \text{ Person/years}^2$$

$$D'(K) = \frac{1}{t_d^2} \text{ year}^{-2}$$

In practice, $D'(K)$ will always be very much smaller than the absolute value of $D'(t_d)$. This difference in sensitivity is shown by considering two projects

Project A : Cost = 20 PY & $t_d = 1$ year

Project B : Cost = 120 PY & $t_d = 2.5$ years

The derivative values are

Project A : $D'(t_d) = -40$ & $D'(K) = 1$

Project B : $D'(t_d) = -15.36$ & $D'(K) = 0.16$

This shows that a given software development is time sensitive.

Putnam also observed that the difficulty derivative relative to time played an important role in explaining the behaviour of software development. He noted that if the project scale is increased the development time also increases to such an extent that the quantity K/t_d^3 remains constant around a value, which could be 8, 15 or 27. This quantity is represented by D_0 and can be expressed as:

$$D_0 = \frac{K}{t_d^3} \text{ Person/year}^2$$

The value of D_0 is related to the nature of software developed in the following way:

- $D_0 = 8$ refers to entirely new software with many interfaces and interactions with other systems.
- $D_0 = 15$ refers to new stand alone system.
- $D_0 = 27$ refers to the software that is rebuilt from existing software.

Putnam also discovered that D_0 could vary slightly from one organization to another depending on the average skill of the analysts, developers and the management involved.

In practice, D_0 has a strong influence on the shape of the manpower distribution. The larger D_0 is, the steeper manpower distribution is, and the faster the necessary manpower build up will be. For this reason, the quantity D_0 is called the manpower build up.

Example 4.14

Consider the example 4.13 and calculate the difficulty and manpower build up.

Solution

We know

$$\text{Difficulty } D = \frac{K}{t_d^2}$$

$$= \frac{600}{(3.5)^2} = 49 \text{ person/year}$$

Manpower build up can be calculated by following equation

$$D_0 = \frac{K}{t_d^3}$$

$$= \frac{600}{(3.5)^3} = 14 \text{ person/year}^2.$$

4.6.3 Productivity Versus Difficulty

It is appropriate to find relationship between productivity and difficulty. Productivity is defined as the number of lines of code developed per person-month. Putnam has observed that productivity is proportional to the difficulty

$$P \propto D^\beta \quad (4.22)$$

The average productivity may be defined as :

$P = \text{Lines of code produced} / \text{Cumulative manpower used to produce code}$

$$P = S/E \quad (4.23)$$

where S is lines of code produced and E is cumulative manpower used from $t = 0$ to $t = t_d$ (inception of the project to the delivery time).

Using nonlinear regression, Putnam determined from an analysis of 50 army projects that

$$P = \phi D^{-2/3} \quad (4.24)$$

Using equation 4.23, this relationship may be written as

$$S = \phi D^{-2/3} E$$

$$= \phi D^{-2/3} (0.3935 K)$$

Using equation 4.21, we have

$$S = \phi \left[\frac{K}{t_d^2} \right]^{-\frac{2}{3}} K(0.3935) \\ S = 0.3935 \phi K^{1/3} t_d^{4/3} \quad (4.25)$$

In the usual form of this expression, the quantity 0.3935ϕ is replaced by a coefficient C , which is given the name of Technology Factor. It reflects the effect of various factors on productivity such as hardware constraints, program complexity, personnel experience levels, and the programming environment. Putnam has proposed using a discrete spectrum of 20 values for C ranging from 610 to 57314 (assuming that K is measured in person-years and T in years) depending on an assessment of the technology factor that applies to the project under consideration. Equation 4.25 may be modified and now written as:

$$S = CK^{1/3} t_d^{4/3} \quad (4.26)$$

The value of C can also be found out as:

$$C = S \cdot K^{-1/3} t_d^{-4/3} \quad (4.27)$$

It is easy to use the size, cost and development time of past projects to determine the value of C and hence to revise the value of C obtained to model forthcoming projects.

4.6.4 The Trade-off between Time Versus Cost

In software projects, time cannot be freely exchanged against cost. Such a trade off is limited by the nature of software development. For a given organization, developing a software of size S , the quantity obtained from equation 4.26 is constant. Using equation 4.26, we have

$$K^{1/3} t_d^{4/3} = S/C$$

If we raise power by 3, then Kt_d^4 is constant for a constant size software. A compression of the development time t_d will produce an increase of manpower cost. If compression is excessive, not only would the software development cost much more, but also the development would become so difficult that it would increase the risk of being unmanageable. This is in line with a remark made by Boehm that the time scale should never be reduced to less than 75% of its initial calculated value [LOND87].

The name given by Putnam to the later versions of this model is Software Life cycle Methodology (SLIM). This model is a combination of expertise and statistical computations and could be used effectively for predictive purposes if we had a suitable algorithm that we might use to predict the value of C for a software project. Using equation 4.27, we have

$$K = \frac{1}{t_d^4} \left[\frac{S}{C} \right]^3 \quad (4.28)$$

For a software product of a given size and fixed development environment, equation (4.28) implies that the effort K varies inversely as the fourth power of the development time. For instance, if we take the constant C to be 5000 and if we estimate the size of the project $S = 500,000$ LOC then

$$K = \frac{1}{t_d^4} (100)^3$$

Table 4.20 shows how the required effort in person-years changes as the development time measured in years changes. Thus, reducing the development time from 5 years to 4 years would increase the total effort and the cost by a factor 2.4, reducing it to 3 years would increase them by a factor of 7.7.

Table 4.20: Manpower versus development time

t_d (years)	K (person-years)
5.0	1600
4.0	3906
3.5	6664
3.0	12346

Putnam attempted to offer support for his use of " t_d^4 " in equation (4.28) after examining 750 software systems [PUTN84]. With 251 of them it was shown that equation (4.28) is an acceptable model of the relationship among "K", "S", and " t_d ". However C was computed using equation 4.27; it was not the result of some independent assessment of the technology level. Therefore, the data from 251 systems given in [PUTN84] may be said to offer only marginal support for the fourth power law. Furthermore, there was no evidence offered that the same relationship holds for the other 499 systems.

4.6.5 Development Sub-cycle

All that has been described so far is related to the project life cycle, as represented in Fig. 4.8, by the project curve.

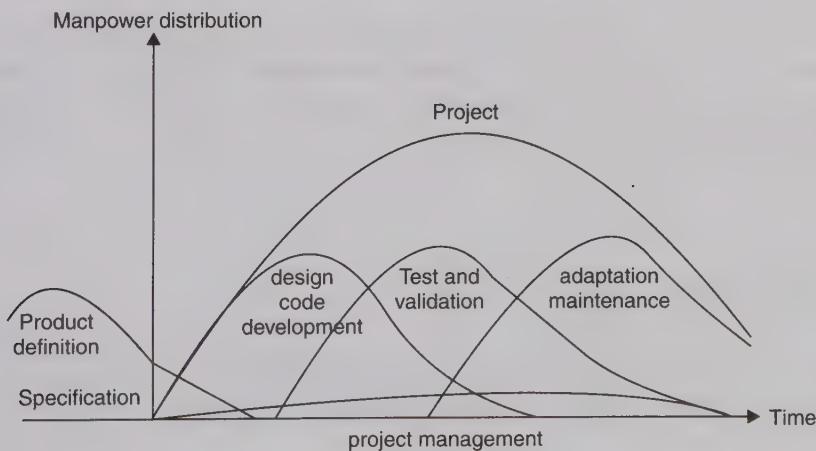


Fig. 4.8: Project life cycle.

Curve is represented by a Rayleigh function, which gives the manning level relative to time and reaches a peak at time t_d . The project curve is the addition of two curves called development curve and test and validation curve. Both the curves are the sub-cycles of the project curve and can be modeled by Rayleigh function.

Let $m_d(t)$ and $y_d(t)$ be the design manning and the cumulative design manpower cost which can be represented by:

$$m_d(t) = 2K_d b t e^{-bt^2} \quad (4.29)$$

$$y_d(t) = K_d [1 - e^{-bt^2}] \quad (4.30)$$

An examination of $m_d(t)$ function shows a non-zero value for m_d at time " t_d ". This is because the manpower involved in design and coding is still completing this activity after " t_d " in the form of rework due to the validation of the product. Nevertheless, for the model a level of completion has to be assumed for development.

It is good practical assumption to assume that the development, will be 95% completed by the time t_d , this gives

$$\frac{y_d(t)}{K_d} = 1 - e^{-bt_d} = 0.95 \quad (4.31)$$

It is then legitimate by set of previous definitions and by analogy with the Norden coefficient "a", to set:

$$b = \frac{1}{2t_{od}^2} \quad (4.32)$$

Where t_{od} is the time at which the development curve exhibits a peak manning. This can then be used to obtain the following relation between the development time " t_d ", and development peak manning, t_{od} :

$$t_{od} = \frac{t_d}{\sqrt{6}} \quad (4.33)$$

Relationship between " K_d " (total manpower cost of the development sub-cycle) and K (total manpower cost of the generic cycle) must be established. This can be obtained by using the observation that at the origin of time both cycles have the same slope. Thus from equation (4.21) and by differentiation of equation (4.29)

$$\left(\frac{dm}{dt} \right)_o = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2} = \left(\frac{dm_d}{dt} \right)_o$$

Consider equation (4.33), and we have

$$K_d = K/6 \quad (4.34)$$

It should also be noted that the difficulty D , is the same whether expressed in terms of " K " and " t_d " or " K_d " and " t_{od} ". More formally, this can be stated by:

$$D = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2}$$

This does not apply to the manpower build up D_o

$$D_o = \frac{K}{t_d^3} = \frac{K_d}{\sqrt{6} t_{od}^3} \quad (4.35)$$

Note here that there is an extra factor $\sqrt{6}$ when the calculations are made at the sub cycle level.

Conte et al., [CONT86] investigated the Putnam models and observed that they work reasonably well on very large systems, but seriously over estimate effort on medium or small size systems. The model emphasises heavily on the size and development schedule attributes while down playing with other attributes. The constant C must be used to reflect all other attributes including complexity, use of modern programming practices and personnel ability.

Example 4.15

A software development requires 90 PY during the total development sub-cycle. The development time is planned for a duration of 3 years and 5 months [CONT86]

- (a) Calculate the manpower cost expended until development time.
- (b) Determine the development peak time
- (c) Calculate the difficulty and manpower build-up.

Solution

- (a) Duration $t_d = 3.41$ years

We know from equation (4.31)

$$\frac{y_d(t_d)}{K_d} = 0.95$$

$$\begin{aligned} Y_d(t_d) &= 0.95 \times 90 \\ &= 85.5 \text{ PY} \end{aligned}$$

- (b) We know from equation (4.33)

$$\begin{aligned} t_{od} &= \frac{t_d}{\sqrt{6}} = 3.41/2.449 = 1.39 \text{ years} \\ &\approx 17 \text{ months.} \end{aligned}$$

- (c) Total Manpower development

$$\begin{aligned} K_d &= y_d(t_d)/0.95 \\ &= 85.5/0.95 = 90 \end{aligned}$$

$$K = 6K_d = 90 \times 6 = 540 \text{ PY}$$

$$D = K/t_d^2 = 540/(3.41)^2 = 46 \text{ person/years}$$

$$D_0 = \frac{K}{t_d^3} = 540/(3.41)^3 = 13.6 \text{ person/year}^2.$$

Example 4.16

A software development for avionics has consumed 32 PY upto development cycle and produced a size of 48000 LOC. The development of project was completed in 25 months. Calculate the development time, total manpower requirement, development peak time, difficulty, manpower build up and technology factor.

Solution

Development time $t_d = 25$ months = 2.08 years

$$\text{Total manpower development } K_d = \frac{Y_d(t_d)}{0.95}$$

$$K_d = \frac{32}{0.95} = 33.7 \text{ PY}$$

$$\text{Development peak time } t_{od} = \frac{(t_d)}{\sqrt{6}}$$

$$= 0.85 \text{ years (or 10 months).}$$

$$K = 6K_d = 6 \times 33.7 = 202 \text{ PY}$$

$$D = \frac{K}{t_d^2} = \frac{202}{(2.08)^2} = 46.7 \text{ person/year}$$

$$D_0 = \frac{K}{t_d^3} = \frac{202}{(2.08)^3} = 22.5 \text{ person/year}^2$$

Technology factor

$$\begin{aligned} C &= SK^{-1/3} t_d^{-4/3} \\ &= 48000 \times (202)^{-1/3} (2.08)^{-4/3} \\ &= 3077. \end{aligned}$$

Example 4.17

What amount of software can be delivered in 1 year 10 months in an organization whose technology factor is 2400 if a total of 25 PY is permitted for development effort?

Solution

$$t_d = 1.8 \text{ years}$$

$$K_d = 25 \text{ PY}$$

$$K = 25 \times 6 = 150 \text{ PY}$$

$$C = 2400$$

We know

$$\begin{aligned} S &= CK^{-1/3} t_d^{4/3} \\ &= 2400 \times 5.313 \times 2.18 = 27920 \text{ LOC} \end{aligned}$$

Example 4.18

The software development environment of an organization developing real time software has been assessed at technology factor of 2200. The maximum value of manpower build up for this type of software is $D_o = 7.5$. The estimated size of the software to be developed is $S = 55000$ LOC [LOND87].

- (a) Determine the total development time, the total development manpower cost, the difficulty and the development peak manning.
- (b) The development time determined in (a) is considered too long. It is recommended that it be reduced by two months. What would happen?

Solution

We have $S = CK^{1/3}t_d^{4/3}$

$$\left(\frac{S}{C}\right)^3 = Kt_d^4$$

which is also equivalent to

$$\left(\frac{S}{C}\right)^3 = D_0 t_d^7$$

then $t_d = \left[\frac{1}{D_0} \left(\frac{S}{C} \right)^3 \right]^{1/7}$

Since $\frac{S}{C} = 25$,

$$t_d = 3 \text{ years}$$

As $K = D_0 t_d^3 = 7.5 \times 27 = 202 \text{ PY}$

$$\text{Total development manpower cost } K_d = \frac{202}{06} = 33.75 \text{ PY}$$

$$D = D_0 t_d = 22.5 \text{ person/year}$$

$$t_{od} = \frac{t_d}{\sqrt{6}} = \frac{3}{\sqrt{6}} = 1.2 \text{ years}$$

Using equations (4.20) and (4.29), we have

$$m_d(t) = 2K_d bte^{-bt^2}$$

$$Y_d(t) = K_d(1 - e^{-bt^2})$$

Here, $t = t_{od}$

Peak manning $= m_{od} = Dt_{od}e^{-1/2}$
 $= 22.5 \times 1.2 \times .606 \approx 16 \text{ persons}$

(b) Developing time reduction means either developing the software at a higher manpower build-up or producing less software.

(i) Increase Manpower Build-up

$$D_o = \frac{1}{t_d^7} \left(\frac{S}{C} \right)^3$$

The new development time would be 2.8 years and new manpower build-up is

$$D_o = (25)^3 / (2.8)^7 = 11.6 \text{ person/year}^2$$

$$K = D_0 t_d^3 = 254 \text{ PY}$$

$$K_d = \frac{254}{6} = 42.4 \text{ PY}$$

$$D = D_0 t_d = 32.5 \text{ person/year}$$

The peak time is $t_{od} = 1.14$ years

$$\begin{aligned}\text{Peak manning } m_{od} &= D t_{od} e^{-0.5} \\ &= 32.5 \times 1.14 \times 0.6 \approx 22 \text{ persons}\end{aligned}$$

Note the huge increase in peak manning and manpower cost.

(ii) Produce Less Software

$$\left(\frac{S}{C}\right)^3 = D_0 t_d^7 = 7.5 \times (2.8)^7 = 10119.696$$

$$\left(\frac{S}{C}\right)^3 = 21.62989$$

Then for

$$C = 2200$$

$$S = 47586 \text{ LOC}$$

The problem is now to decide which software functions can be cut down.

Example 4.19

A stand-alone project for which the size is estimated at 12500 LOC is to be developed in an environment such that the technology factor is 1200. Choosing a manpower build up $D_o = 15$, calculate the minimum development time, total development man power cost, the difficulty, the peak manning, the development peak time, and the development productivity.

Solution

$$\text{Size (S)} = 12500 \text{ LOC}$$

$$\text{Technology factor (C)} = 1200$$

$$\text{Manpower build up (D}_o\text{)} = 15$$

$$\text{Now } S = CK^{1/3}t_d^{4/3}$$

$$\frac{S}{C} = K^{1/3}t_d^{4/3}$$

$$\left(\frac{S}{C}\right)^3 = Kt_d^4$$

$$\begin{aligned}\text{Also we know } D_o &= \frac{K}{t_d^3} \\ K &= D_o t_d^3 = D_o t_d^3\end{aligned}$$

$$\text{Hence } \left(\frac{S}{C}\right)^3 = D_o t_d^7$$

Substituting the values, we get

$$\left(\frac{12500}{1200}\right)^3 = 15t_d^7$$

$$t_d = \left[\frac{(10.416)^3}{15} \right]^{1/7}$$

$$t_d = 1.85 \text{ years}$$

(i) Hence Minimum development time (t_d) = 1.85 years.

$$(ii) \text{ Total development manpower cost } K_d = \frac{K}{6}$$

Hence,

$$\begin{aligned} K &= 15t_d^3 \\ &= 15(1.85)^3 = 94.97 \text{ PY} \end{aligned}$$

$$K_d = \frac{K}{6} = \frac{94.97}{6} = 15.83 \text{ PY}$$

$$(iii) \text{ Difficulty D} = \frac{K}{t_d^2} = \frac{94.97}{(1.85)^2} = 27.75 \text{ Person/year}$$

$$(iv) \text{ Peak Manning } m_0 = \frac{K}{t_d \sqrt{e}}$$

$$= \frac{94.97}{1.85 \times 1.648} = 31.15 \text{ Persons}$$

$$\begin{aligned} (v) \text{ Development Peak time } t_{od} &= \frac{t_d}{\sqrt{6}} \\ &= \frac{1.85}{2.449} = 0.755 \text{ years} \end{aligned}$$

(vi) Development Productivity

$$= \frac{\text{No. of lines of code (S)}}{\text{effort (K}_d)}$$

$$= \frac{12500}{15.83} = 789.6 \text{ LOC/PY.}$$

4.7 SOFTWARE RISK MANAGEMENT

We, software developers are extremely optimists. When planning software projects, we often assume that everything will go exactly as planned. Alternatively, we take the other extreme position. The creative nature of software development means we can never accurately predict what is going to happen, so what is the point of making detailed plans? Both these perspectives can lead to software surprises, when unexpected things happen that throw the project completely off track. Software surprises are never good news.

Risk Management is becoming recognized as an important area in the software industry to reduce this surprise factor. Risk management means dealing with a concern before it becomes a crisis. Therefore, most of the software development activities include risk management as a key part of the planning process and expect the plan to highlight the specific risk areas. The project planning is expected to quantify both probability of failure and consequences of failure and to describe what will be done to reduce the risk.

4.7.1 What is Risk?

Tomorrow's problems are today's risks. Hence, a simple definition of a "risk" is a problem that could cause some loss or threaten the success of the project, but which has not happened yet.

These potential problems might have an adverse impact on cost, schedule, or technical success of the project, the quality of our software products, or project team morale. Risk management is the process of identifying, addressing and eliminating these problems before they can damage the project.

We need to differentiate risks, as potential problems, from the current problems of the project. Different approaches are required to address these two kinds of issues. For example, a staff shortage because we have not been able to hire people with the right technical skills is a current problem; but the threat of our technical people being hired away by the competition is a risk. Current real problems require prompt, corrective action, whereas risk can be dealt with in several different ways. We might choose to avoid the risk entirely by changing the project approach or even cancelling the project.

Whether we tackle them head-on or keep our heads in the sand, risks have a potentially huge impact on many aspects of the project. We do far too much pretending in software. We pretend, we know who our users are, we know what their needs are, that we would not have staff turn over problems, that we can solve all technical problems that arise, that our estimates are achievable, and that nothing unexpected will happen.

Risk management is about discarding the rose-coloured glasses and confronting the very real potential of undesirable events conspiring to throw our project off track [WIEG98].

4.7.2 Typical Software Risks

The list of evil things that can befall a software project is depressingly long. Possible risks can come from group brainstorming activities, or from a risk factor chart accumulated from previous projects. There are no magic solutions to any of these risk factors, so we need to rely on past experience and a strong knowledge of contemporary software engineering and management practices to control these risks. Capers Jones has identified the top five risk factors that threaten projects in different applications [JONE94].

Dependencies

Many risks arise due to dependencies of project on outside agencies or factors. It is not easy to control these external dependencies. Some typical dependency-related risk factors are:

- Availability of trained, experienced people
- Intercomponent or inter-group dependencies
- Customer-furnished items or information
- Internal and external subcontractor relationships

Requirement issues

Many projects face uncertainty and turmoil around the product's requirements. While some of this uncertainty is tolerable in early stages, but the threat to success increases if such issues are not resolved as the project progresses. If we do not control requirements-related risk factors, we might either build the wrong product, or build the right product badly. Either situation results in unpleasant surprises and unhappy customers. Some typical factors are:

- Lack of clear product vision
- Lack of agreement on product requirements

- Unprioritized requirements
- New market with uncertain needs
- Rapidly changing requirements
- Inadequate impact analysis of requirements changes

Management issues

Project Managers usually write the risk management plan, and most people do not wish to air their weaknesses (assuming they even recognize them) in public. Nonetheless, issues like those listed below can make it harder for projects to succeed. If we do not confront such touchy issues, we should not be surprised if they bite us at some point. Defined project tracking processes, and clear roles and responsibilities, can address some of these risk factors.

- Inadequate planning and task identification
- Inadequate visibility into actual project status
- Unclear project ownership and decision making
- Unrealistic commitments made, sometimes for the wrong reasons
- Managers or customers with unrealistic expectations
- Staff personality conflicts
- Poor communication

Lack of knowledge

The rapid rate of change of technologies, and the increasing change of skilled staff, mean that our project teams may not have the skills we need to be successful. The key is to recognize the risk areas early enough so that we can take appropriate preventive actions, such as obtaining training, hiring consultants, and bringing the right people together on the project team. Some of the factors are:

- Inadequate training
- Poor understanding of methods, tools, and techniques
- Inadequate application domain experience
- New technologies
- Ineffective, poorly documented, or neglected processes

Other risk categories

The list of potential risk areas is long. Some of the critical areas are:

- Unavailability of adequate testing facilities
- Turnover of essential personnel
- Unachievable performance requirements
- Technical approaches that may not work

4.7.3 Risk Management Activities

Risk management involves several important steps, each of which is illustrated in Fig. 4.9. We should assess the risks on the project, so that we understand what may occur during the

course of development or maintenance. The assessment consists of three activities: identifying the risks, analyzing them, and assigning priorities to each of them.

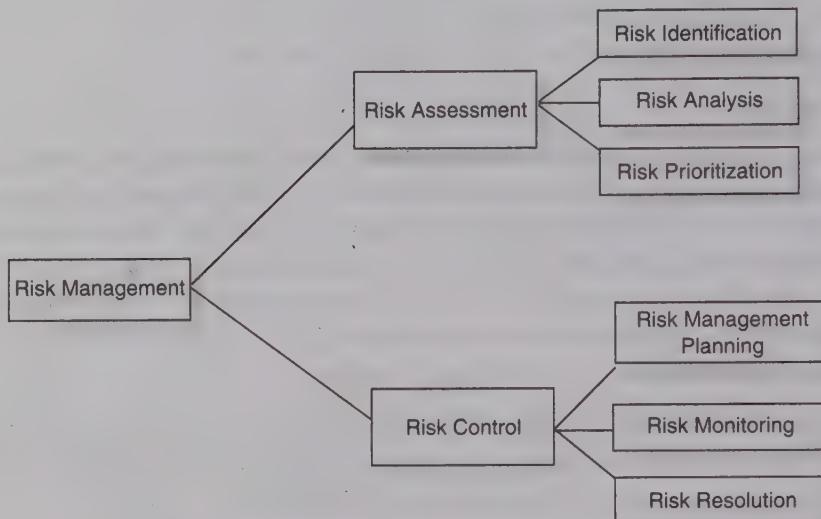


Fig. 4.9: Risk management activities.

Risk assessment

It is the process of examining a project and identifying areas of potential risk. Risk identification can be facilitated with the help of a checklist of common risk areas of software projects; or by examining the contents of an organizational database of previously identified risks. Risk analysis involves examining how project outcomes might change with modification of risk input variables. Risk prioritization helps the project focus on its most severe risks by assessing the risk exposure. Exposure is the product of the probability of incurring a loss due to the risk and the potential magnitude of that loss. This prioritization can be done in a quantitative way, by estimating the probability (0.1 – 1.0) and relative loss, on a scale of 1 to 10. Multiplying these factors together provide an estimation of risk exposure due to each risk item, which can run from 0.1 (do not give it another thought) through 10 (stand back, here it comes!). The higher the exposure, the more aggressively the risk should be tackled. It may be easier to simply estimate both probability and impact as High, Medium, or Low. Those items having at least one dimension rated as High are the ones to worry about first.

Another way of handling risk is the risk avoidance. Do not do the risky things! We may avoid risks by not undertaking certain projects, or by relying on proven rather than cutting edge technologies.

Risk control

It is the process of managing risks to achieve the desired outcomes. Risk management planning produces a plan for dealing with each significant risk. It is useful to record decisions in the plan, so that both customer and developer can review how problems are to be avoided, as well as how they are to be handled when they arise. We should also monitor the project as

development progresses, periodically reevaluating the risks, their probability, and likely impact. Risk resolution is the execution of the plans for dealing with each risk.

Simply identifying the risks of any project is not enough. We should write them down in a way that communicates the nature and status of risks over the duration of the project.

REFERENCES

- [BASL80] Basil V.R., "Resource Models: Models & Metrics for Software Management and Engineering", IEEE, pp-4–9, 1980.
- [BOEH81] Boehm B.W., "Software Engineering Economics", Prentice -Hall, 1981.
- [UCSD01] "COCOMO-II" Model Definition Manual", Version 1.4, University of Southern California, 2001.
- [GHEZ94] Ghezzi C., et Al., "Software Engineering", PHI, 1994.
- [HUMP95] Humphrey Watts S., "A Discipline for Software Engineering", Addison-Wesley, Pub. Co., 1995.
- [JONE91] Jones C., "Applied Software Measurement, Assuring Productivity & Quality", McGraw Hill, New York, NY, 1991.
- [JONE94] Jones C., "Assessment and Control of Software Risks", Englewood Cliffs, N.J., Prentice-Hall, 1994.
- [LOND87] Londeix B., "Cost Estimation for Software Development", Addison -Wesley Pub. Co., 1987.
- [NORD58] Norden P.V., "Curve Fitting for a Model of Applied Research and Development Scheduling", IBM Journal, Research & Development, Vol 3, No.2, PP.232–248, July, 1958.
- [NORD77] Norden P.V., "Project Life Cycle Modeling: Background and Application of the Life Cycle Curves", US Army Computer Systems Command, 1977.
- [PRESS2K] Pressman Roger, "Software Engineering", McGraw Hill Pub., 2000.
- [PUTN84] Putnam I.H., D.T. Putnam, "A Verification of the Software Fourth Power Trade off Law", Proc. Of the Int. Soc. of para-metric Analysis 3, 1, May 184, 443–471
- [SAGE90] Sage A.P., & J.D. Palmer, "Software System Engineering", John Wiley & Sons Pub. Co., 1990.
- [SOMM96] Summerville Ian, "Software Engineering", Addison - Wesley Pub Co., 1996.
- [WALS77] Walson C.E. & C.P. Felix, "A Method for Programming Measurement and Estimation", IBM System Journal, 16(1), pp - 54–73, 1977.
- [WIEG98] Wiegers K.E., "Know Your Enemy: Software Risk Management", Software Development Magazine, October, 1998.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions

- 4.1. After the finalisation of SRS, we may like to estimate
 - (a) Size
 - (b) Cost
 - (c) Development time
 - (d) All of the above.
- 4.2. Which one is not a size measure for software
 - (a) LOC
 - (b) Function Count
 - (c) Cyclomatic Complexity
 - (d) Halstead's program length.
- 4.3. Function count method was developed by
 - (a) B. Beizer
 - (b) B. Boehm
 - (c) M. Halstead
 - (d) Alan Albrecht.

- 4.16.** In COCOMO model, if project size is typically 2 – 50 KLOC, then which mode is to be selected?
- (a) Organic
 - (b) Semidetached
 - (c) Embedded
 - (d) None of the above.
- 4.17.** COCOMO-II was developed at
- (a) University of Maryland
 - (b) University of Southern California
 - (c) IBM
 - (d) AT & T Bell labs
- 4.18.** Which one is not a Category of COCOMO-II?
- (a) End User Programming
 - (b) Infrastructure Sector
 - (c) Requirement Sector
 - (d) System Integration.
- 4.19.** Which one is not an infrastructure software?
- (a) Operating system
 - (b) Database management system
 - (c) Compilers
 - (d) Result management system.
- 4.20.** How many stages are in COCOMO-II?
- (a) 2
 - (b) 3
 - (c) 4
 - (d) 5.
- 4.21.** Which one is not a stage of COCOMO-II?
- (a) Application Composition estimation model
 - (b) Early design estimation model
 - (c) Post architecture estimation model
 - (d) Comprehensive cost estimation model.
- 4.22.** In Putnam resource allocation model, Rayleigh curve is modeled by the equation
- (a) $m(t) = 2at e^{-at^2}$
 - (b) $m(t) = 2Kt e^{-at^2}$
 - (c) $m(t) = 2Kat e^{-at^2}$
 - (d) $m(t) = 2Kt e^{-at^2}$.
- 4.23.** In Putnam resource allocation model, technology factor 'C' is defined as
- (a) $C = SK^{-1/3} t_d^{-4/3}$
 - (b) $C = SK^{1/3} t_d^{4/3}$
 - (c) $C = SK^{1/3} t_d^{-4/3}$
 - (d) $C = SK^{-1/3} t_d^{4/3}$.
- 4.24.** Risk management activities are divided in
- (a) 3 Categories
 - (b) 2 Categories
 - (c) 5 Categories
 - (d) 10 Categories.
- 4.25.** Which one is not a risk management activity?
- (a) Risk assessment
 - (b) Risk control
 - (c) Risk generation
 - (d) None of the above.

EXERCISES

- 4.1.** What are various activities during software project planning?
- 4.2.** Describe any two software size estimation techniques.
- 4.3.** A proposal is made to count the size of 'C' programs by number of semicolons, except those occurring with literal strings. Discuss the strengths and weaknesses to this size measure when compared with the lines of code count.
- 4.4.** Design a LOC counter for counting LOC automatically. Is it language dependent? What are the limitations of such a counter?

- 4.5. Compute the function point value for a project with the following information domain characteristics.

Number of user inputs = 30
Number of user outputs = 42
Number of user enquiries = 08
Number of files = 07
Number of external interfaces = 6

Assume that all complexity adjustment values are moderate.

- 4.6. Explain the concept of function points. Why FPs are becoming acceptable in industry?
- 4.7. What are size metrics? How is function point metric advantageous over LOC metric? Explain.
- 4.8. Is it possible to estimate software size before coding? Justify your answer with suitable examples.
- 4.9. Describe the Albrecht's function count method with a suitable example.
- 4.10. Compute the function point FP for a payroll program that reads a file of employees and a file of information for the current month and prints cheques for all the employees. The program is capable of handling an interactive command to print an individually requested cheque immediately.
- 4.11. Assume that the previous payroll program is expected to read a file containing information about all the cheques that have been printed. The file is supposed to be printed and also used by the program next time it is run, to produce a report that compares payroll expenses of the current month with those of the previous month. Compute function points for this program. Justify the difference between the function points of this program and previous one by considering how the complexity of the program is affected by adding the requirement of interfacing with another application (in this case, itself).
- 4.12. Explain the Walson & Felix model and compare with the SEL model.
- 4.13. The size of a software product to be developed has been estimated to be 22000 LOC. Predict the manpower cost (effort) by Walston-Felix Model and SEL Model.
- 4.14. A database system is to be developed. The effort has been estimated to be 100 Persons-Months. Calculate the number of lines of code and productivity in LOC/Person-Month.
- 4.15. Discuss various types of COCOMO mode. Explain the phase wise distribution of effort.
- 4.16. Explain all the levels of COCOMO model. Assume that the size of an organic software product has been estimated to be 32,000 lines of code. Determine the effort required to develop the software product and the nominal development time.
- 4.17. Using the basic COCOMO model, under all three operating modes, determine the performance relation for the ratio of delivered source code lines per person-month of effort. Determine the reasonableness of this relation for several types of software projects.
- 4.18. The effort distribution for a 240 KLOC organic mode software development project is: product design 12%, detailed design 24%, code and unit test 36%, integrate and test 28%. How would the following changes, from low to high, affect the phase distribution of effort and the total effort: analyst capability, use of modern programming languages, required reliability, requirements volatility?
- 4.19. Specify, design, and develop a program that implements COCOMO. Using reference [BOEH81] as a guide, extend the program so that it can be used as a planning tool.
- 4.20. Suppose a system for office automation is to be designed. It is clear from requirements that there will be five modules of size 0.5 KLOC, 1.5 KLOC, 2.0 KLOC, 1.0 KLOC and 2.0 KLOC respectively. Complexity, and reliability requirements are high. Programmer's capability and experience is

- low. All other factors are of nominal rating. Use COCOMO model to determine overall cost and schedule estimates. Also calculate the cost and schedule estimates for different phases.
- 4.21.** Suppose that a project was estimated to be 600 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.
- 4.22.** Explain the COCOMO-II in detail. What types of categories of projects are identified?
- 4.23.** Discuss the Infrastructure Sector of COCOMO-II.
- 4.24.** Describe various stages of COCOMO-II. Which stage is more popular and why?
- 4.25.** A software project of application generator category with estimated size of 100 KLOC has to be developed. The scale factor (B) has high precedentness, high development flexibility. Other factors are nominal. The cost drivers are high reliability, medium database size, high Personnel capability, high analyst capability. The other cost drivers are nominal. Calculate the effort in Person-months for the development of the project.
- 4.26.** Explain the Putnam resource allocation model. What are the limitations of this model?
- 4.27.** Describe the trade-off between time versus cost in Putnam resource allocation model.
- 4.28.** Discuss the Putnam resource allocation model. Derive the time and effort equations.
- 4.29.** Assuming the Putnam model, with $S = 100,000$, $C = 5000$, $D_o = 15$, Compute development time t_d and manpower development K_d .
- 4.30.** Obtain software productivity data for two or three software development programs. Use several cost estimating models discussed in this chapter. How do the results compare with actual project results?
- 4.31.** It seems odd that cost and size estimates are developed during software project planning—before detailed software requirements analysis or design has been conducted. Why do we think this is done? Are there circumstances when it should not be done?
- 4.32.** Discuss typical software risks. How staff turnover problem affects software projects?
- 4.33.** What are risk management activities? Is it possible to prioritize the risk?
- 4.34.** What is risk exposure? What techniques can be used to control each risk?
- 4.35.** What is risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?
- 4.36.** There are significant risks even in student projects. Analyse a student project and list the risks.

5

Software Design

Contents

5.1 What is Design ?

- 5.1.1 Conceptual and Technical Design
- 5.1.2 Objectives of Design
- 5.1.3 Why Design is Important?

5.2 Modularity

- 5.2.1 Module Coupling
- 5.2.2 Module Cohesion
- 5.2.3 Relationship between Cohesion and Coupling

5.3 Strategy of Design

- 5.3.1 Bottom up Design
- 5.3.2 Top Down Design
- 5.3.3 Hybrid Design

5.4 Function Oriented Design

- 5.4.1 Design Notations
- 5.4.2 Functional Procedure Layers

5.5 IEEE Recommended Practice for Software Design Descriptions (IEEE Std. 1016-1998)

- 5.5.1 Scope
- 5.5.2 References
- 5.5.3 Definitions
- 5.5.4 Purpose of SDD
- 5.5.5 Design Description Information Content
- 5.5.6 Design Description Organisation

5.6 Object Oriented Design

- 5.6.1 Basic Concepts
- 5.6.2 Steps to Analyze and Design Object Oriented System
- 5.6.3 Case Study of Library Management System

5

Software Design

Software design is more creative process than analysis because it deals with the development of the actual mechanics for a new workable system. While analysing, it is possible to produce the correct model of an existing system. However, there is, no such thing as correct design. Good design is always system dependent and what is good design for one system may be bad for another.

The design of the new system must be done in great detail as it will be the basis for future computer programming and system implementation. Design is a problem-solving activity and as such, very much a matter of trial and error. The designer, together with users, has to search for a solution using his/her professional wisdom until there is an agreement that a satisfactory solution has been found.

For small projects (such as student's projects), one can sit with the specifications and simply write a program. For larger projects, it is necessary to bridge the gap between specifications and the coding with something more concrete. This bridge is the software design.

5.1 WHAT IS DESIGN?

Design is the highly significant phase in the software development where the designer plans "how" a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain. A software requirements specifications (SRS) document tells us "what" a system does, and becomes input to the design process, which tells us "how" a software system works. Designing software systems means determining how requirements are realized and result is a software design document (SDD). Thus, the purpose of design phase is to produce a solution to a problem given in SRS document.

A framework of the design is given in Fig. 5.1. It starts with initial requirements and ends up with the final design. Here, data is gathered on user requirements and analysed accordingly. A high level design is prepared after answering questions of requirements. Moreover, design is validated against requirements on regular basis. Design is refined in every cycle and finally it is documented to produce software design document.

Fig. 5.1 shows the design framework.

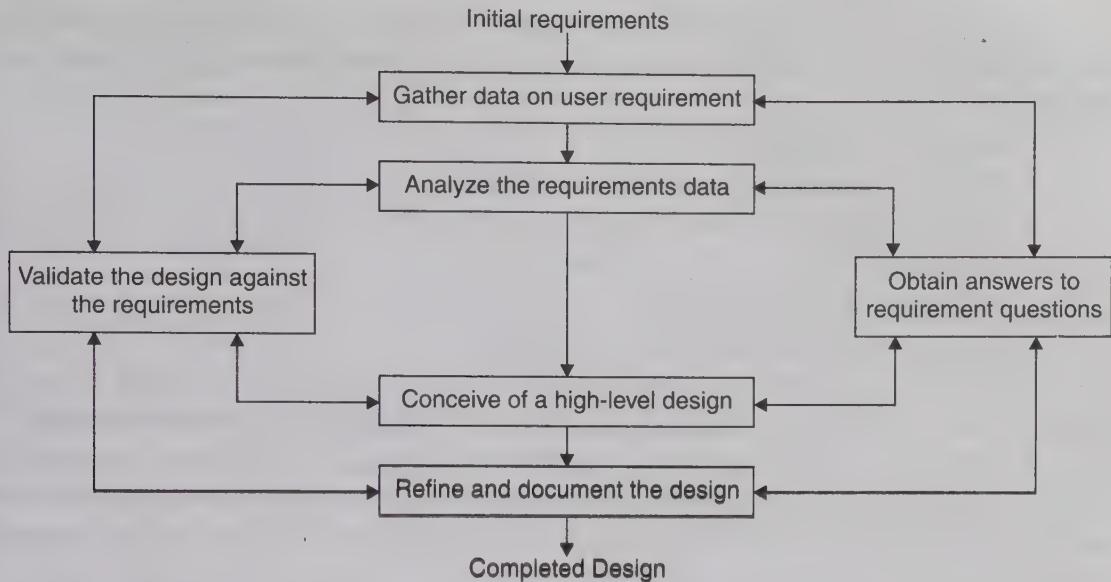


Fig. 5.1: Design framework.

5.1.1 Conceptual and Technical Designs

The process of software design involves the transformation of ideas into detailed implementation descriptions, with the goal of satisfying the software requirements. To transform requirements into a working system, designers must satisfy both customers and the system builders (coding persons). The customers understand what the system is to do. At the same time, the system builders must understand how the system is to work. For this reason, design is really a two part, iterative process. First, we produce conceptual design that tells the customer exactly what the system will do. Once the customer approves the conceptual design, we translate the conceptual design into a much more detailed document, the technical design, that allows system builders to understand the actual hardware and software needed to solve the customer's problem. This two part design process is shown in Fig. 5.2.

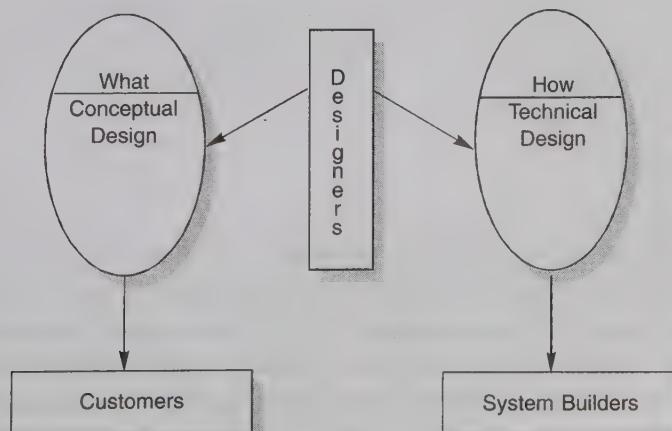


Fig. 5.2: A two-part design process.

The two design documents describe the same system, but in different ways because of the different audiences for the documents. The conceptual design answers the following questions [PFLE98].

- Where will the data come from?
- What will happen to the data in the system?
- How will the system look to users?
- What choices will be offered to users?
- What is the timing of events?
- How will the reports and screens look like?

The conceptual design describes the system in language understandable to the customer. It does not contain any technical jargons and is independent of implementation.

By contrast, the technical design describes the hardware configuration, the software needs, the communications interfaces, the input and output of the system, the network architecture, and anything else that translates the requirements into a solution to the customer's problem.

Sometimes customers are very sophisticated and they can understand the "what" and "how" together. This can happen when customers are themselves software developers and may not require conceptual design. In such cases comprehensive design document may be produced.

5.1.2 Objectives of Design

The specification (*i.e.* the "outside" view) of a program should obviously be as free as possible of aspects imposed by "how" the program will work (*i.e.* the "inside" view). It is seldom a document from which coding can directly be done. So design fills the gap between specifications and coding; taking the specifications, deciding how the program will be organized, and the methods it will use, in sufficient detail as to be directly codeable.

If the specification calls for a large or complex program (or both), then the design is quite likely to work down through a number of levels. At each level, breaking the implementation problem into a combination of smaller and simpler problems. Filling a large gap will involve a number of stepping-stones! The wider the gap, the larger the number of stepping-stones. The design needs to be

- Correct and complete
- Understandable
- At the right level
- Maintainable, and to facilitate maintenance of the produced code

Software designers do not arrive at a finished design document immediately but develop the design iteratively through a number of different phases. The design process involves adding details as the design is developed with constant backtracking to correct earlier, less formal, designs. The starting point is an informal design which is refined by adding information to make it consistent and complete and this is shown in Fig. 5.3 [SOMM2K].

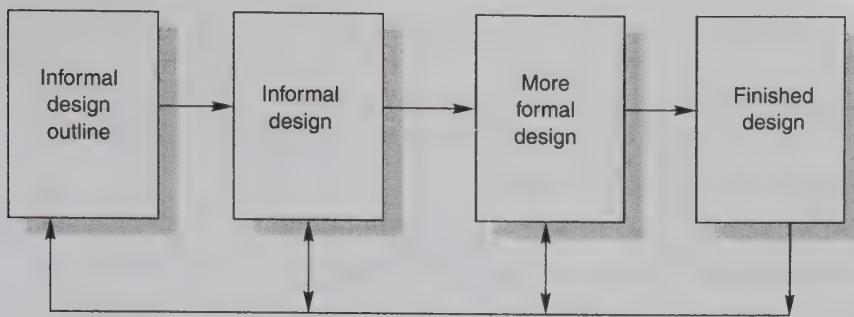


Fig. 5.3: The transformation of an informal design to a detailed design.

5.1.3 Why Design is Important?

A good design is the key to successful product. Almost 2000 years ago Roman Architect Vitruvius recorded the following attributes of a good design:

- Durability
- Utility and
- Charm

A well-designed system is easy to implement, understandable and reliable and allows for smooth evolution. Without design, we risk building an unstable system:

- One that will fail when small changes are made
- One that will be difficult to maintain
- One whose quality can not be assessed until late in the software process.

Therefore, software design should contain a sufficiently complete, accurate and precise solution to a problem in order to ensure its quality implementation.

There are three characteristics that serve as a guide for the evolution of a good design.

- The design must implement all of the explicit requirements contained in the analysis model and it must accommodate all of the implicit requirements desired by the customer.
- The design must be readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional and behavioural domain from an implementation perspective.

5.2 MODULARITY

There are many definitions of the term “module”. They range from “a module is a FORTRAN subroutine” to “a module is an Ada package” to “procedures and functions of PASCAL and C”, to “C++ / Java Classes”, to “Java packages” to “a module is a work assignment for an individual programmer” [FAIR2K]. All of these definitions are correct. A modular system consist of well defined, manageable units with well defined interfaces among the units. Desirable properties of a modular system include:

- Each module is a well defined subsystem that is potentially useful in other applications
- Each module has a single, well defined purpose
- Modules can be separately compiled and stored in a library
- Modules can use other modules
- Modules should be easier to use than to build
- Modules should be simpler from the outside than from the inside

Modularity is the single attribute of software that allows a program to be intellectually manageable [MYER78]. It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

A system is considered modular if it consists of discrete components so that each component can be implemented separately and a change to one component has minimal impact on other components. Here, one important question arises is to what extent we shall modularize. As the number of modules grows, the effort associated with integrating the module also grows. Fig. 5.4 establishes the relationship between cost/effort and number of modules in a software.

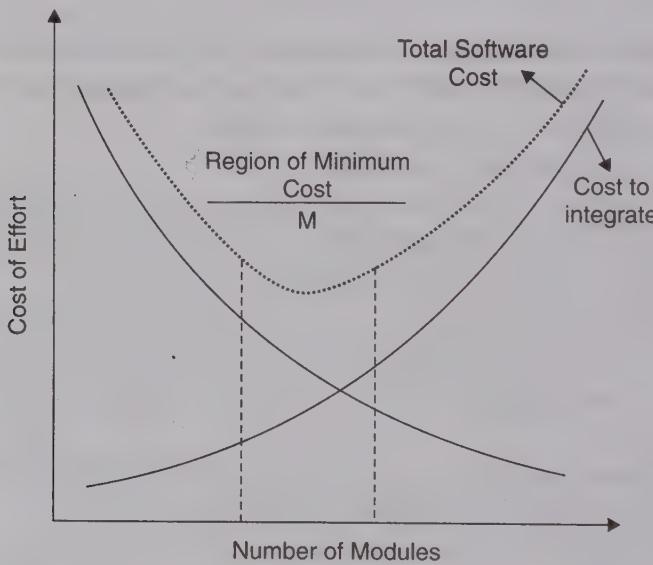


Fig. 5.4: Modularity and software cost.

It can be observed that a software system can not be made modular by simply chopping it into a set of modules. Each module needs to support a well defined abstraction and should have a clear interface through which it can interact with other modules. Thus, it is felt that under modularity and over modularity in a software should be avoided.

5.2.1 Module Coupling

Coupling is the measure of the degree of interdependence between modules. Two modules with high coupling are strongly interconnected and thus, dependent on each other. Two modules with low coupling are not dependent on one another. “Loosely coupled” systems are made

up of modules which are relatively independent. “Highly coupled” systems share a great deal of dependence between modules. For example, if modules make use of shared global variables. “Uncoupled” modules have no interconnections at all; they are completely independent as shown in Fig. 5.5.

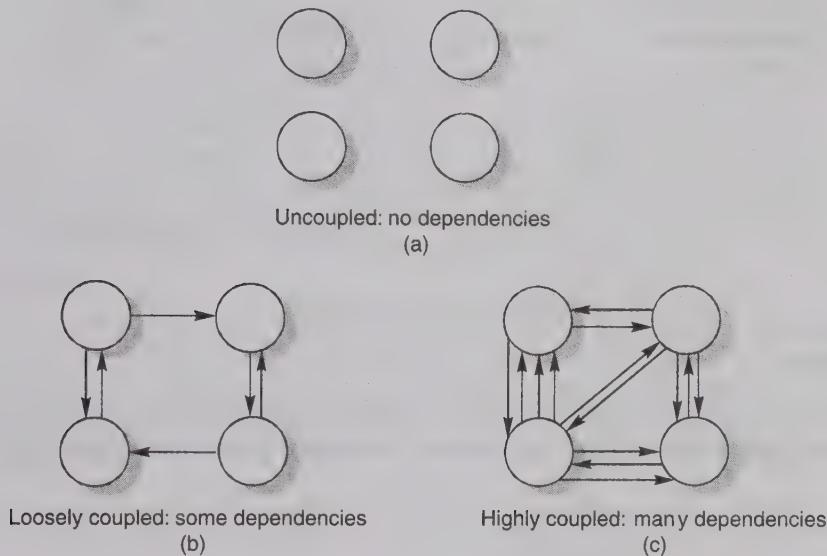


Fig. 5.5: Module coupling.

A good design will have low coupling. Thus, interfaces should be carefully specified in order to keep low value of coupling.

Coupling is measured by the number of interconnections between modules. For example, coupling increases as the number of calls between modules increases, or the amount of shared data increases. The hypothesis is that design with high coupling will have more errors. Loose coupling, on the other hand, minimizes the interdependence amongst modules. This can be achieved in the following ways:

- Controlling the number of parameters passed amongst modules
- Avoid passing undesired data to calling module
- Maintain parent/child relationship between calling and called modules
- Pass data, not the control information

Fig. 5.6 demonstrates two alternative design for editing a student record in a “Student Information System”.

The first design demonstrates tight coupling wherein unnecessary information as student name, student address, course is passed to the calling module. Passing superfluous information unnecessary increases the overhead, reducing the system performance/efficiency.

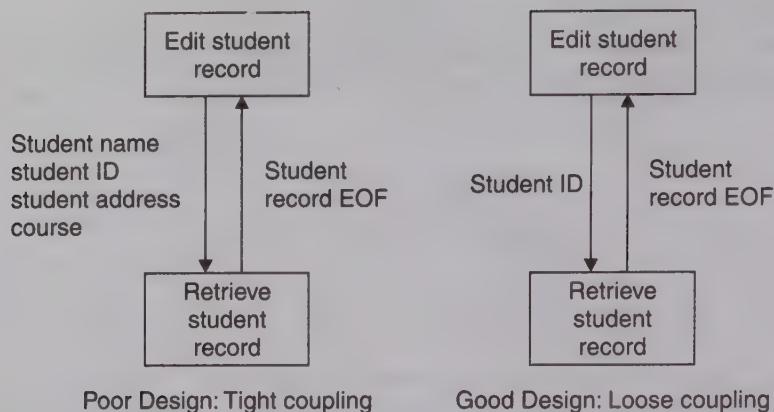


Fig. 5.6: Example of coupling.

Types of coupling

Different types of coupling are content, common, external, control, stamp and data. The strength of coupling from lowest coupling (best) to highest coupling (worst) is given in Fig. 5.7.

Data coupling	Best
Stamp coupling	
Control coupling	
External coupling	
Common coupling	
Content coupling	(Worst)

Fig. 5.7: The types of module coupling.

Given two procedures A and B, we can identify a number of ways in which they can be coupled.

Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent. A good strategy is to ensure that no module communication contains “tramp data”. In Fig. 5.6 above students name, address, course are examples of tramp data that are unnecessarily communicated between modules. By ensuring that modules communicate only necessary data, module dependency is minimized.

Stamp coupling

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another. Since not all data making up the structure are usually necessary in

communication between the modules, stamp coupling typically involves tramp data. If one procedure only needs a part of a data structure, calling module should pass just that part, not the complete data structure.

Control coupling

Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

External coupling

A form of coupling in which a module has a dependency to other module, external to the software being developed or to a particular type of hardware. This is basically related to the communication to external tools and devices.

Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of change. With common coupling, it can be difficult to determine which module is responsible for having set a variable to a particular value. Fig. 5.8 shows how common coupling works [PFLE98]

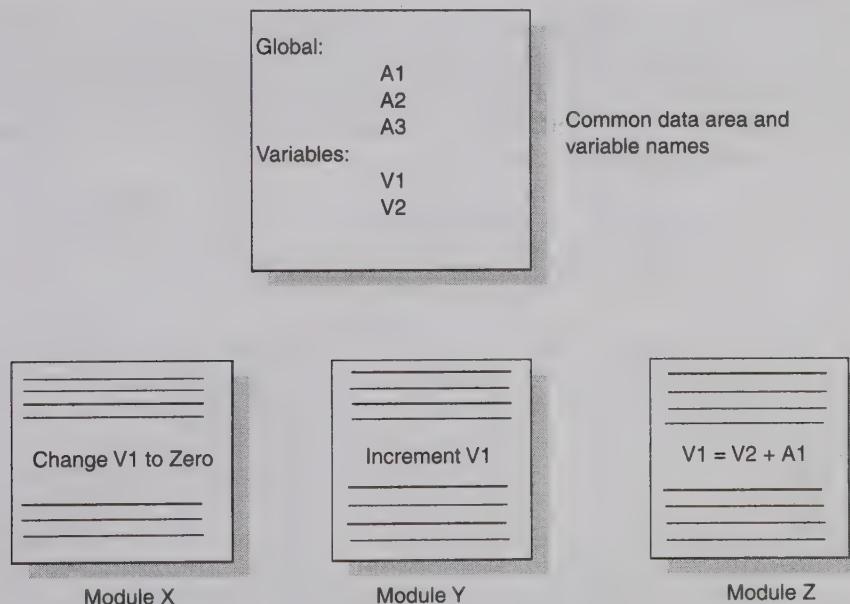


Fig. 5.8: Example of common coupling.

Content coupling

Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. In Fig. 5.9, module B branches into D, even though D is supposed to be under the control of C.

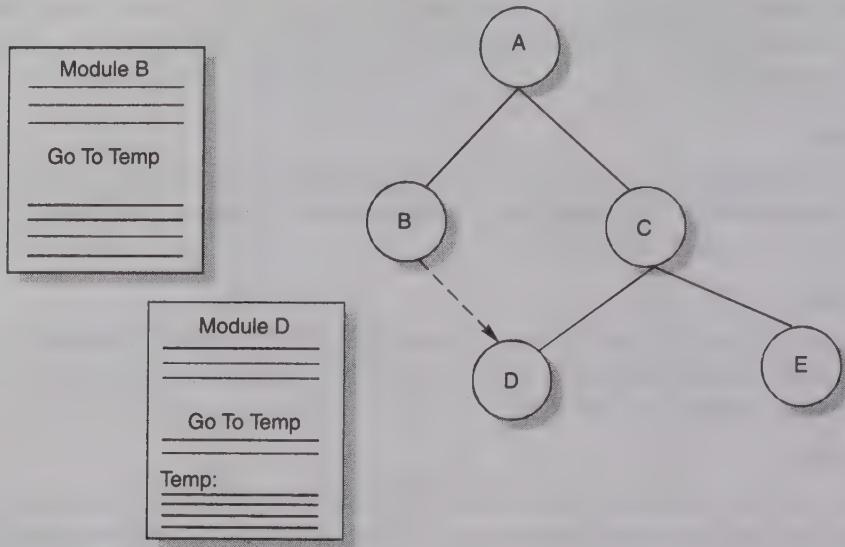


Fig. 5.9: Example of content coupling.

5.2.2 Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related. A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with other modules. This is shown in Fig. 5.10. Cohesion may be viewed as a glue that keeps the module together. It is a measure of the mutual officity of the components of a module.

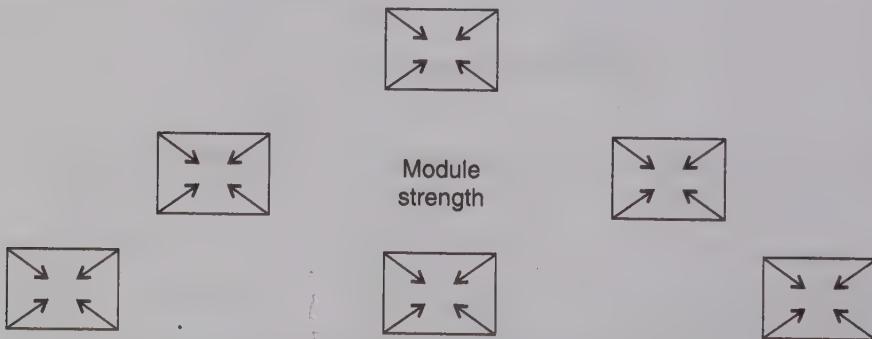


Fig. 5.10: Cohesion = Strength of relations within modules.

Thus, we want to maximize the interaction within a module. Hence, an important design objective is to maximize the module cohesion and minimize the module coupling.

Types of cohesion

There are seven types or levels of Cohesion and are shown in Fig. 5.11. Given a procedure that carries out operations X and Y, we can describe various forms of cohesion between X and Y.

Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Fig. 5.11: Types of module cohesion.

Functional cohesion

X and Y are part of a single functional task. This is very good reason for them to be contained in the same procedure. Such a module often transformed a single input datum into a single output datum. The mathematical subroutines such as ‘calculate current GPA’ or ‘cumulative GPA’ are typical examples of functional cohesion.

Sequential cohesion

X outputs some data which forms the input to Y. This is the reason for them to be contained in the same procedure.

For example, addition of marks of individual subjects into a specific format is used to calculate the GPA as input for preparing the result of the students.

A component is made of parts that need to communicate/exchange data from one source for different functional purposes. They are together in a component for communicational convenience. For example calculate current and cumulative GPA uses the “Student Grade Record” as input.

Communicational cohesion

X and Y both operate on the same input data or contribute towards the same output data. This is okay, but we might consider making them separate procedures.

Procedural cohesion

X and Y are both structured in the same way. This is a poor reason for putting them in the same procedure. Thus, procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed. These types of modules are typically the result of first flow charting the solution to a program and then selecting a sequence of instructions to serve as a module. Since these modules consist of instructions that accomplish several tasks that are virtually unrelated these types of modules tend to be less maintainable. For example, if a report module of an examination system includes the following “calculate student GPA, Print student record, calculate cumulative GPA, print cumulative GPA” is a case of Procedural cohesion.

Temporal cohesion

X and Y both must perform around the same time. So, module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span. The set of functions responsible for initialization, start up activities such as setting program counters or control flags associated with programs exhibit temporal cohesion. This is not a good reason to put them in same procedure.

Logical cohesion

X & Y perform logically similar operations. Therefore, logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions. Considerable duplication can exist in the logical strength level. For example, more than one data item in an input transaction may be a date. Separate code would be written to check that each such date is a valid date. A better way to construct a DATECHECK module and call this module whenever a date check is necessary.

Coincidental cohesion

X and Y here no conceptual relationship other than shared code. Hence, Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another. That is, instead of creating two components, each of one part, only one component is made with two unrelated parts. For example, check validity and print is a single component with two parts. Coincidental cohesion is to be avoided as far as possible.

5.2.3 Relationship between Cohesion & Coupling

The essence of the design process is that the system is decomposed into parts to facilitate the capability of understanding and modifying a system. Projects rarely gets into trouble because of massive requirement changes. These changes can be properly recognized and properly reviewed.

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a good software design professes clean decomposition of a problem into modules and the arrangement of these modules in a neat hierarchy. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

A good example of a system that has high cohesion and low coupling is the 'plug and play' feature of the computer system. Various slots in the mother board of the system simply facilitate to add or remove the various services/functionalities without affecting the entire system. This is because the add on components provide the services in highly cohesive manner. Fig. 5.12 provides a graphical review of cohesion and coupling.

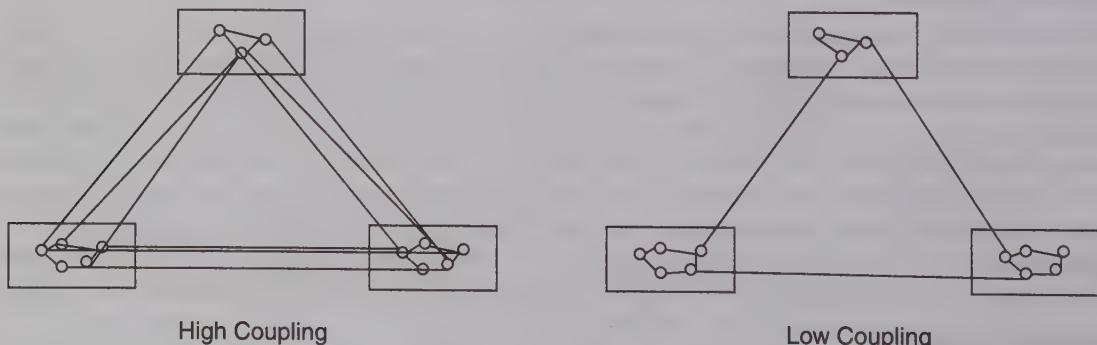


Fig. 5.12: View of cohesion and coupling.

Module design with high cohesion and low coupling characterizes a module as black box when the entire structure of the system is described. Each module can be dealt separately when the module functionality is described.

5.3 STRATEGY OF DESIGN

A good system design strategy is to organize the program modules in such a way that are easy to develop and later to, change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:

- First, even pre-existing code, if any, needs to be understood, organized and pieced together.
- Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system

In early days, if any design was done, it was just “writing down the flowchart in words”. Many people feel that flowcharts are too detailed, so leading to the detail often being decided too early, and too far from the specifications. Hence, there is a sudden jump from specifications to flow chart that leads to the cause of many errors. Flowcharts are at a low level. As a result, errors in flow-charts could only be found by coding them, seeing that the code ran wrongly, diagnosing that the error is in the flow chart, then diagnosing where in the flowchart, then fixing it, modifying code and recording. Repeated surgery on the flow chart sometimes lead to the final flow chart where further errors can not be fixed and the project may fail.

So writers of large and complex software now seldom use flow charts for design. The result is that we have designed other notations for expressing designs, and they are at a “higher level” than flow charts. This helps us to minimize the length of jumps from specifications to design and design to code. These notations usually permit multiple levels of design, and many small jumps in place of one or two massive jumps.

There are many strategies or techniques for performing system design. They include bottom up approach, top down approach, and hybrid approach.

5.3.1 Bottom-Up Design

A common approach is to identify modules that are required by many programs. These modules are collected together in the form of a “library”. These modules may be for math functions, for input-output functions, for graphical functions etc. We may have collections of modules for result preparation system like “maintain student detail”, “maintain subject details”, “marks entry” etc.

This approach lead to a style of design where we decide how to combine these modules to provide larger ones; to combine those to provide even larger ones, and so on, till we arrive at one big module which is the whole of the desired program. The set of these modules form a hierarchy as shown in Fig. 5.13. This is a cross-linked tree structure in which each module is subordinate to those in which it is used.

Since the design progressed from bottom layer upwards, the method is called bottom-up design. The main argument for this design is that if we start coding a module soon after its design, the chances of recoding is high; but the coded module can be tested and design can be validated sooner than a module whose sub modules have not yet been designed.

This method has one terrible weakness; we need to use a lot of intuition to decide exactly what functionality a module should provide.

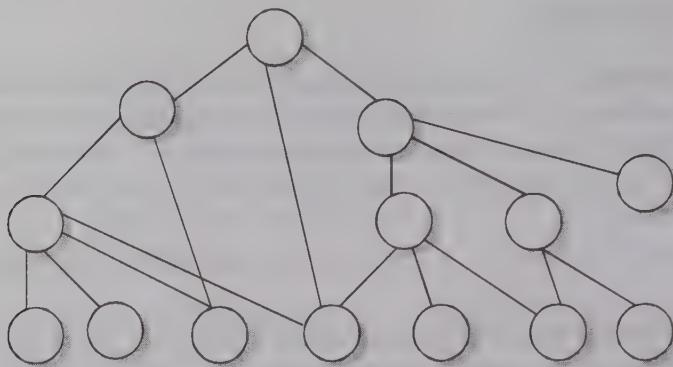


Fig. 5.13: Bottom-up tree structure.

If we get it wrong, then at a higher level, we will find that it is not as per requirements; then we have to redesign at a lower level. If a system is to be built from an existing system, this approach is more suitable, as it starts from some existing modules.

5.3.2 Top-Down Design

The essential idea of top-down design is that the specification is viewed as describing a black box for the program? The designer should decide how the internals of the black box is constructed from smaller black boxes; and that those inner black boxes be specified. This process is then repeated for those inner boxes, and so on till the black boxes can be coded directly.

A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. Most design methodologies are based on this approach and this is suitable, if the specifications are clear and development is from the scratch. If coding of a part starts soon after its design, nothing can be tested until all its subordinate modules are coded.

5.3.3 Hybrid Design

Pure top-down or pure bottom-up approaches are often not practical. For a bottom-up approach to be successful, we must have a good notion of the top to which the design should be heading. Without a good idea about the operations needed at the higher layers, it is difficult to determine what operations the current layer should support [JALO98].

For top-down approach to be effective, some bottom-up (mostly in the lowest design levels) approach is essential for the following reasons:

- To permit common sub modules
- Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more numbers of modules at low levels than at high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

Hybrid approach has really become popular after the acceptance of reusability of modules. Standard Libraries, Microsoft foundation classes (MFCs), object oriented concepts are the steps in this direction. We may soon have internationally acceptable standards for reusability.

5.4 FUNCTION ORIENTED DESIGN

The design activity begins when the SRS document for the software to be developed is available. The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, their specifications of these modules, and how the modules should be interconnected.

Function Oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional viewpoint.

One of the best-known advocates of this method is Niklaus Wirth, the creator of PASCAL and a number of other languages. His special variety is called stepwise refinement, and it is a top down design method. We start with a high level description of what the program does. Then, in each step, we take one part of our high level description and refine it, i.e. specify in somewhat greater detail what that particular part does.

This method works fine for small programs. For large programs its value is more questionable. The main problem is that it is not easy to know what a large program does. For instance, what does UNIX do? Or an airline reservation system? Or a scheme interpreter? The answer is that it depends on what the user types at the terminal. Still, one can usually come up with some kind of high-level function. The risk is that this function is a highly artificial description of reality.

Consider the example of scheme interpreter. Top-level function may look like:

While (not finished)

{

 Read an expression from the terminal;

 Evaluate the expression;

 Print the value;

}

We thus get a fairly natural division of our interpreter into a "read" module, an "evaluate" module and a "print" module. Now we consider the "print" module and is given below:

Print (expression exp)

{

 Switch (exp → type)

 Case integer: /*print an integer*/

 Case real: /*print a real*/

 Case list: /*print a list*/

 :::

}

The other modules are structured in a similar way. We notice that the different kinds of objects that are to be manipulated by the Scheme interpreter (integer, real, etc.) need to be known by every module. Thus, if we need to add a type, every module needs to be altered. Needless to say, we would like to avoid that.

We continue the refinement of each module until we reach the statement level of our programming language. At that point, we can describe the structure of our program as a tree of refinements as in design top-down structure as shown in Fig. 5.14.

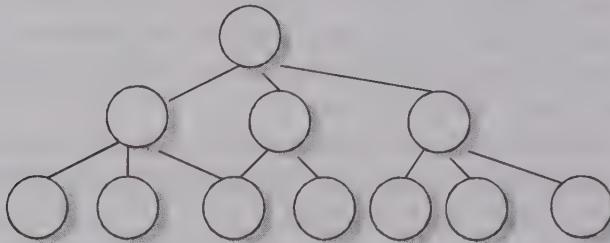


Fig. 5.14: Top-down structure.

Unfortunately, if a program is created top-down, the modules become very specialized. As one can easily see in top down design structure, each module is used by at most one other module, its parent. For a module to be reusable, however, we must require that several other modules as in design-reusable structure as shown Fig. 5.15.

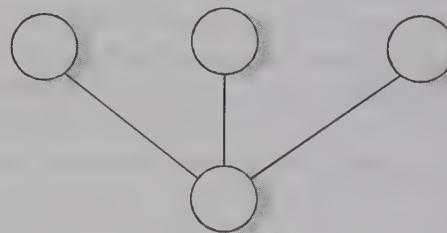


Fig. 5.15: Design reusable structure

It is, of course, not necessary to create a program top-down, even though its structure is function-oriented. However, if we want to delay the decision of what the system is supposed to do as long as possible, a better choice is to structure the program around the data rather than around the actions taken by the program.

5.4.1 Design Notations

During the design phase there are two things of interest: the design of the system, and the process of designing itself. It is for the latter that principles and methods are needed.

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. For a function oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts
- Pseudocode

The first two techniques have been discussed in Chapter 3 and other two are discussed in this section.

Structure chart

The Structure chart is one of the most commonly used method for system design. It partitions a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design. Inputs are given to black box and appropriate outputs are generated by the black box. This concept reduces the complexity because details are hidden from those who have no need or desire to know. Thus, systems are easy to construct and easy to maintain. Here, black boxes are arranged in hierarchical format as shown in Fig. 5.16.

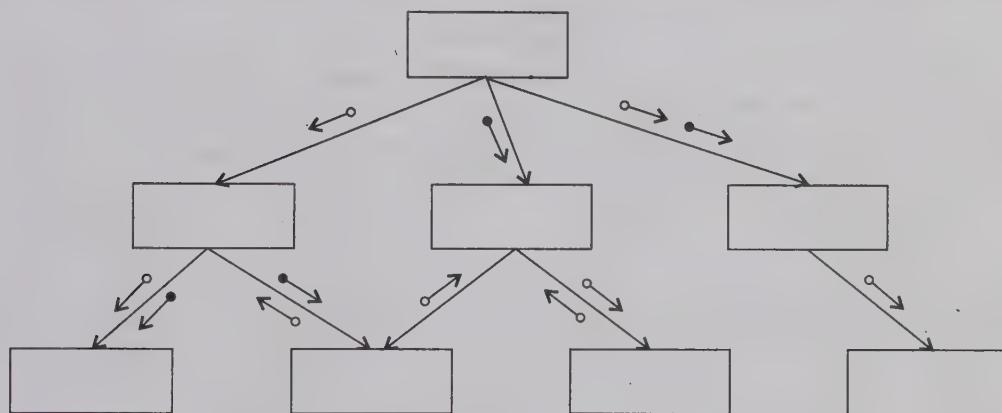


Fig. 5.16: Hierarchical format of a structure chart.

In a structure chart, each program module is represented by a rectangular box. Modules at the top level call the modules at the lower level. The connection between modules are represented by lines between the rectangular boxes. The components are generally read from top to bottom, left to right. Modules are numbered in hierarchical numbering scheme.

When a module calls another, it views the called module as a black box, passing parameters needed for the called module's functionality and receiving answers. Control data passed between modules on a structure chart are represented by labelled directed arrow with filled in circle and data is depicted with an open circle. When a module is used by many other modules, it is put into the library of modules. The diamond symbol is used to represent the fact that one module out of several modules connected with the diamond symbol is used depending on the outcome of the condition attached to the diamond symbol. A loop around the control flow arrows denotes that the respective modules are used repeatedly and is called repetition symbol. Fig. 5.17 shows the notations used in structure chart:

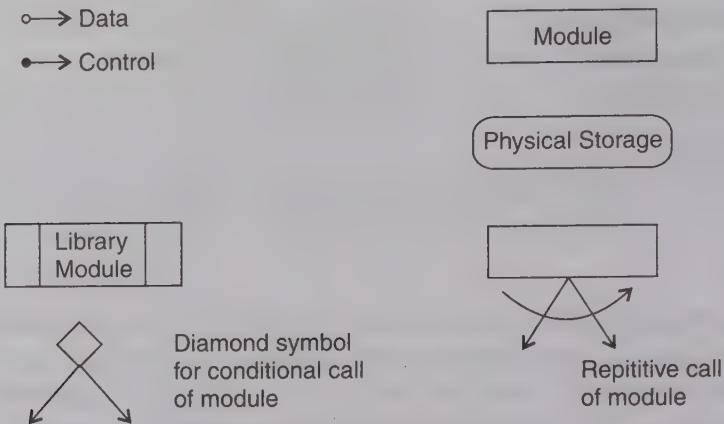


Fig. 5.17: Structure chart notations.

A structure chart for “update file” is given in Fig. 5.18.

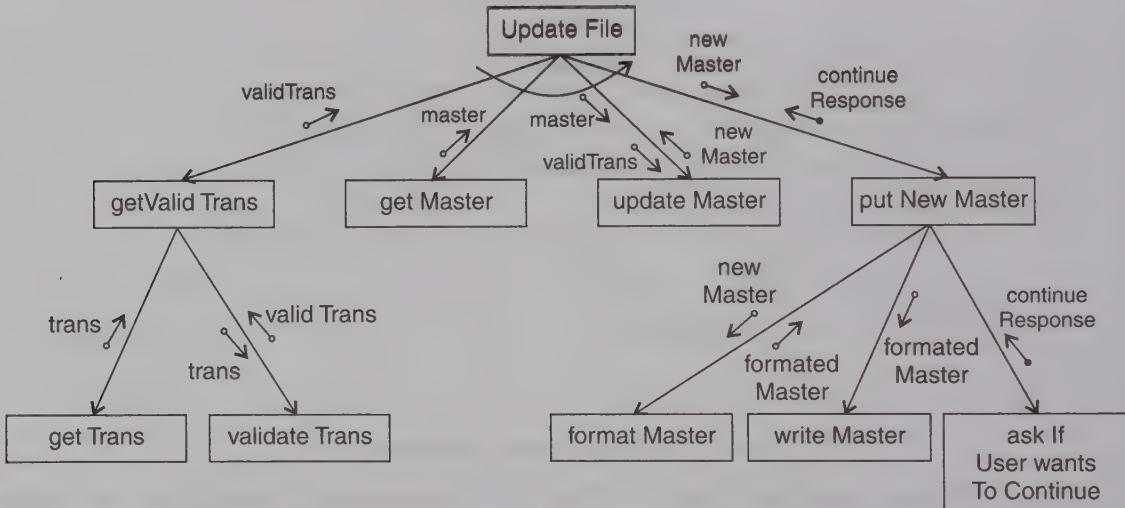


Fig. 5.18: Update file.

The ‘Update file’ calls ‘get Valid Trans’ to get input parameter valid Trans. Module ‘get valid trans’ calls ‘get trans’ module to read the trans from the input device and pass it back to ‘get Valid Trans’. ‘Get Valid Trans’ then calls validate Trans module to validate the transaction which is subsequently passed to the ‘update file’ module. ‘Update file’ module then calls the get master and passes the master and ‘valid trans’ information to ‘update master’. ‘Update master module passes the new master data to ‘update file’.

‘Update file’ then involves ‘put new master’ by passing ‘new master’ data to it. Put new Master involves format master ‘write Master’ and ask if user wants to continue Module ‘Continue Resource control data’ is passed to ‘update file’ to decide user wants to continue by repeating the above procedure.

This type of structure chart is often called as transform-centred structures. Transform-centered structure chart receive an input which is transformed by a sequence of operations,

with each operation being carried out by one module. Fig. 5.19 above is an example of transform-centred structures. Another common type of structure is transaction centred structure. A transaction centred structure describes a system that processes a number of different types of transactions. It is illustrated in Fig. 5.19.

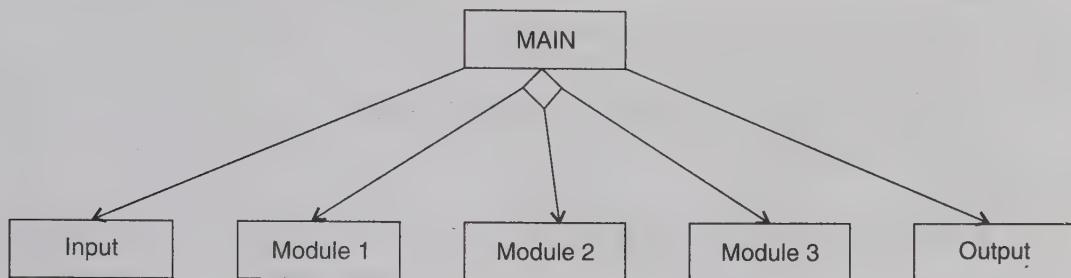


Fig. 5.19: Transaction-centered structure.

In the above figure the MAIN module controls the system operation its function is to:

- invoke the INPUT module to read a transaction;
- determine the kind of transaction and select one of a number of transaction modules to process that transaction, and
- output the results of the processing by calling OUTPUT module.

Pseudocode

Pseudocode notation can be used in both the preliminary and detailed design phases. Like flowcharts, pseudocode can be used at any desired level of abstraction. Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as If-Then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Using the top-down design strategy, each English phrase is expanded into more detailed pseudocode until the design specification reaches the level of detail of the implementation language.

Pseudocode can replace flowcharts and reduce the amount of external documentation required to describe a system [FAIR2K].

5.4.2 Functional Procedure Layers

- Functions are built in layers, Additional notation is used to specify details.
- Level 0
 - ◆ Function or procedure name
 - ◆ Relationship to other system components (e.g., part of which system, called by which routines, etc.)
 - ◆ Brief description of the function purpose.
 - ◆ Author, date.
- Level 1
 - ◆ Function parameters (problem variables, types, purpose, etc.)
 - ◆ Global variables (problem variable, type, purpose, sharing information)

- ◆ Routines called by the function.
- ◆ Side effects.
- ◆ Input/Output Assertions.
- Level 2
 - ◆ Local data structures (variable etc.)
 - ◆ Timing constraints
 - ◆ Exception handling (conditions, responses, events)
 - ◆ Any other limitations.
- Level 3
 - ◆ Body (structured chart, English pseudo code, decision tables, flow charts, etc.)

5.5 IEEE RECOMMENDED PRACTICE FOR SOFTWARE DESIGN DESCRIPTIONS (IEEE STD 1016-1998)

5.5.1 Scope

This is a recommended practice for describing software designs. This is designed by IEEE (Institution of Electricals & Electronics Engineers) and is known as IEEE Standard (IEEE std. 1016-1998) for software design description (SDD). An SDD is a representation of a software system that is used as a medium for communicating software design information.

5.5.2 References

This standard shall be used in conjunction with the following publications.

- (i) IEEE std 830-1998, IEEE recommended practice for software requirements specifications
- (ii) IEEE std 610.12-1990, IEEE glossary of software engineering terminology.

5.5.3 Definitions

Few important definitions are given below:

- (i) **Design entity.** An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.
- (ii) **Design View.** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.
- (iii) **Entity attribute.** A named property or characteristics of a design entity. It provides a statement of fact about the entity.
- (iv) **Software design description (SDD).** A representation of a software system created to facilitate analysis, planning, implementation and decision making. A blueprint or model of the software system. The SDD is used as the primary medium for communicating software design information.

5.5.4 Purpose of an SDD

The SDD shows how the software system will be structured to satisfy the requirements identified in the SRS. It is basically the translation of requirements into a description of the software

structure, software components, interfaces, and data necessary for the implementation phase. Hence, SDD becomes the blue print for the implementation activity.

5.5.5 Design Description Information Content

(i) **Introduction.** The SDD is a representation or model of the system to be created. The model should provide the precise design information needed for planning and implementation of the software system. It should represent partitioning of the system into design entities and describe the important properties and relationship among those entities.

(ii) **Design entities.** A design entity is an element (component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

Design entities result from a decomposition of the software system requirements. The objective is to divide the system into separate components that can be considered, implemented, changed, other components.

The entities may have different nature, but may have common characteristics. Each design entity will have a name, purpose, and function. There are common relationship among entities such as interfaces or shared data. The common characteristics of entities are described by design entity attributes.

(iii) **Design entity attributes.** A design entity attribute is a named characteristic or property of a design entity. It provides a statement of fact about the entity.

Design attributes can be thought of as questions about design entities. The answer to those questions are the values of the attributes. All the questions can be answered, but the content of the answer will depend upon the nature of the entity. The collection of answers provides a complete description of an entity.

All attributes shall be specified for each entity. Attribute descriptions should include references and design considerations such as tradeoffs and assumptions when appropriate. In some cases, attribute descriptions may have the value none. When additional attributes are identified for a specific software project, they should be included in the design description. The attributes and associated information items are defined in the following subsections (from 'a' to 'j').

(a) **Identification.** The name of the entity: Two entities shall not have the same name. The names for the entities may be selected to characterise their nature. This will simplify referencing & tracking in addition to providing identification.

(b) **Type.** A description of the kind of entity. The type attribute shall describe the nature of entity. It may simply name the kind of entity, such as subprogram, module, procedure, process, or data store. Alternatively, design entities may be grouped into major classes to assist in locating an entity dealing with a particular type of information. For a given design description, the chosen entity types shall be applied consistently.

(c) **Purpose.** A description of why the entity exists. The purpose attribute shall provide the rationale for the creation of the entity. Therefore, it shall designate the specific functional and performance requirements for which the entity is created.

(d) **Function.** A statement of what the entity does. The function attribute shall state the transformation applied by the entity to inputs to produce the desired output. In the case of

a data entity, this attribute shall state the type of information stored or transmitted by the entity.

(e) **Subordinates.** The identification of all entities composing this entity. The subordinates attribute shall identify the “composed of relationship” for an entity. This information is used to trace requirements to design entities and to identify parent/child structural relationships through a software system decomposition.

(f) **Dependencies.** A descriptions of the relationships of this entity with other entities. The dependencies attribute shall identify the uses or requires the presence of relationship for an entity. These relationships are often graphically depicted by structure charts, data flow diagrams etc.

(g) **Interface.** A description of how other entities interact with this entity. The interface attribute shall describe the methods of interaction and the rules governing those interactions. The methods of interaction include the mechanisms for invoking or interrupting the entity, for communicating through parameters, common data areas or messages, and for direct access to internal data. The rules governing the interaction include the communications protocol, data format, acceptable values, and the meaning of each value.

This attribute shall provide a description of the input ranges, the meaning of inputs & outputs, the type and format of each input or output error codes. For information systems, it should include inputs, screen formats, and a complete description of the interactive language.

(h) **Resources.** A description of the elements used by the entity that are external to the design. The resources attribute shall identify and describe all of the resources external to the design that are needed by this entity to perform its function. The interaction rules and methods for using the resource shall be specified by this attribute.

This attribute provides information about items such as physical devices (like printers), software services (like math libraries, OS services), and processing resources (like CPU cycles, memory allocation, buffers).

(i) **Processing.** A description of the rules used by the entity to achieve its function. The processing attribute shall describe the algorithm used by the entity to perform a specific task and shall include contingencies. This description is a refinement of the function attribute.

(j) **Data.** A description of data elements internal to the entity. The data attribute shall describe the method of representation, initial values, use, semantics, format, and acceptable values of internal data.

The description of data may be in the form of a data dictionary that describes the content, structure, and use of all data elements. Data information shall describe everything pertaining to the use of data or internal data structures by this entity. It shall include data specifications such as formats, number of elements, and initial values. It shall also include the structures to be used for representing data such as file structures, arrays, stacks, queues, and memory partitions.

5.5.6 Design Description Organisation

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organisation of SDD is given in Table 5.1. This is one of the possible ways to organise and format the SDD.

Table 5.1: Organisation of SDD

1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions and acronyms
2. References
3. Decomposition description
 - 3.1 Module decomposition
 - 3.1.1 Module 1 description
 - 3.1.2 Module 2 description
 - 3.2 Concurrent Process decompostion
 - 3.2.1 Process 1 description
 - 3.2.2 Process 2 description
 - 3.3 Data decomposition
 - 3.3.1 Data entity 1 description
 - 3.3.2 Data entity 2 description
4. Dependency description
 - 4.1 Intermodule dependencies
 - 4.2 Interprocess dependencies
 - 4.3 Data dependencies
5. Interface description
 - 5.1 Module Interface
 - 5.1.1 Module 1 description
 - 5.1.2 Module 2 description
 - 5.2 Process interface
 - 5.2.1 Process 1 description
 - 5.2.2 Process 2 description
6. Detailed design
 - 6.1 Module detailed design
 - 6.1.1 Module 1 detail
 - 6.1.2 Module 2 detail
 - 6.2 Data detailed design
 - 6.2.1 Data entry 1 detail
 - 6.2.2 Data entry 2 detail

Entity attribute information can be organised in several ways to reveal all of the essential aspects of a design. There may be number of ways to view the design. Hence, each design view represents a separate concern about a software system. Together, these views provide a comprehensive description of the design in a concise and usable form that simplifies information access and assimilation. A recommended organisation of the SDD into separate design views to facilitate information access and assimilation is given in Table 5.2.

Table 5.2: Design views

<i>Design View</i>	<i>Scope</i>	<i>Entity attribute</i>	<i>Example representation</i>
Decomposition description	Partition of the system into design entities.	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

5.6 OBJECT ORIENTED DESIGN

“Object Oriented” has clearly become the buzzword of choice in the industry. Almost everyone talks about it. Almost everyone claims to be doing it, and almost everyone says it is better than traditional function oriented design. Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to be manipulated by the program. Thus, it is orthogonal to function oriented design.

Object Oriented Design begins with an examination of the real world “things” that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes (transient state information) and behaviour (functional process information). Each object maintains its own state, and offers a set of services to other objects. Shared data areas are eliminated and objects communicate by message passing (e.g. parameters). Objects are independent entities that may readily be changed because all state and representation information is held within the object itself. Objects may be distributed and may execute either sequentially or in parallel [BOOC03].

5.6.1 Basic Concepts

Object Oriented Design is not dependent on any specific implementation language. Problems are modelled using objects. Objects have:

- Behaviour (they do things)
- State (which changes when they do things)

For example, a car is an object. It has state: whether its engine is running; and it has a behaviour: starting the car, which changes its state from “engine not running” to “engine running”.

The various terms related to object oriented design are Objects, Classes, Abstraction, Inheritance and Polymorphism.

(i) **Objects.** The word “Object” is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behaviour) to either examine or affect this state. Hence, an object is characterised by number of operations and a state which remembers the effect of these operations [JACO98].

All objects have unique identification and are distinguishable. Two bananas may be of same colour, shape and texture but still are different. Each has an identity. There may be four dogs of same colour, breed & size but all are distinguishable. The term identity means that objects are distinguished by their inherent existence and not by descriptive properties [JOSHO3].

As discussed above, object is an entity, which has a state (whose representation is hidden from other objects) and a defined set of operations, which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects, which request these services when some computation is required. In principle, objects communicate by passing messages to each other and these messages initiate object operations.

(ii) **Messages.** Conceptually, objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. In some distributed systems, object communications are implemented directly as text messages which objects exchange. The receiving object parses the message, identifies the service and the associated data and carries out the requested service. Messages are often implemented as procedure or function calls (name = procedure name, information = parameter list).

(iii) **Abstraction.** In object oriented design, complexity is managed using abstraction. *Abstraction is the elimination of the irrelevant and the amplification of the essentials.*

We can teach someone to drive any car using an abstraction. We amplify the essentials: we teach about the ignition and steering wheel, and we eliminate the details, such as details of the particular engine in this car or the way fuel is pumped to the engine where a spark ignites it, it explodes pushing down a piston and driving a crankshaft [FIEL01].

Problems usually have levels of abstraction. We can see a car at a high level of abstraction for driving, but mechanics need to work at a lower level of abstraction. They do care about details and need to know about batteries and engines.

However, there are abstractions at the mechanic's level too. The mechanic might test or charge a battery without caring that inside the battery there is a complex chemical reaction going on. A battery designer would care about these details but would not care about, say, the electronics that goes into the car's stereo.

We have seen that we can look at the details such as the battery or stereo design, and they are separated in manageable chunks, and by using abstraction and ignoring the details, we can also look at the whole car as a manageable chunk.

(iv) **Class.** In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping is known as a class. Hence, a class is a set of objects that share a common structure and a common behaviour.

We may have a class “car” with objects like Indica, Santro, Maruti, Indigo. These objects are related due to some common characteristics to constitute a class named “car”. A class may be object defined as [JACO98]:

“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 object oriented system, each object belongs to a class. An object, that belongs to a certain class is called an instance of that class. We often use object & instance as synonyms. Hence, an instance is an object created from a class. The class describes the structure (behaviour and information) of the instance, while the current state of the instance is defined by the operations performed on the instance.

We may define a class “car” and each object that represents a car becomes an instance of this class. In this class “car”, Indica, Santro, Maruti, Indigo are instances of this class as shown in Fig. 5.20.

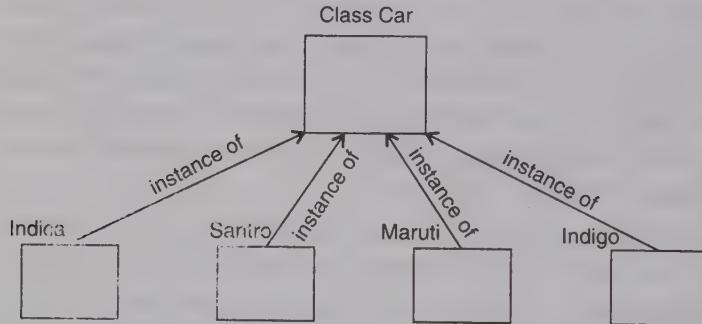


Fig. 5.20: Indica, Santro, Maruti, Indigo are all instances of the class “car”.

We may have different types of classes depending upon common characteristics. The class is the static description and the object is an instance in runtime of that class.

Imagine a picture made up of squares. Each square is an object. It has a state: its colour and position, and behaviour. We can, amongst other things, change its colour and draw it. Each square is different but has much in common with other squares. So, we abstract out the commonalities: they share the same behaviour and have same sort of attributes. We have ignored same sort of attributes. We have ignored the particular values of the attributes.

Classes are useful because they act as a blueprint for objects. If we want a new square we may use the square class and simply fill in the particular details (*i.e.*, colour and position). Fig. 5.21 shows how we can represent the square class.

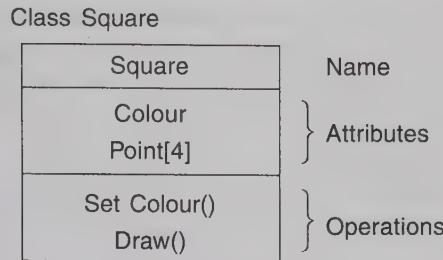


Fig. 5.21: The square class.

(v) **Attributes.** An attribute is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attribute has a value for each object instance. For example, colour may be different in different objects and “array of points” size may also be different in different squares. The attributes are shown as second part of the class as shown in figure 5.21.

(vi) **Operations.** An operation is a function or transformation that may be applied to or by objects in a class. In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. Each operation has a target object as an implicit argument. The behaviour of the operation depends on the class of its target. An object “knows” its class, and hence the right implementation of the operation [JOSHO3]. Operations are shown in the third part of the class as indicated in Fig. 5.21.

(vii) **Inheritance.** Imagine that, as well as squares, we have triangle class. Fig. 5.22 shows the class for a triangle.

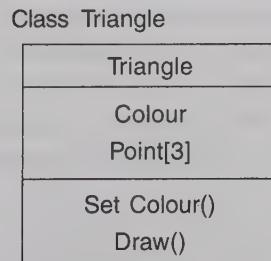


Fig. 5.22: The triangle class.

Now, comparing Fig. 5.21 and Fig. 5.22, we can see that there is some difference between triangle and squares classes. Triangles have three vertices ; squares have four. Also, the way that these shapes are drawn is different. However, there are some similarities. For example, we can set the colour of both and both can be drawn (even if the way they are drawn is different). It would be nice if we could abstract; eliminate the details of each shape and amplify the fact that both can have their colours set and can be drawn. For example, at a high level of abstraction, we might want to think of a picture as made up of shapes and to draw the picture, we draw each shape in turn. We want to eliminate the irrelevant details: we do not care that one shape is a square and the other is a triangle as long as both can draw themselves.

To do this, we consider the important parts out of these classes in to a new class called Shape. Fig. 5.23 shows the results.

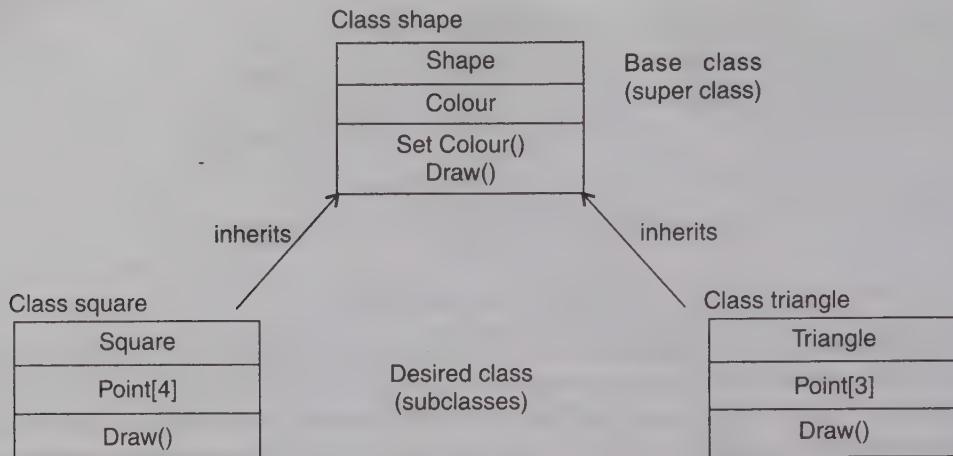


Fig. 5.23: Abstracting common features in a new class.

This sort of abstraction is called inheritance. The low level classes (known as subclasses or derived classes) inherit state and behaviour from this high level class (known as a super class or base class).

Hence, a triangle object will have a colour, a setcolour() operation, three point and a draw() operation. We can inherit three sorts of things:

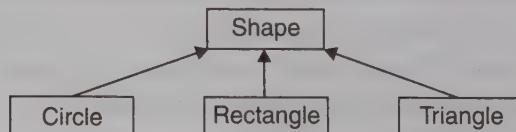
- (i) State: colour
- (ii) Operations: setcolour() should be able to set the colour for any shape
- (iii) The interface of an operation.

Shape contains the interface of the draw() operation because draw() interface for triangle and square is identical but code for the operation remains in the subclasses because it is different.

(viii) Polymorphism. When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

So, we can abstract by pulling out important state, behaviour and interface into a new class. We can also abstract by combining object's inside a new object. In the car example, a car combines a battery, engine and other objects into a new object and provides a simple interface for driving that hides these details. There are different sorts of abstraction and finding the best ways to apply abstraction to a problem is what design is all about.

Another example may be for a base class *shape*, polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the *area* method to it will return the correct results. Polymorphism is considered to be a requirement of any true object-oriented programming language (OOPL).



(ix) **Encapsulation (Information Hiding).** Encapsulation is also commonly referred to as “Information Hiding. It consists of the separation of the external aspects of an object from the internal implementation details of the object. The external aspects of an object are accessible by other objects through methods of object, while the internal implementation of those methods are hidden from the external object sending the message. Thus, it is possible to change the implementation without updating the clients as long as the interface is unchanged. Clients will not be affected by changes in implementation, thus reducing the “ripple effect” in which a correction to one operation forces the corresponding correction in a client operation which in turn causes a change in a client of the client. This makes maintenance is easier and less expensive.

Encapsulation deals with permitting or restricting a client class’ ability to modify the attributes or invoke the methods of the class or object of concern. If a class allows another to modify its attributes or invoke its methods, the attributes and methods are said to be part of the class’ public interface. If a class doesn’t allow another to modify its attributes or invoke its methods, those are part of the class’ private implementation. Thus, Encapsulation protects a) an object’s internal state from being corrupted by its clients and b) Client code from changes in the object’s implementation.

A “Queue” provides a good example of this characteristic. A queue is an abstract concept that represents an ordered list of things. The implementation of a queue may be an array or it may be by a linked-list. If the implementation were known, a developer writing a client of the queue class may use this knowledge and directly access the internal storage mechanism. If the implementation changed, the client would then have to be modified also. This type of tight coupling between classes would cause a very brittle system and would increase maintenance costs as parts of the system were modified. Therefore, the levels of encapsulation that a language supports and how those mechanisms are used directly impacts the level of coupling between classes and it can significantly affect the cost of maintenance in an application.

(x) **Hierarchy.** Hierarchy involves organizing something according to some particular order or rank (e.g., complexity, responsibility, etc.). It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way. This hierarchy is implemented in software via a mechanism called “Inheritance”. Just as a child inherits genes from its parent, a class can inherit attributes and behaviours from its parent. The parent class is commonly referred to as the supper-class and the child class as the sub-class. *Classes at the same level of the hierarchy should be at the same level of abstraction.*

Using a hierarchy to describe differences or variations of a particular concept provides for more descriptive and cohesive abstractions, as well as a better allocation of responsibility. In any one system, there may be multiple abstraction hierarchies (e.g., in a financial application, you may have different types of customers and accounts). Generalization can be used to realize a hierarchy within an object-oriented system, starting at the most general of the abstractions, and then defining more specialized abstractions through sub-classing. Generalization can take place in several stages, which lets you model complex, multilevel inheritance hierarchies. General properties are placed in the upper part of the inheritance hierarchy, and

special properties lower down. In other words, you can use generalization to model specialization of a more general concept.

This relationship can be easily understood by the Fig. 5.24 below.

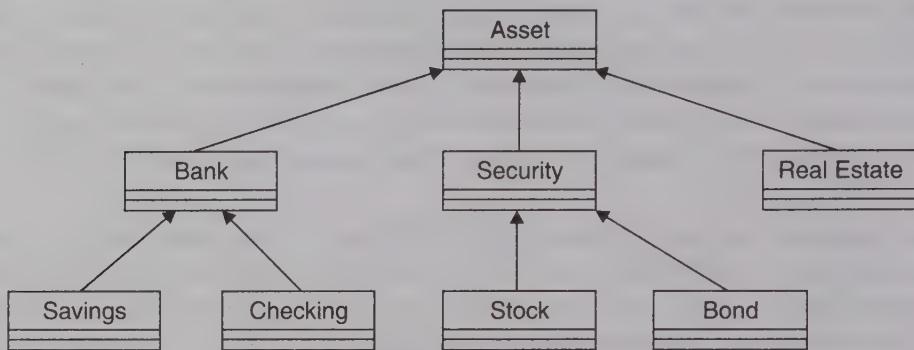


Fig. 5.24: Hierarchy.

In summary, with object-oriented design, we use abstraction to break a problem into manageable chunks. We can comprehend the problem as a whole or study parts of the problem at lower level of abstraction.

If we work at an appropriate level of abstraction then any problems we find can be solved relatively easily. As an example of not working at the right level of abstraction, imagine that we tried to build a house by going out with some bricks and just building. The chances are we would finish, try to put the bath in and discover that the bathroom is too small. So, we have to knock down and rebuild a wall. Meaning, thereby, lot of work, assuming it could be done. If we had designed the house including, where the fittings were going. We would have noticed the problem and fixing it would have taken about 30 seconds with our eraser and pencil.

5.6.2 Steps to Analyze and Design Object Oriented System

There are various steps in the analysis and design of an object oriented system and are given in Fig. 5.25.

(i) **Create use case model.** First step is to identify the actors interacting with the system. We should then write the use case and draw the use case diagram.

(ii) **Draw activity diagram (If required.)**

Activity Diagram illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. It is essentially like a flow chart. Fig. 5.26 shows the activity diagram processing an order to deliver some goods.

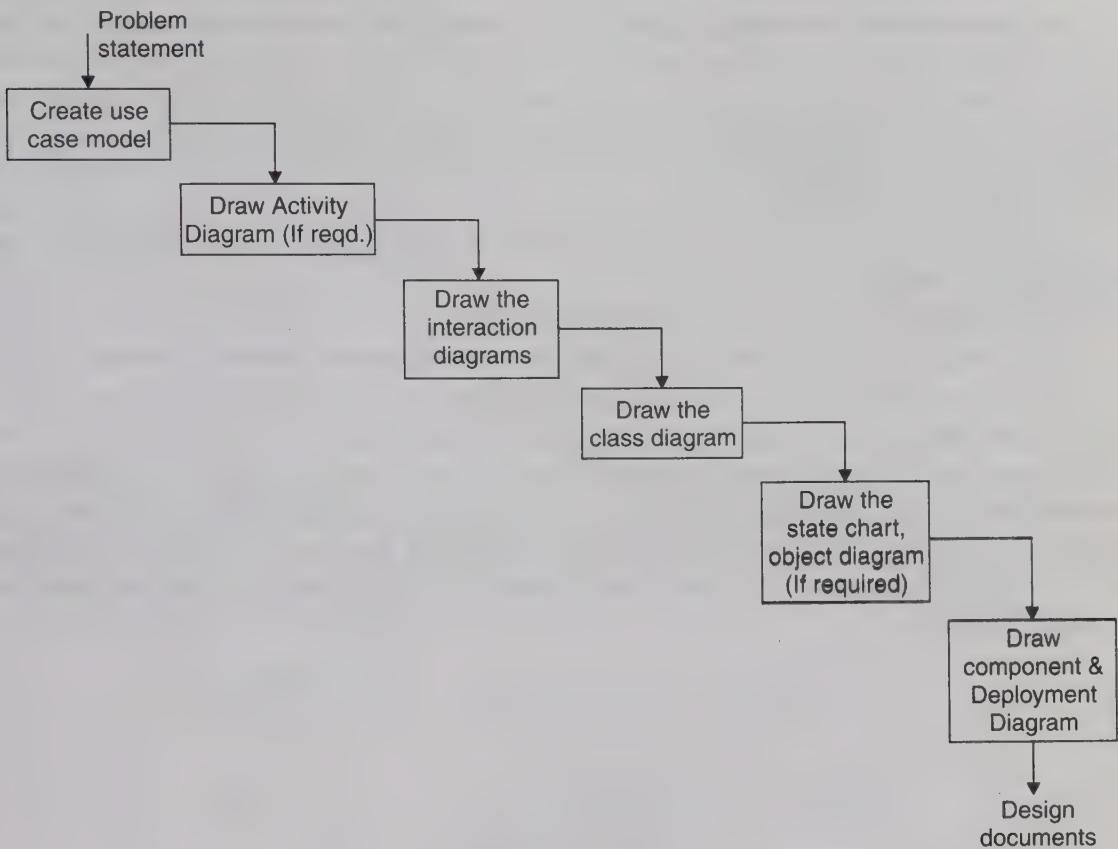


Fig. 5.25: Steps for analysis & design of object oriented system.

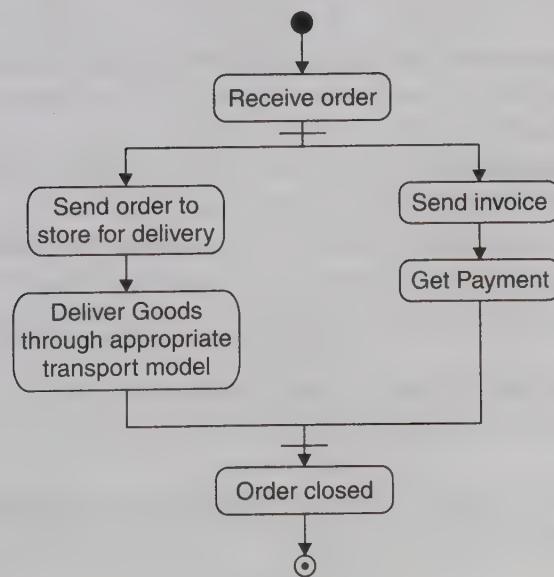


Fig. 5.26: Activity diagram.

(iii) **Draw the interaction diagram:** An interaction diagram shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time ordering of messages; a collaboration diagram is an interaction diagram that emphasizes the structural organisation of the objects that send and receive messages.

Sequence diagrams and collaboration diagrams are isomorphic, meaning that we can take one and transform it into the other.

Steps to draw interaction diagrams are as under :

- (a) Firstly, we should identify that the objects with respect to every use case.
- (b) We draw the sequence diagrams for every use case.
- (c) We draw the collaboration diagrams for every use case.

Many object analysis techniques define only one type of object, which can be used anywhere in the system. Jacobson *et.al* [JACO98] have chosen to use three object types. The reason for this is to have a structure that is more flexible and adaptable to changes. The object types used in this analysis model are entity objects, interface objects and control objects as given in Fig. 5.27.

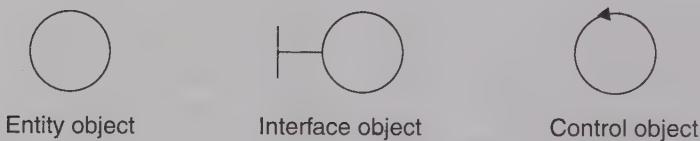


Fig. 5.27: Object types.

The entity object models information in the system that should be held for a longer time, and should typically survive a use case. All behaviour naturally coupled in this information should be placed in the entity object. For a login use case, login-detail may be the entity object. The interface object models behaviour & information that is dependent on the interface to the system. Thus everything concerning any interface to the system is placed in an interface object. For example login screen of the login use case.

(iv) **Draw the class diagram:** The class diagram shows the relationship amongst classes. There are four types of relationships in class diagrams.

(a) **Association** are semantic connection between classes. When an association connects two classes, each class can send messages to the other in a sequence or a collaboration diagram. Associations can be bi-directional or unidirectional.



(b) **Dependencies** connect two classes. Dependencies are always unidirectional and show that one class depends on the definitions in another class.



(c) **Aggregations** are stronger form of association. An aggregation is a relationship between a whole and its parts.

(d) **Generalizations** are used to show an inheritance relationship between two classes.



We may design the class diagram of a complete system. Obviously, this may become very huge, for complex systems. In such a case, one class diagram is made for each use case. This restricts the size of the class diagram and is useful when delegating/assigning the task to a software developer ; since he/she would only need to see the related class diagram of the assigned use case; rather than looking at the complete class diagram and then filtering out the concepts related to his/her use-case. We may also draw object diagram, which shows a set of objects and their relationships. They represent static snapshots of instances of the things found in class diagrams, from the perspective of real cases.

(v) **Design of state chart diagrams:** A state chart diagram is used to show the state space of a given class, the event that cause a transition from one state to another, and the actions that result from a state change. It shows the changing behaviour of an object in response to different events. State transition diagrams are made only for the objects that are either very complicated or have a very dynamic behaviour with respect to various events. Normally, we would not need to make state diagrams. A state transition diagram for a "book" in the library system is given in Fig. 5.28.

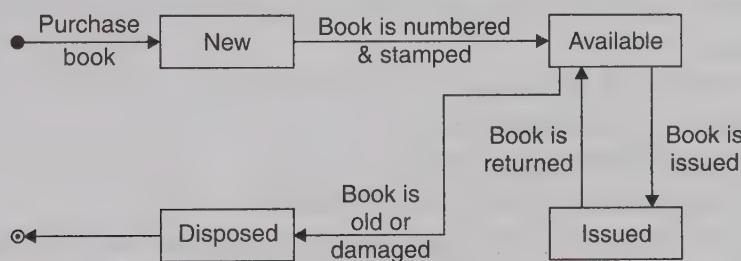


Fig. 5.28: Transition chart for 'book' in a library system.

(vi) **Draw component and development diagram:** Component diagrams address the static implementation view of a system they are related to class diagrams in that a component typically maps to one or more classes, interfaces or collaboration. Deployment Diagram Captures relationship between physical components and the hardware.

After the completion of above mentioned steps, we will have many documents and complete understanding of flow of data, control and relationships of classes. These things are the foundations for implementing an object oriented system.

5.6.3 Case Study of Library Management System

The problem statement for library management system is given below:

Problem statement

A software has to be developed for automating the manual library of a University. The system should be stand alone in nature. It should be designed to provide functionality's as explained below:

Issue of Books:

- A student of any course should be able to get books issued.
- Books from General Section are issued to all but Book bank books are issued only for their respective courses.
- A limitation is imposed on the number of books a student can issue.
- A maximum of 4 books from Book bank and 3 books from General section is issued for 15 days only.
- The software takes the current system date as the date of issue and calculates date of return.
- A bar code detector is used to save the student as well as book information.
- The due date for return of the book is stamped on the book.

Return of Books:

- Any person can return the issued books.
- The student information is displayed using the bar code detector.
- The system displays the student details on whose name the books were issued as well as the date of issue and return of the book.
- The system operator verifies the duration for the issue.
- The information is saved and the corresponding updating take place in the database.

Query Processing:

The system should be able to provide information like:

- Availability of a particular book.
- Availability of book of any particular author.
- Number of copies available of the desired book.

The system should also be able to generate reports regarding the details of the books available in the library at any given time. The corresponding printouts for each entry (issue/return) made in the system should be generated. Security provisions like the 'login authenticity' should be provided. Each user should have a user id and a password. Record of the users of the system should be kept in the log file. Provision should be made for full backup of the system.

Use cases

From the problem description, we can see that the system has four actors.

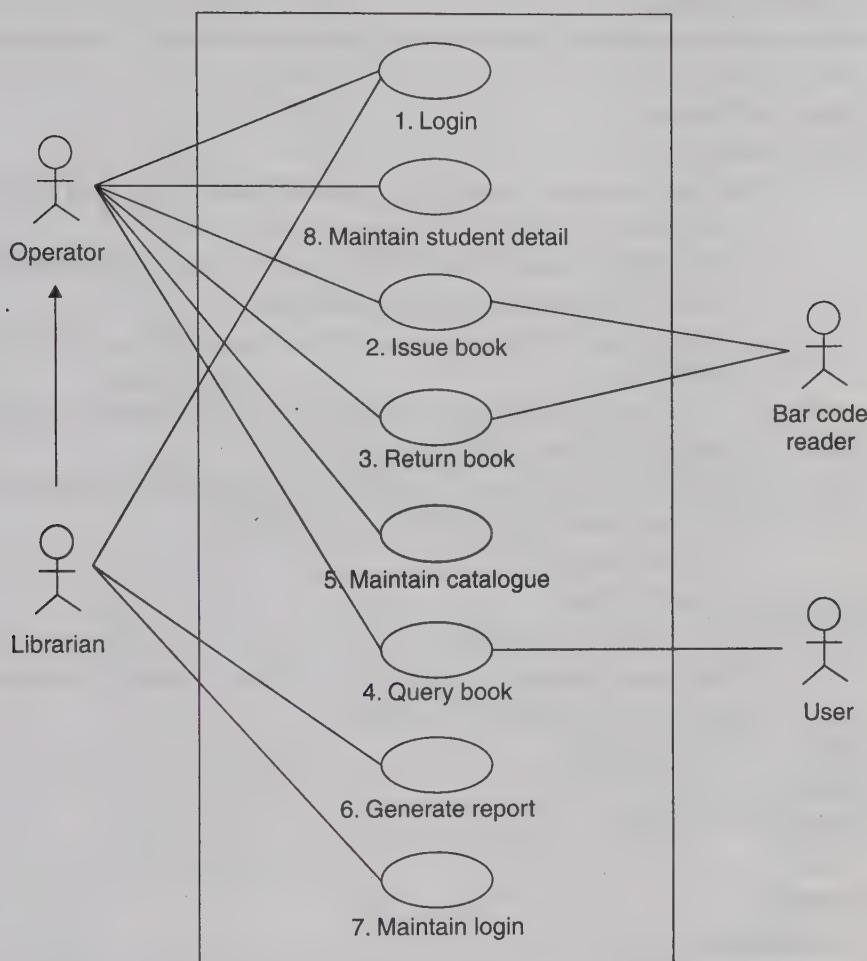
1. Librarian
2. Operator
3. Bar Code Reader
4. User

To define system's functionalities, we can view the system as a collection of following use cases:

- | | |
|----------------|---------------|
| 1. Login | 2. Issue Book |
| 3. Return Book | 4. Query Book |

5. Maintain Catalog
7. Maintain Login

6. Generate Report
8. Maintain student Details



Use case diagram for library management system

Use case description

1. LOGIN

1.1. Introduction

This use case documents the procedure for logging into the Library Management System based on user privileges.

- Operator (Issue Book, Return Book, Query a Book, Maintain Catalogue, Maintain Student Detail)
- Librarian (Generate Reports, Maintain Login)

1.2 Actors

Operator, Librarian

1.3 Pre-Condition

None

1.4 Post-Condition

If use case is successful, the user is logged into the system, otherwise the system state is unchanged.

1.5 Flow of Events

1.5.1 Basic Flow

This use case starts when actor wishes to log in to the Library Management System.

- The system requests that the actor enters his/her user_id and password.
- The actor enters user_id and password.
- The system validates the user_id and password and checks for his/her privileges.
- If the user is “operator”, he/she will be logged into the system and presented with operator’s menu.
- Otherwise, if the user is “librarian”, he/she will be logged into the system and presented with librarian’s menu.
- The use case ends.

1.5.2 Alternate Flow

1.5.2.1 Invalid Name / Password

If the system receives an invalid user_id or password, an error message is displayed and the use case ends.

1.6 Special Requirements

None

1.7 Related Use Cases

None

2. ISSUE BOOK

2.1 Introduction

This use case documents the procedure of issuing a book for following accounts:

- General (for 15 days)
- Book Bank (for the semester)

2.2 Actors

Operator, Barcode reader

2.3 Pre-Condition

Operator must be logged in to the system

2.4 Post-Condition

If use case is successful, the book is issued to the student in his/her general or book bank account, otherwise the system state is unchanged.

2.5 Flow of Events

2.5.1 Basic Flow

The use case starts when a student wants to get a book issued.

- The system reads and validates the student's information using the Bar Code Reader.
- The system reads the book's information using the Bar Code Reader.
- The return date of the book is calculated as per the account in which the student wishes to get the book issued – 15 days for General account and whole semester for Book Bank account.
- The book and student's information is saved into the database.
- The issue details are sent to the printer to generate the receipt.
- The use case ends.

2.5.2 Alternate Flow

2.5.2.1 Unauthorized Student

If the system doesn't validate the student, then an error message is flagged and the use case ends.

2.5.2.2 General Account is Full

If the student has requested a book in General Account and the later is full, i.e. he has already 3 books issued on his name, then the request for issue is denied and the use case ends.

2.5.2.3 Book Bank Account is Full

If the student has requested a book in Book Bank Account and the later is full, i.e. he has already 4 books issued on his name, then an error message is shown and the use case ends.

2.5.2.4 Course Mismatch between Student and the Book Bank Book.

If the student of a particular course has requested a book from some other course on Book Bank Account, then an option for getting the book issued in general account is given and the use case ends.

2.6 Special Requirements

None

2.7 Related Use Cases

Generate Barcode

3. RETURN BOOK

3.1 Introduction

This use case documents the procedure of returning a book and calculating the fine amount if the student has returned the book after the specified return date.

3.2 Actors

Operator, Barcode reader

3.3 Pre-Condition

Operator must be logged in to the system.

3.4 Post-Condition

If use case is successful, the book is returned back to the library and if needed, the fine is calculated, otherwise the system state is unchanged.

3.5 Flow of Events

3.5.1 Basic Flow

This use case starts when a student wants to return a book.

- The system reads the book's information using the Bar Code Reader.
- The book is returned to the library.
- The database entries corresponding both to the student account and the book are updated.
- The return details are sent to the printer to generate the receipt.
- The use case ends.

3.5.2 Alternate Flow

3.5.2.1 Late return of book

If the book is returned after the due date, fine is calculated and database is updated accordingly. The use case ends here.

3.6 Special Requirements

None

3.7 Related Use Cases

Generate Barcode

4. QUERY A BOOK

4.1 Introduction

This use case documents the procedure for searching a book based on the specified criteria, which are:

- Search by Author Name
- Search by Title Name

4.2 Actors

Operator, user

4.3 Pre-Condition

Operator user must be logged in to the system

4.4 Post-Condition

If use case is successful, the book details are displayed.

4.5 Flow of Events

4.5.1 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.
- The result is displayed to the user.
- The use case ends.

4.6 Special Requirements

None

4.7 Related Use Cases

None

5. MAINTAIN CATALOG

5.1 Introduction

This use case documents the procedure for updating the catalog of the library.

5.2 Actors

Operator

5.3 Pre-Condition

Operator must be logged into the system.

5.4 Post-Condition

If use case is successful, the book should be updated, otherwise the system state in unchanged.

5.5 Flow of Events

5.5.1 Basic Flow

This use case starts when the operator wishes to add , delete or modify some details in the library.

- The corresponding changes are saved in the database.
- The use case ends.

5.5.2 Alternate Flow

None

5.6 Special Requirements

None

5.7 Related Use Cases

None

6. GENERAL REPORTS

6.1 Introduction

This use case documents the procedure for generating the reports as desired by the Librarian.

6.2 Actors

Librarian

6.3 Pre-Condition

Librarian must be logged into the system.

6.4 Post-Condition

If use case is successful, the various reports, regarding the details of the books available in the library at any given time, are generated.

6.5 Flow of Events

6.5.1 Basic Flow

This use case starts when a librarian wants to generate reports of the books available in the library.

- The system displays the various report generating criteria to the user, which can be the books issued to the students at a particular time, books available in the library etc.

- The librarian selects the criteria and enters the various parameters based on the criteria selected.
- The system generates the report and sends that to printer.
- The use case ends.

6.5.2 Alternate Flow

6.5.2.1 Printer out of paper or low on ink

If the printer goes out of paper or low on ink, then the printing operation is aborted and the necessary action needs to be taken, which can be feeding paper to the printer or replacing the ink cartridge. The use case ends.

6.6 Special Requirements

None

6.7 Related Use Cases

None

7. MAINTAIN LOGIN

7.1 Introduction

This use case documents the procedure for maintaining Login Details.

7.2 Actors

Librarian.

7.3 Pre-Condition

Librarian must be logged into the system.

7.4 Post-Condition

If use case is successful, the Login details should be updated, otherwise the system state is unchanged.

7.5 Flow of Events

7.5.1 Basic Flow

This use case starts when the Librarian wishes to add, delete or modify some details of login.

- The corresponding changes will be done.
- The use case ends.

7.5.2 Alternate Flow

None

7.6 Special Requirements

None

7.7 Related Use Cases

None

8. MAINTAIN STUDENT DETAILS

8.1 Introduction

This use case documents the procedure for maintaining Student Details.

8.2 Actors

Operator

8.3 Pre-Condition

Operator must be logged into the system.

8.4 Post-Condition

If use case is successful, the Student Details should be updated, otherwise the system state is unchanged.

8.5 Flow of Events

8.5.1 Basic Flow

This use case starts when the operator wishes to add, delete or modify some details of student.

- The corresponding changes will be done.
- The use case ends.

8.5.2 Alternate Flow

None

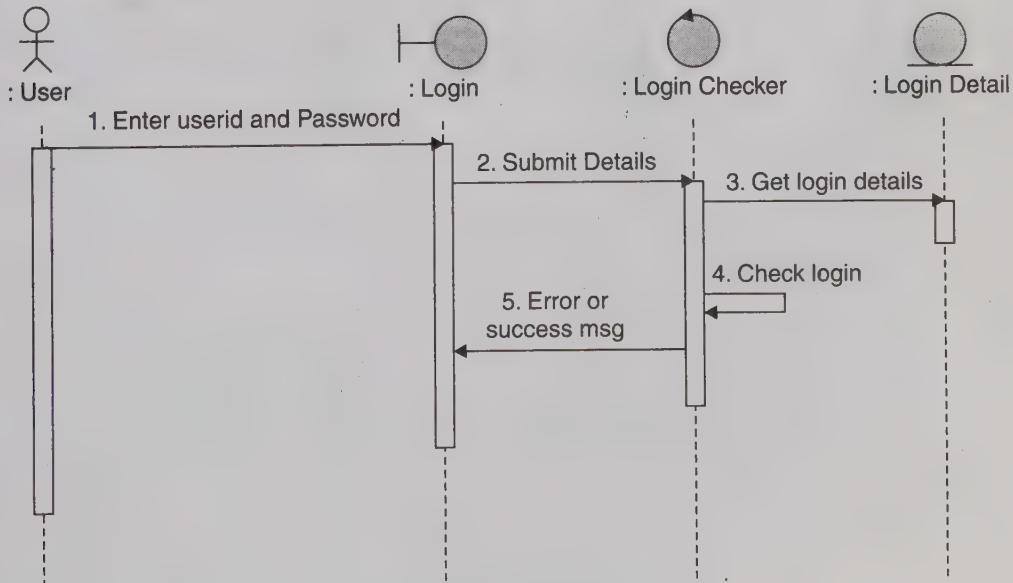
8.6 Special Requirements

None

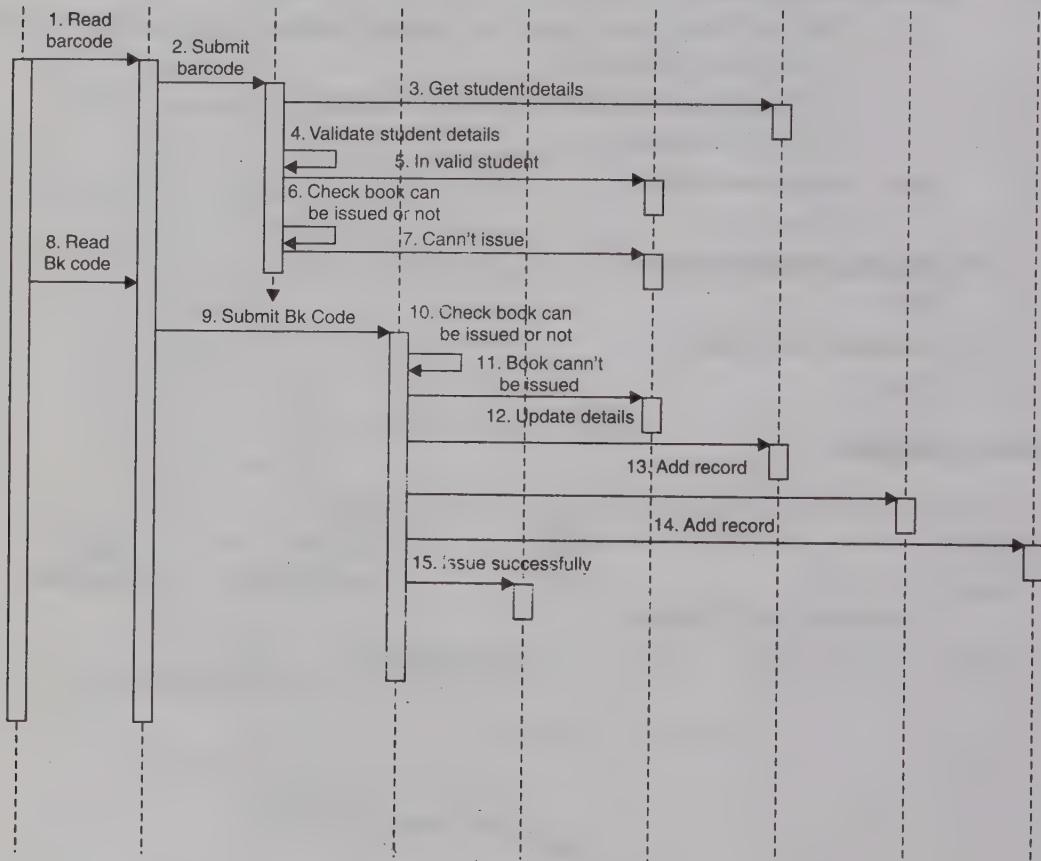
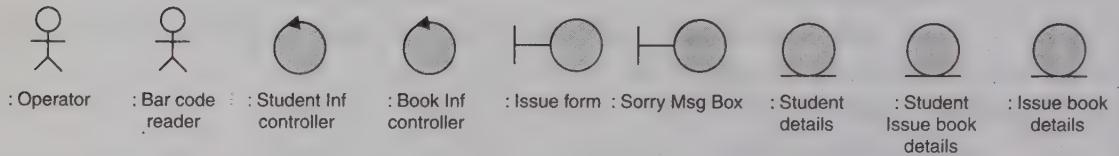
8.7 Related Use Cases

None

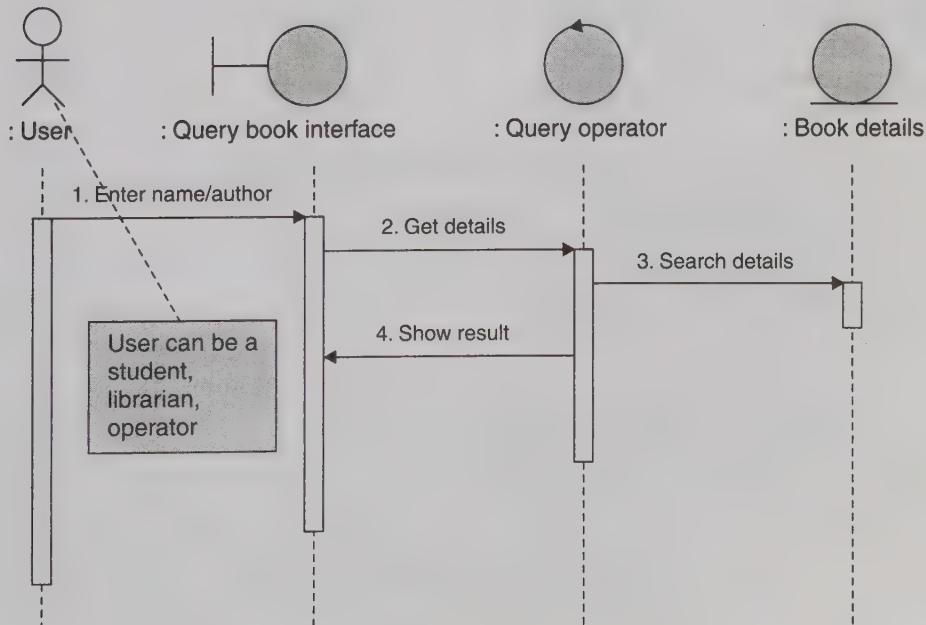
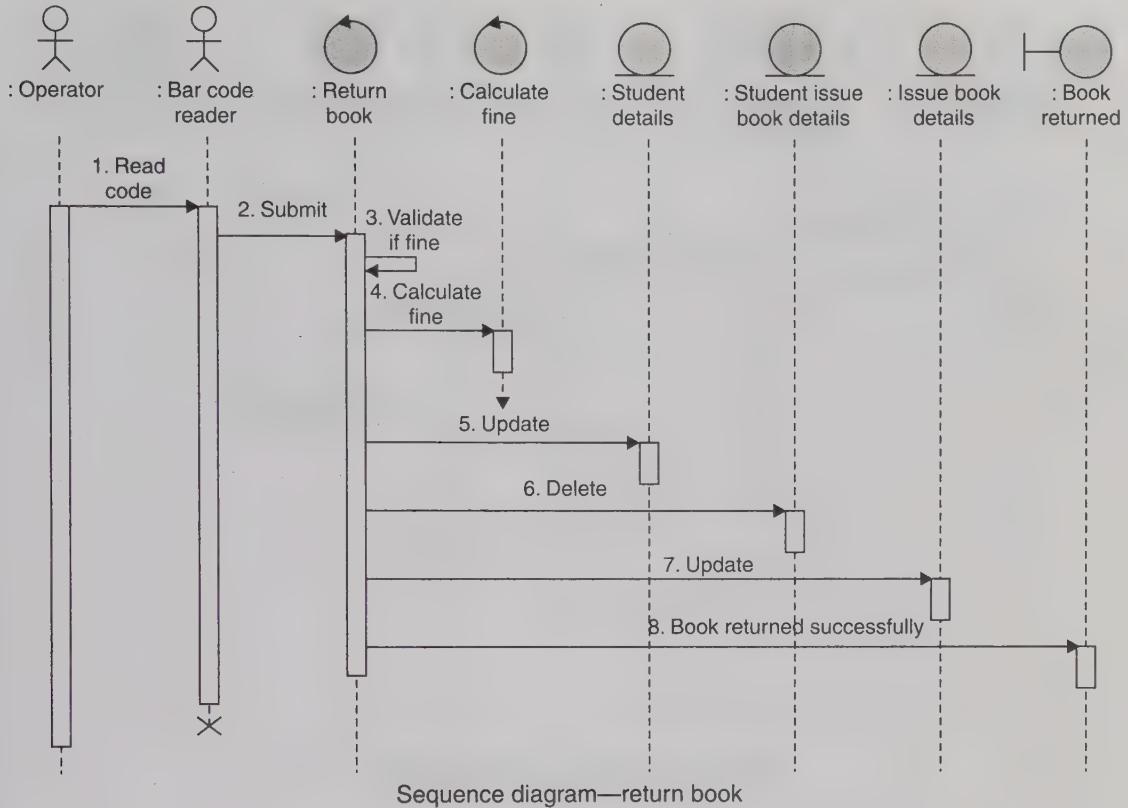
Sequence diagrams

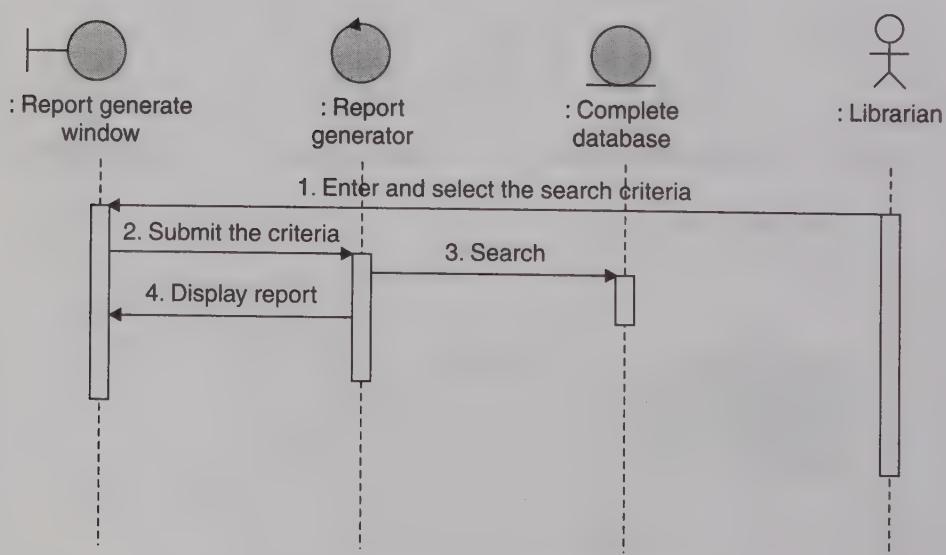
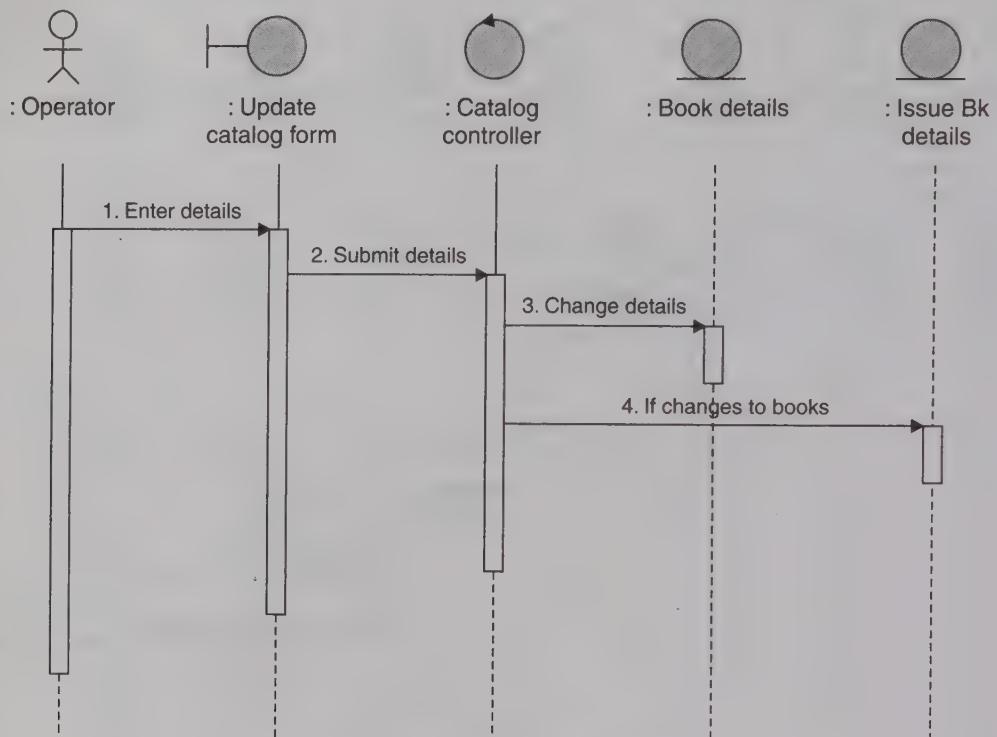


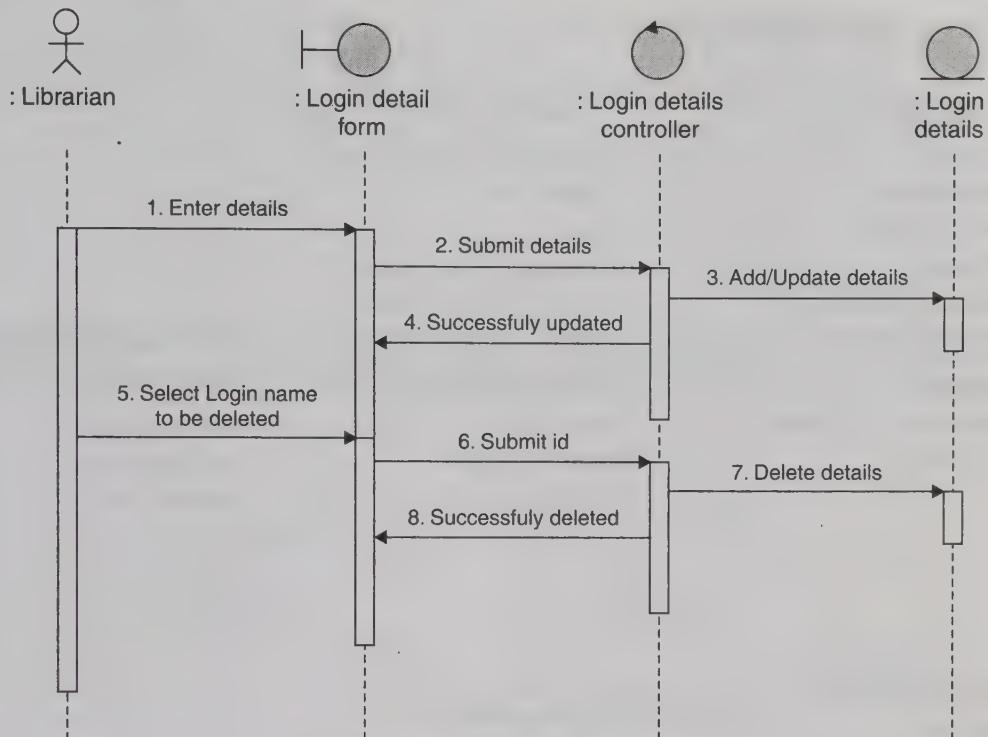
Sequence diagram—Login



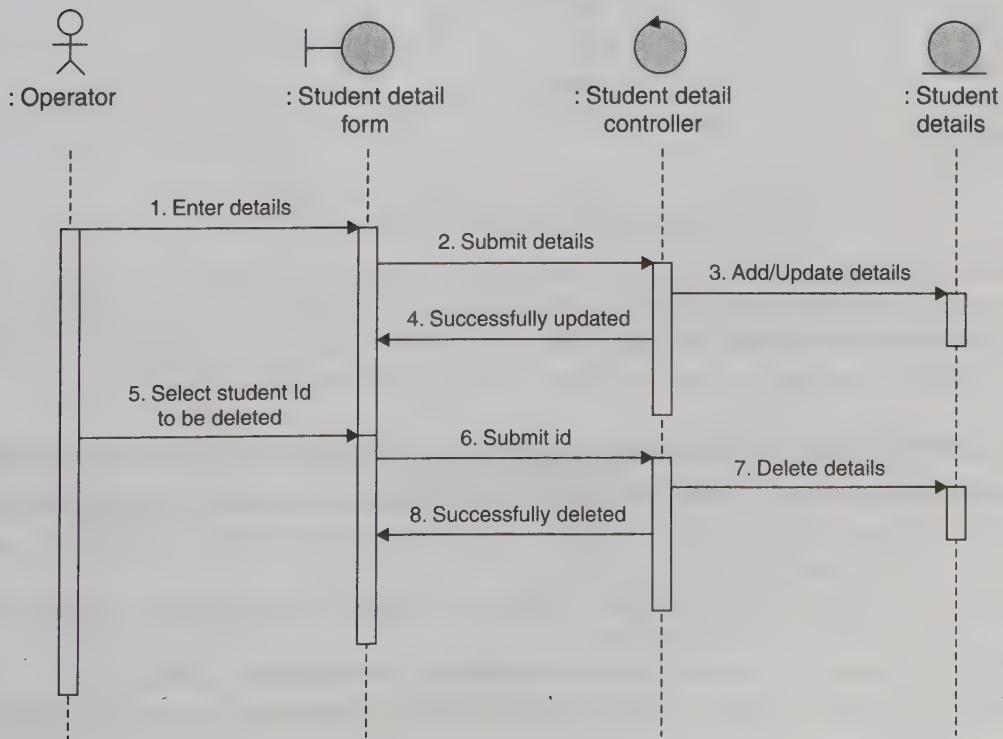
Sequence diagram—issue book





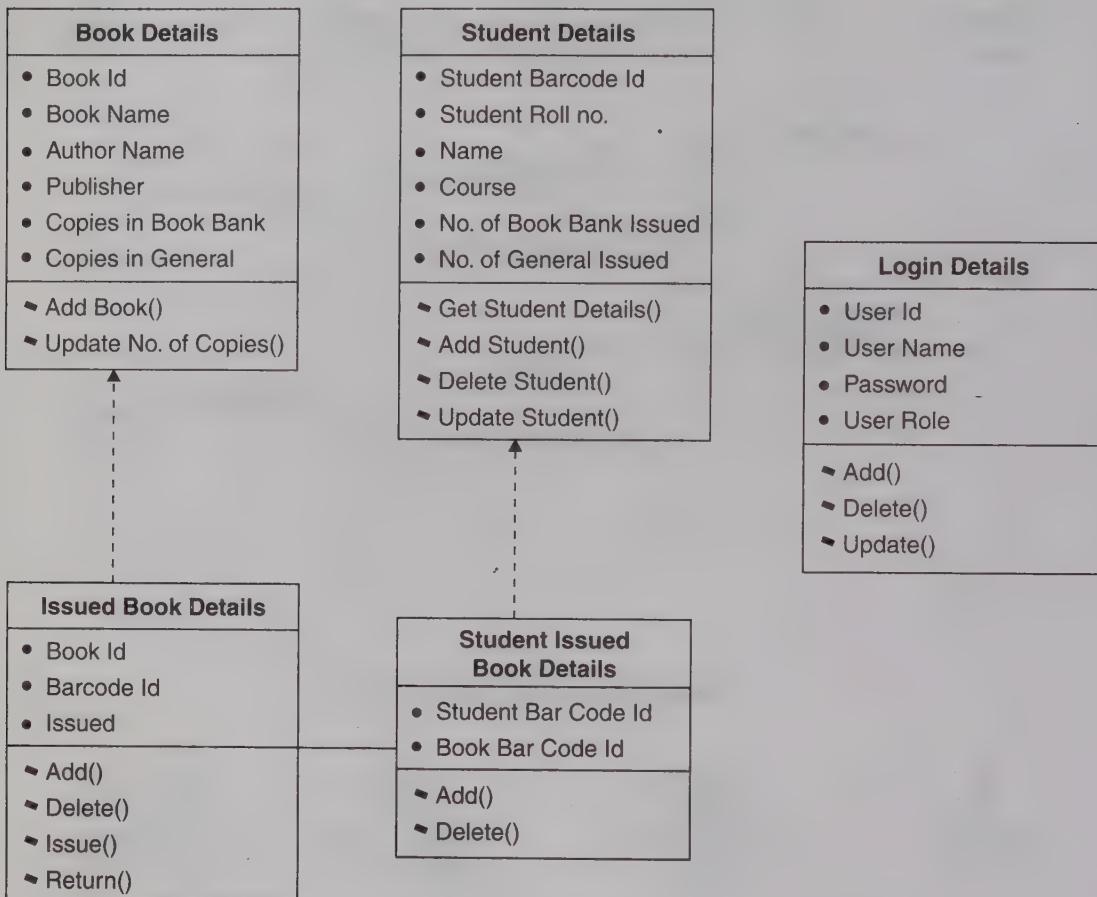


Sequence diagram—maintain login



Sequence diagram—maintain student details

Class diagram of entity classes



Class diagram of entity classes

In this design, we have shown class diagram of entity classes only. These entity classes generally becomes tables in the database. The other diagrams of the design document can be drawn using Rational Rose facilities (or other tools) and may or may not be required in each case study. All these diagrams are the foundations of the implementation phase.

REFERENCES

- [KHAL01] Beth Khalil, "Software Design and Development", Homepage, <http://attila.stevens-tech.edu>, 2001.
- [BOOC03] Booch G., "Object Oriented Analysis and Design with Applications" Pearson Edition, 2003
- [FAIR2K] Fairly R., "Software Engineering Concepts", Tata McGraw Hill Pub., 2000.
- [JACO98] Jacobson I., et al., "Object Oriented Software Engineering", Addison Wesley 1998.

- [JOSH03] Joshi S.D., "object oriented modelling and design", Tech-Max, 2003.

[MYER78] Myers C., "Composite Structured Design", Van Nostrand, Reinhold, 1978.

[FIEL01] Paul Field, "An Introduction to Object Oriented Design", Paulfield @dial.pipex.com, 2001.

[JALO98] Pankaj Jalote, "An Integrated Approach to Software Engineering", Narosa Publications, Delhi, 1998.

[PFLE98] Pfleeger S.L., "Software Engineering", Prentice-Hall International, Inc, 1998.

[SOMM2K] Sommerville I., "Software Engineering", Addison-Wesley, 2000.

MULTIPLE CHOICE QUESTIONS

EXERCISES

- 5.1.** What is design ? Describe the difference between conceptual design and technical design.
 - 5.2.** Discuss the objectives of software design. How do we transform an informal design to a detailed design ?
 - 5.3.** Do we design software when we “write” a program ? What makes software design different from coding ?
 - 5.4.** What is modularity ? List the important properties of a modular system.
 - 5.5.** Define module coupling and explain different types of coupling.
 - 5.6.** Define module cohesion and explain different types of cohesion.
 - 5.7.** Discuss the objectives of modular software design. What are the effects of module coupling and cohesion ?
 - 5.8.** If a module has logical cohesion, what kind of coupling is this module likely to have with others ?
 - 5.9.** What problems are likely to arise if two modules have high coupling ?
 - 5.10.** What problems are likely to arise if a module has low cohesion ?
 - 5.11.** Describe the various strategies of design. Which design strategy is most popular and practical ?
 - 5.12.** If some existing modules are to be re-used in building a new system, which design strategy is used and why ?
 - 5.13.** What is the difference between a flow chart and a structure chart ?
 - 5.14.** Explain why it is important to use different notations to describe software designs.
 - 5.15.** List a few well-established function oriented software design techniques.
 - 5.16.** Define the following terms: Objects, Messages, Abstraction, Class, Inheritance and Polymorphism.
 - 5.17.** What is the relationship between abstract data types and classes ?
 - 5.18.** Can we have inheritance without polymorphism ? Explain.
 - 5.19.** Discuss the reasons for improvement using object-oriented design.
 - 5.20.** Explain the design guidelines that can be used to produce “good quality” classes or reusable classes.
 - 5.21.** List the points of a simplified design process.
 - 5.22.** Discuss the differences between object oriented and function oriented design.
 - 5.23.** What documents should be produced on completion of the design phase ?
 - 5.24.** Can a system ever be completely “decoupled” ? That is, can the degree of coupling be reduced so much that there is no coupling between modules ?

6

Software Metrics

Contents

6.1 Software Metrics: What & Why ?

- 6.1.1 Definition
- 6.1.2 Areas of Applications
- 6.1.3 Problems During Implementation
- 6.1.4 Categories of Metrics

6.2 Token Count

- 6.2.1 Estimated Program Length
- 6.2.2 Potential Volume
- 6.2.3 Estimated Program Level/Difficulty
- 6.2.4 Effort & Time
- 6.2.5 Language Level

6.3 Data Structure Metrics

- 6.3.1 The Amount of Data
- 6.3.2 The Usage of Data within a Module
- 6.3.3 Program Weakness
- 6.3.4 The Sharing of Data Among Modules

6.4 Information Flow Metrics

- 6.4.1 The Basic Information Flow Model
- 6.4.2 A more Sophisticated Information Flow Model

6.5 Metrics Analysis

- 6.5.1 Using Statistics for Assessment
- 6.5.2 Problems with Metrics Data
- 6.5.3 The Common Pool of Data
- 6.5.4 A Pattern for Successful Applications

An eminent physicist, Lord Kelvin said, "when you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you can not measure it, when you can not express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science."

It is clear from Lord Kelvin's statement that everything should be measurable. If it is not measurable, we should make an effort to make it measurable. Thus, the area of measurement is very significant and important in all walks of life. When we discuss metrics for software engineering, situation is entirely different. Some feel that metrics are valuable management and engineering tools, while others feel that they are useless and expensive exercises in pointless data collection.

6.1 SOFTWARE METRICS: WHAT AND WHY ?

Science begins with quantification; we cannot do physics without a notion of length and time; we cannot do thermodynamics until we measure temperature. All engineering disciplines have metrics (such as metrics for weight, density, wave length, pressure and temperature) to quantify various characteristics of their products. The most fundamental question we can ask is "how big is the program"? Without defining what big means, it is obvious that it makes no sense to say, "this program will need more testing than that program" unless we know "how big they are relative to one another. Comparing two strategies also needs a notion of size. The number of tests required by a strategy should be normalized to size. For example A needs 1.4 tests per unit of size, while strategy B needs 4.3 tests per unit of size.

What is meant by size was not obvious in the early phases of science development. Newton's use of mass instead of weight was a breakthrough for physics, and early researchers in thermodynamics had heat, temperature, and entropy hopelessly confused. Size is not obvious for the software. Metrics must be objective in the sense that the measurement process is algorithmic and will yield the same results no matter who applies it [BEIZ90]. To see what kinds of metrics, we need, let us ask some questions.

1. How to measure the size of a software?
2. How much will it cost to develop a software?
3. How many bugs can we expect?
4. When can we stop testing?
5. When can we release the software?

6. What is the complexity of a module?
7. What is the module strength and coupling?
8. What is the reliability at the time of release?
9. Which test technique is more effective?
10. Are we testing hard or are we testing smart?
11. Do we have a strong program or a weak test suite?

If we want an answer to the above questions, we will have to do our own measuring and fit our own empirical laws to the measured data. Most of the metrics are aimed at getting empirical laws that relate program size (however it be measured) to expected number of bugs, expected number of tests required to find bugs, test technique effectiveness, resource requirement, release instant, reliability and quality requirement, etc.

The groundwork of software metrics was laid in the seventies. The earliest paper on the subject was of course published in 1968 [RUBE68]. From these earlier works, interesting results have emerged in the eighties. The term software metrics designates here “a unit of measurement of a software product or software related process [HAME85].”

6.1.1 Definition

Software metrics can be defined as [GOOD93] *“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”*.

This definition covers quite a lot. Software metrics are all about measurements which, in turn, involve numbers, the use of numbers to make things better, to improve the process of developing software and to improve all aspects of the management of that process. Software metrics are applicable to the whole development life cycle from initiation, when costs must be estimated, to monitoring the reliability of the end product in the field, and the way that product changes over time with enhancement. It covers the techniques for monitoring and controlling the progress of the software development, such that the fact that it is going to be six months late is recognized as early as possible rather than the day before delivery is due. It even covers organizations determining which of its software products are the cash cows and which are the dogs.

6.1.2 Areas of Applications

The most established area of software metrics is cost and size estimation techniques. There are many proprietary packages in the market that provide estimates of software system size, cost to develop a system, and the duration of the development or enhancement of the project. These packages are based on estimation models, like COCOMO81, COCOMO-II, developed by Berry Boehm [BOEH81]. Various techniques that do not require the use of ready-made tools are also available. There has been a great deal of research carried out in this area, and this research continues in all important software industries and other organizations. One thing that does come across strongly from the results of this research work is that organizations cannot rely, solely, on the use of proprietary packages.

Controlling software development projects through measurement is an area that is generating a great deal of interest. This has become much more relevant with the increase in fixed price contracts and the use of penalty clauses by customers who deal with software developers.

The prediction of quality levels for software, often in terms of reliability, is another area where software metrics have an important role to play. Again, there are proprietary models in the market that can assist this, but debate continues about their accuracy. The requirement is there both from the customer's point of view and that from the developer's, who needs to control testing and other costs. Various techniques can be used now, and this area will become more and more important in future.

The use of software metrics to provide quantitative checks on software design is also a well-established area. Much research has been carried out, and some organizations have used such techniques to very good effect. This area of software metrics is also being used to control software products, which are in place and are subject to enhancement.

Software metrics are also used to provide management information. This includes information about productivity, quality and process effectiveness. It is important to realize that this should be seen as an ongoing activity. Snapshots of the current situation have their place, but the most valuable information comes when we can see trends in data. Is productivity or quality getting better or worse over time? Why is this happening? What can management do to improve things? The provision of management information is as much an art as a science. Statistical analysis is a part of it, but the information must be presented in a way that managers find it useful, at the right time and for the right reasons. All this shows that software metrics is a vast field and have wide variety of applications throughout the software life cycle [GOOD93].

6.1.3 Problems During Implementation

Implementing software metrics in an organization of any size is difficult. There are many problems that have to be overcome and decisions that have to be made, often with limited information on hand. The first decision concerns the scope of the work. There is a rule in software development that we do not try something new on a large or critical system and this translates, in the software metrics area, to '*do not try to do too much*'. On the other hand, there is evidence that concentrating on too small an area can result in such a limited pay back as to invalidate software metrics in an organization. It is always said, '*do not bet your career on a single metric*'.

Another problem during implementation arises if we start measuring the performance of individuals. There was an organization, which used individual productivity, in terms of functionality divided by effort, as a major determinant of salary increase. While this may appear attractive to some managers, the organization later stated that this was one of the worst mistakes it ever made. Using measurement in this way is counter productive, divisive and simply ensures that developers will rig the data they supply. We may say that management has one chance and one chance only. The first time that a manager uses data supplied by an individual against that individual is the last time that the manager will get accurate or true data from that person. The reasoning behind such a statement is simple, "*employees do not like upsetting the boss!*"

There is a great temptation to seek the “silver bullet”, the single measure that tells all about the software development process. Currently, this does not exist. We must realize that implementing software metrics means changing the way in which people work and think. The software engineering industry is maturing. Customers are no longer willing to accept poor quality and late deliveries. Management is no longer willing to pour money into the black hole of IT, and more management control is now a key business requirement. Competition is growing. Developers have to change and mature as well, and this can be a painful experience as they find tenets of their beliefs being challenged and destroyed. Some simple statements that could be made by many in our industry together with interpretation of current management trends will illustrate the following points [GOOD93].

- **Statement** : Software development is so complex; it cannot be managed like other parts of the organization.
Management view: Forget it, we will find developers and managers who will manage that development.
- **Statement** : I am only six months late with this project.
Management view: Fine, you are only out of a job.
- **Statement** : But you cannot put reliability constraints in the contract.
Management view: Then we may not get the contract.

The list is almost endless and the message is clear.

Software metrics cannot solve all our problems, but they can enable managers to improve their processes, to improve productivity and quality, to improve the probability of survival. But it is not an easy option. Many metrics programs fail. One reason for this is that organizations and individuals who would never dream of introducing a new system without a structured approach ignore the problems of introducing change inherent in software metrics implementation. Only by treating the implementation of software metrics as a project or programme in its own right with plans, budgets, resources and management commitment can make such an implementation succeed. Hence the use of software metrics does not ensure survival, but it improves the probability of survival.

6.1.4 Categories of Metrics

There are three categories of software metrics which are given below:

(i) **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

(ii) **Process metrics:** describe the effectiveness and quality of the processes that produce the software product. Examples are:

- effort required in the process
- time to produce the product
- effectiveness of defect removal during development
- number of defects found during testing
- maturity of the process.

(iii) **Project metrics:** describe the project characteristics and execution. Examples are:

- number of software developers
- staffing pattern over the life cycle of the software
- cost and schedule
- productivity

Some metrics belong to multiple categories like quality metric may belong to all three categories. It focuses on the quality aspects of the product process, and the project. Some important metrics are discussed in subsequent sections of the chapter.

6.2 TOKEN COUNT

Two important size metrics (LOC and Function count) are discussed in chapter 4 (software project planning). These metrics have established their applicability in the field, specifically as a key input in the costing models like COCOMO, COCOMO-II, Putnam resource allocation model etc. The major problem with LOC measure is that it is not consistent because some lines are more difficult to code than others. A program is considered to be a series of tokens and if we count the number of tokens, some interesting results may emerge.

In the early 1970s, the late Professor Maurice Halstead and his co-workers at Purdue University developed the software science family of measures [HALS77]. Tokens are classified as either operators or operands. All software science measures are functions of the counts of these tokens.

Generally, any symbol or keyword in a program that specifies an algorithmic action is considered an operator, while a symbol used to represent data is considered an operand. Most punctuation marks are also categorized as operators. Variables, constants and even labels are operands. Operators consist of arithmetic symbols such as $+$, $-$, $/$, $*$ and command names such as "while", "for", "printf", special symbols such as $:$, braces, parentheses, and even function names such as "eof" (end of file). The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

$$\eta = \eta_1 + \eta_2 \quad (6.1)$$

where η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

The length of the program in terms of the total number of tokens used is

$$N = N_1 + N_2 \quad (6.2)$$

where N : program length.

N_1 : total occurrences of operators

N_2 : total occurrences of operands

It should be noted that N is closely related to the lines of code (LOC) measure of program. For machine language programs where each line consists of one operator and one operand, the program length is

$$N = 2 * LOC \quad (6.3)$$

Additional metrics are defined using these basic terms. Another measure for size of the program is called the volume.

$$V = N * \log_2 \eta \quad (6.4)$$

The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used. Volume may also be interpreted as the number of mental comparisons needed to write a program of length N , assuming a binary search method is used to select a member of the vocabulary of size η . Since an algorithm may be implemented by many different but equivalent programs, a program that is minimal in size is said to have the potential volume V^* . Any given program with volume V is considered to implement at the program level L , which is defined by

$$L = V^* / V \quad (6.5)$$

The value of L ranges between zero and one, with $L = 1$ representing a program written at the highest possible level (*i.e.*, with minimum size). The inverse of the program level is termed the difficulty. That is

$$D = 1 / L \quad (6.6)$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

The effort required to implement a program increases as the size of the program increases. It also takes more effort to implement a program at a lower level (higher difficulty) when compared with another equivalent program at a higher level (lower difficulty). Thus the effort in software science is defined as

$$E = V / L = D * V \quad (6.7)$$

The unit of measurement of E is elementary mental discriminations.

6.2.1 Estimated Program Length

As stated earlier, size of the program is the total number of tokens, referred to as program length, N . The first hypothesis of software science is that the length of a well-structured program is a function only of the number of unique operators and operands. This function, called the estimated length equation is denoted by \hat{N} and is defined by

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \quad (6.8)$$

For example, the sorting program in Fig. 4.2 (given in chapter 4) has 14 unique operators and 10 unique operands. Suppose that these numbers are known before the completion of the program, possibly by using a program design language and knowing the programming language chosen. It is then possible to estimate the length N of the program in number of tokens using equation 6.8.

$$\begin{aligned}\hat{N} &= 14 \log_2 14 + 10 \log_2 10 \\ &= 53.34 + 33.22 = 86.56\end{aligned}$$

Thus, even before constructing the sorting program, equation 6.8 predicts that there will be about 87 tokens (operators and operands) used in the completed program, which is within 10% of the program's actual length N of 91 tokens (refer Table 6.2).

There are several major flaws in the derivation of the length equation (equation 6.8). These flaws are discussed in [SHEN83]. Most of empirical support for software science is based on analysis of the relationship between estimated and actual lengths. Researchers frequently report correlation of 0.95 or higher between these quantities [FITZ78, CONT86].

The length equation can be viewed as a hypothesis whose credibility rests on several independent experiments, which have indicated that \hat{N} is a reasonable estimator of N. In the process of studying Halstead's length estimator the following alternate expressions have been published to estimate program length.

$$N_j = \log_2 (\eta_1!) + \log_2 (\eta_2!) \quad (6.9)$$

This equation was proposed by Jensen & Variavan [JENS85] and found to be a much better approximation for their data set. Another equation was derived by Mehndiratta & Grover [MEHN86], which is the slight modification of Halstead length estimator.

$$N_B = \eta_1 \log_2 \eta_2 + \eta_2 \log_2 \eta_1 \quad (6.10)$$

The Halstead estimator was further modified by Card & Agresti [CARD87] by substituting $\sqrt{\eta_1}$ for $\log_2 \eta_1$ and $\sqrt{\eta_2}$ for $\log_2 \eta_2$, thus

$$N_C = \eta_1 \sqrt{\eta_1} + \eta_2 \sqrt{\eta_2} \quad (6.11)$$

It is claimed that for small values of η , $\sqrt{\eta}$ and $\log_2 \eta$ behave similarly. Unfortunately, the accuracy of the estimator does not improve.

Although it is easy to construct a pathological program to make \hat{N} a poor predictor of N, there is overwhelming evidence using existing analysers to suggest the validity of the length equation in several languages. Shen et.al. [SHEN83] suggested that a misclassification of any token has virtually no effect on the final estimate.

Since $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \approx (\eta \log_2 \eta) / 2$ regardless of how the vocabulary of size η is divided into operators and operands. Hence

$$N_S = (\eta \log_2 \eta) / 2 \quad (6.12)$$

The definitions of unique operators, unique operands, total operators and total operands are not specifically delineated. Bulut [BULU73] has considered the counting methods used for FORTRAN, ALGOL & machine languages. Conte [CONE86] suggested counting rules for PASCAL and James Elshoff [JAME78] reported rules for PL/1 programs. Counting rules for C languages are suggested [YOGE95] and are given below:

Counting rules for C language

1. Comments are not considered.
2. The identifier and function declarations are not considered.

3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.
5. Local variables with the same name in different functions are counted as unique operands.
6. Function calls are considered as operators.
7. All looping statements e.g., do{...} while (), while () {...}, for () {...}, all control statements e.g., if () {...}, if () {...} else {...}, etc. are considered as operators.
8. In control construct switch () { case : ...}, switch as well as all the case statements are considered as operators.
9. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.
10. All the brackets, commas, and terminators are considered as operators.
11. GOTO is counted as an operator and the label is counted as an operand.
12. The unary and binary occurrence of “+” and “-” are dealt separately. Similarly “*” (multiplication operator), and “*” (de referencing operator), “&” (bitwise AND operator) and “&” (address operator) are dealt with separately.
13. In the array variables such as “array-name[index]” “array-name” and “index” are considered as operands and [] is considered as operator.
14. In the structure variables such as “struct-name, member-name” or “struct-name →member-name”, struct-name, member-name are taken as operands and ‘,’ ‘→’ are taken as operators. Same names of member elements in different structure variables are counted as unique operands.
15. All the hash directives are ignored.

The counting method presented here is fairly complete, however, “C” is so rich and complex that probability of additions and exceptions is always there.

6.2.2 Potential Volume

Many different but equivalent programs may implement an algorithm. Amongst all these programs, the one that has minimal size is said to have the potential volume V^* . Halstead argued that the minimal implementation of any algorithm was through a reference to a procedure that had been previously written. The implementation of this algorithm would then require nothing more than invoking the procedure and supplying the operands for its input and output parameters. It is shown that the potential volume of an algorithm implemented as a procedure call could be expressed as

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*) \quad (6.13)$$

The first term in the parentheses, 2, represents the two unique operators for the procedure call — the procedure name and a grouping symbol that separates the procedure name from its parameters. The second term, η_2^* represents the number of conceptually unique input and output parameters. η_2^* can probably be determined for small application programs, it is much more difficult to compute for large programs, such as compiler or an operating system. In these cases it is difficult to identify precisely the conceptually unique operands.

6.2.3 Estimated Program Level / Difficulty

Halstead offered an alternate formula that estimates the program level.

$$\hat{L} = 2 \eta_2 / (\eta_1 N_2) \quad (6.14)$$

Hence

$$\hat{D} = \frac{1}{\hat{L}} = \frac{\eta_1 N_2}{2\eta_2}$$

An intuitive argument for this formula is that programming difficulty increases, if additional operators are introduced (*i.e.*, if $\eta_1/2$ increases) and if an operand is used repetitively (*i.e.*, if N_2/η_2 increases). Every parameter in equation 6.14 may be obtained by counting the operators and operands in a finished computer program.

6.2.4 Effort and Time

Halstead hypothesized that the effort required to implement a program increases as the size of the program increases. It also takes more effort to implement a program at a lower level (higher difficulty) than another equivalent program at a higher level (lower difficulty). Thus, effort E measured in elementary mental discriminations is:

$$\begin{aligned} E &= V / \hat{L} = V * \hat{D} \\ &= (\eta_1 N_2 N \log_2 \eta) / 2\eta_2 \end{aligned} \quad (6.15)$$

A major claim for software science is its ability to relate its basic metrics to actual implementation. A psychologist, John Stroud, suggested that the human mind is capable of making a limited number of elementary discriminations per second [STRO67]. Stroud claimed that this number β (called the stroud number) ranges between 5 and 20. Since effort E uses elementary mental discriminations as its unit of measure, the programming time T of a program in seconds is simply

$$T = E / \beta \quad (6.16)$$

β is normally set to 18 since this seemed to give best results in Halstead's earliest experiments, which compared the predicted times with observed programming times, including the time for design, coding, and testing. Halstead claimed that this formula can be used to estimate programming time when a problem is solved by one proficient, concentrating programmer writing a single module program.

6.2.5 Language Level

There are now literally hundreds of programming languages. In some organizations, several languages may be used on a regular basis for software development. For example, in some large companies, software is being developed using FORTRAN, PASCAL, COBOL, and assembly language. There are proponents of each major language (including those of Ada, C, and Prolog) that argue that their favourite language is the best to use. These arguments suggest the need for a metric that expresses the power of a language [SHEN83].

Halstead hypothesized that, if the programming language is kept fixed, as V^* increases L decreases in such a way that the product $L \times V^*$, remains constant. Thus, this product, which he called the language level, λ can be used to characterize a programming language. Lower value of λ means that the language is closer to the machine.

$$\lambda = L \times V^* = L^2 V \quad (6.17)$$

Using this formula, Halstead and other researchers determined the language level for various languages as shown in Table 6.1 [HALS77, SHEN83, YOGE95].

Table 6.1: Language levels

Language	Language Level λ	Variance σ
PL/1	1.53	0.92
ALGOL	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42
PASCAL	2.54	—
APL	2.42	—
C	0.857	0.445

These averages, λ 's follow the intuitive rankings of most programmers for these languages, but they all have large variances.

The current state of software science seems to be still that of a evolving theory. There are those who question (with good reason in most cases) some of its underlying assumptions. However, there is large body of published data that suggests that software science metrics may be useful; especially η_1 and η_2 have been shown to strongly correlate to program size and error rates. Researchers are therefore continuing to find these metrics useful as a basis for size and effort models. Thus, Halstead work served to stimulate a great deal of interest in software metrics, as well as to contribute some basic metrics that survive till today.

Example 6.1

Consider the sorting program given in Fig. 4.2 of chapter 4 (software project planning). List out the operators and operands and also calculate the values of software science measures like η , N, V, E, λ etc.

Solution

The list of operators and operands is given in Table 6.2.

Table 6.2: Operators and operands of sorting program of Fig. 4.2 of chapter 4.

Operators	Occurrences	Operands	Occurrences
int	4	SORT	1
()	5	x	7
,	4	n	3
[]	7	i	8
if	2	j	7
<	2	save	3

(Contd.)...

;	11	im1	3
for	2	2	2
=	6	1	3
-	1	0	1
<=	2	—	—
++	2	—	—
return	2	—	—
{ }	3	—	—
$\eta_1 = 14$	$N_1 = 53$	$\eta_2 = 10$	$N_2 = 38$

Here $N_1 = 53$ and $N_2 = 38$. The program length $N = N_1 + N_2 = 91$

Vocabulary of the program $\eta = \eta_1 + \eta_2 = 14 + 10 = 24$

$$\begin{aligned} \text{Volume } V &= N \times \log_2 \eta \\ &= 91 \times \log_2 24 = 417 \text{ bits.} \end{aligned}$$

If a binary encoded scheme is used to represent each of the 24 items in the vocabulary, it would take 5 bits per item, since a 4 bit scheme leads to 16 unique codes (which is not enough), and a 5-bit scheme leads to 32 unique codes (which is more than sufficient). Each of the 91 tokens used in the program could be represented in order by a 5-bit code, leading to a string of $5 \times 91 = 455$ bits that would allow us to store the entire program in memory. Notice that size analysis is based on storing not a compiled version of the subroutine, but a binary translation of the original program. We could then say that this program occupies 455 bits of storage in its encoded form. However, also notice that a 5-bit scheme allows for 32 tokens instead of just 24 tokens, so instead of using the integer 5, volume uses the non-integer $\log_2 \eta = 4.58$ to arrive at a slightly smaller volume of 417.

The estimated program length \hat{N} of the program

$$\begin{aligned} &= 14 \log_2 14 + 10 \log_2 10 \\ &= 14 * 3.81 + 10 * 3.32 \\ &= 53.34 + 33.2 = 86.45 \end{aligned}$$

Conceptually unique input and output parameters are represented by η_2^*
 $\eta_2^* = 3$ {x : array holding the integer to be sorted. This is used both as input and output}.

{N : the size of the array to be sorted}.

The potential volume $V^* = 5 \log_2 5 = 11.6$

Since $L = V^* / V$

$$= \frac{11.6}{417} = 0.027$$

$$D = I / L$$

$$= \frac{1}{0.027} = 37.03$$

Estimated program level

$$\hat{L} = \frac{2}{\eta_1} \times \frac{\eta_2}{N_2} = \frac{2}{14} \times \frac{10}{38} = 0.038$$

which is not very close to the 0.027 level, we determined earlier using the conceptually unique operands.

We may use another formula

$$\hat{V}^* = V \times \hat{L} = 417 \times 0.038 = 15.67$$

The discrepancy between V^* and \hat{V}^* does not inspire confidence in the application of this portion of software science theory to more complicated programs.

$$\begin{aligned} E &= V/\hat{L} = \hat{D} \times V \\ &= 417 / 0.038 = 10973.68 \end{aligned}$$

Therefore, 10974 elementary mental discriminations are required to construct the program.

$$T = E / \beta = \frac{10974}{18} = 610 \text{ seconds} \approx 10 \text{ minutes}$$

This is probably a reasonable time to produce the program, which is very simple.

Table 6.3

```
#include < stdio.h >
#define MAXLINE 100
int getline(char line[], int max);
int strindex(char source[], char search for[]);
char pattern[ ]="ould";
int main()
{
    char line[MAXLINE];
    int found = 0;
    while(getline(line,MAXLINE)>0)
        if(strindex(line, pattern)>=0)
        {
            printf("%s",line);
            found++;
        }
    return found;
}
```

```

int getline(char s[], int lim)
{
    int c, i=0;
    while(--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++]=c;
    if(c=='\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

int strindex(char s[], char t[])
{
    int i, j, k;
    for(i=0; s[i] != '\0'; i++)
    {
        for(j=i, k=0; t[k] != '\0', s[j] == t[k]; j++, k++);
        if(k>0 && t[k] == '\0')
            return i;
    }
    return -1;
}

```

Example 6.2

Consider the program shown in Table 6.3. Calculate the various software science metrics.

Solution

List of operators and operands are given in Table 6.4.

Table 6.4

Operators	Occurrences	Operands	Occurrences
main ()	1	—	—
-	1	Extern variable pattern	1
for	2	main function line	3
==	3	found	2
!=	4	getline function s	3
getchar	1	lim	1

(Contd.)...

()	1	<i>c</i>	5
&&	3	<i>i</i>	4
--	1	Strindex function <i>s</i>	2
return	4	<i>t</i>	3
++	6	<i>i</i>	5
printf	1	<i>j</i>	3
>=	1	<i>k</i>	6
strindex	1	Numerical Operands 1	1
If	3	MAXLINE	1
>	3	0	8
getline	1	'\0'	4
while	2	'\n'	2
{ }	5	strings "ould"	1
=	10	—	—
[]	9	—	—
,	6	—	—
;	14	—	—
EOF	1	—	—
$\eta_1 = 24$	$N_1 = 84$	$\eta_2 = 18$	$N_2 = 55$

Program vocabulary

$$\eta = 42$$

Program length

$$N = N_1 + N_2$$

$$= 84 + 55 = 139$$

Estimated length

$$\hat{N} = 24 \log_2 24 + 18 \log_2 18 = 185.115$$

% error

$$= 24.91$$

Program volume

$$V = 749.605 \text{ bits}$$

Estimated program level

$$= \frac{2}{\eta_1} \times \frac{\eta_2}{N_2}$$

$$= \frac{2}{24} \times \frac{18}{55} = 0.02727$$

Minimal volume

$$V^* = 20.4417$$

Effort

$$= V/\hat{L}$$

$$= \frac{749.605}{0.02727}$$

 $= 27488.33$ elementary mental discriminations.

$$\begin{aligned}
 \text{Time } T &= E/\beta = \frac{27488.33}{18} \\
 &= 1527.1295 \text{ seconds} \\
 &= 25.452 \text{ minutes}
 \end{aligned}$$

6.3 DATA STRUCTURE METRICS

Essentially, the need for software development and other activities are to process data. Some data is input to a system, program, or module; some data may be used only internally; and some data is the output from a system, program, or module. A few examples of input, internal, and output data appear in Fig. 6.1.

<i>Program</i>	<i>Data Input</i>	<i>Internal Data</i>	<i>Data Output</i>
Payroll	Name / Social Security No./Pay Rate / Number of hours worked	Withholding rates Overtime factors Insurance premium Rates	Gross pay withholding Net pay Pay ledgers
Spreadsheet	Item Names / Item amounts/Relationships among items	Cell computations Sub-totals	Spreadsheet of items and totals
Software Planner	Program size / No. of software developers on team	Model parameters Constants Coefficients	Est. project effort Est. project duration

Fig. 6.1: Some examples of input, internal, and output data [CONT86].

Thus, an important set of metrics is that, which captures the amount of data input to, processed in and output from software. For example, assume that a problem can be solved in two ways, resulting in programs A and B. A has 25 input parameters, 35 internal data items, and 10 output parameters. B has 5 input parameters, 12 internal data items, and 4 output parameters. We can assume that A is probably more complicated, took more time to program, and has a greater probability of errors than B.

A count of the amount of data input to, processed in, and output from software is called a data structure metric. This section presents several data structure metrics. Some concentrate on variables (and even constants) within each module and ignore the input/output dependencies. Others concern themselves primarily with the input/output situation. There is no general agreement on how the line of code measure (the classical and best-known software metric) is to be counted. Thus, it is not surprising that there are various methods for measuring data structures as well. In the following subsections, we will discuss the metrics proposed to measure the amount of data, the usage of data within modules, and the degree to which data is shared among modules [CONT86].

6.3.1 The Amount of Data

Most compilers and assemblers have an option to generate a cross-reference list, indicating the line where a certain variable is declared and the line or lines where it is referenced. Such a list is useful in debugging and maintenance, and can help determine the amount of data in the program. Consider a simple program which appears in Fig. 6.2. It inputs work hours and pay rates, and computes gross pay, taxes, and net pay. The C compiler produces a cross-reference listing for this program, which appears in Fig. 6.3.

One method for determining the amount of data is to count the number of entries in the cross-reference list. Be careful to exclude from the count those variables that are defined but never used. The definitions of these variables may be made for future reference, but they do not affect the operational characteristics of the program or, more importantly, the difficulty of development, and should not be counted. Such a count of variables will be referred to as VARS. Thus, for the program *payday* appearing in the Fig. 6.2 VARS = 7. For the sample program *SORT* in Fig. 4.2 VARS = 6 (from a cross-reference listing that identified X, N, I, J, SAVE and IM1 as the variables in the program). The count of variables VARS depends on the following definition:

A variable is a string of alphanumeric characters that is defined by a developer and that is used to represent some value during either compilation or execution.

1.	#include < stdio. h >
2.	struct check
3.	{
4.	float gross, tax, net;
5.	} pay;
6.	float hours, rate;
7.	void main ()
8.	{
9.	while (! feof (stdin))
10.	{
11.	scanf("%f %f", & hours, & rate);
12.	pay. gross = hours * rate;
13.	pay. tax = 0.25 * pay. gross;
14.	pay. net = pay. gross - pay. tax;
15.	printf("%f %f %f/n", pay. gross, pay. tax, pay. net);
16.	}
17.	}

Fig. 6.2: Payday program

check	2				
gross	4	12	13	14	15
hours	6	11	12		
net	4	14	15		
pay	5	12	13	13	14
	14	14	15	15	15
rate	6	11	12		
tax	4	13	14	15	

Fig. 6.3: A cross reference of program payday

Although a simple way to obtain VARS is from a cross-reference list, it can also be generated using a software analyzer that counts the individual tokens [CONT86].

While it may sound simple to determine the value of VARS — certainly, counting the number of variables in a program seems straightforward—there are some items in the cross-reference listing of the program in Fig. 6.2 that have been deleted. These items are listed in Fig. 6.4. The items feof stdin are related to I/O. The cross-referencing software called the name of the program payday a variable. However, because none of these are variables in the sense of variables that we create to produce a program, we deleted them from Fig. 6.3, but it should be clear that this “algorithmic” metric VARS is, in fact, a little subjective. In determining this metric, as with all other software metrics including lines of code, we attempt to establish guidelines that eliminate as much subjectivity as possible. But, the reader is well advised to realize that total objectivity is impossible.

feof	9
stdin	10

Fig. 6.4: Some items not counted as VARS

Among all the variable names in line 13 of Fig. 6.2 is the constant 0.25. This program assumes that all pay will be taxed at a 25% rate. Also, in line 05 of Fig. 4.2 of chapter 4, the constant 2 is used to avoid sorting arrays with less than two elements. None of these constants 0.25 or 2 are counted as VARS, and yet they play special purposes in the programs. Furthermore, mathematical constants such as τ and e are important for programs involving trigonometric or logarithmic applications. Even array references with an explicit index, such as A[11], may indicate some special meaning for that particular location.

Halstead [HALS77] introduced a metric that he referred to as η_2 to be a count of the operands in a program—including all variables, constants, and labels. Thus,

$$\eta_2 = \text{VARS} + \text{unique constants} + \text{labels}.$$

The sample SORT program in Fig. 4.2 which is analysed in Table 6.2 has 6 variables (X, N, I, J SAVE, IM1), 3 constants (1, 2, 0) and 01 labels (SORT) so that $\eta_2 = 10$. The name of the subroutine SORT is treated as a label since it is the label that will be used by any other

program or sub-program that wants to access SORT. The program *payday* in Fig. 6.2 has 7 variables (check, gross, hours, net, pay, rate, tax), 1 constant 0.25, and no label. Thus, its η_2 is 8 note that η_2 is the count of the number of unique operands. Thus, this metric fails to capture an important feature of the “amount of data”—namely, the total operand usage. For example, given the 8 operands in Fig. 6.2, it is possible to construct the program shown or to construct a much larger and more complicated program in order to measure the quantity of usage of the operands. Halstead further defined the metric total occurrence of operands, and named it N_2 . Figure 6.5 repeats Fig. 6.2 and encloses each operand occurrences in brackets [CONT86].

The program *payday* uses the 8 operands 30 times: some are used several times (like *pay*) and some are used sparingly (0.25 is used only once). Thus, $N_2 = 30$ for this program.

The metrics VARS, η_2 , and N_2 are the most popular data structure measures. They seem to be robust, slight variations in algorithm computation schemes for computing them do not seem to affect inordinately other measures based upon them.

1	# include < stdio. h >
2	struct [check]
3	{
4	float [gross], [tax], [net];
5	} [pay];
6	float [hours], [rate];
7	void main ()
8	{
9	while (! feof (stdin))
10	{
11	scanf("% f % f", & [hour], & [rate]);
12	[pay] . [gross] = [hours] * [rate];
13	[pay] . [tax] = 0.25 * [pay] . [gross];
14	[pay] . [net] = [pay] . [gross] - [pay] . [tax];
15	printf("% f % f % f/n", [pay] . [gross] [pay] . [tax], [pay] . [net]);
16	}
17	}

Fig. 6.5: Program *payday* with operands in brackets

6.3.2 The Usage of Data within a Module

In Fig. 6.6 the program “bubble” inputs two related integer arrays (*a* and *b*) of the same size up to 100 elements each. It uses a bubble sort on the *a*-array, interchanges the *b*-array values to keep them with the accompanying *a*-array values, and outputs the results. Prior to Fig. 6.6, all of our examples have illustrated small, single-module programs or subroutines. Fig. 6.6 contains a main program in lines 11–37 and a sub-program procedure *swap* in lines 3–9.

Several metrics may be computed for individual modules. In order to characterize the intra-module data usage, we may use the metrics live variables and variable spans that are discussed below.

Live Variables: While constructing program “bubble”, the developer created a variable “last”. Analyze Fig. 6.6 carefully to see that all array elements beyond the “last” one are sorted. While the program is running, if size = 25 and last = 14, then all items a[15]–a[25] and b[15]–b[25] are in order even though the first 14 elements of each array are not yet sorted. A beginning value for “last” is established in line 17, decremented in line 22, and used in the logical expression in line 24.

There are only three statements in this program in which “last” appears, excluding the declaration in line 13. Does this mean that we do not need to be concerned with “last” while constructing the statements other than 17, 22, and 24? Certainly not. Between statements 17 and 24, it is important to keep in mind what “last” is doing. For example statements 18-19 are used to set up a potentially never-ending loop. However, even though these statements never mention “last”, the developer realized that each time on a-value “bubbles down” to its appropriate position. “Last” will be decremented by one. Eventually on some cycle through the a-values none will be swapped and the loop beginning in statement 19 will be exited. As we will show later, “last” has life span that begins at statement 17 and extends through statement 24.

Thus, a developer must constantly be aware of the status of a number of data items during the development process. A reasonable hypothesis is: more the data items that a developer must keep track of when constructing statements, the more difficult it is to construct. Thus, our interest lies in the size of the set of those data items called live variables (LV) for each statement in the program.

As suggested earlier, the set of live variables for a particular statement is not limited to the number of variables referenced in that statement. For example, the statement being considered may be just one of the several that set up the parameters for a complex procedure. The developer must be aware of entire list of parameters to know that they are being set up in an orderly fashion, so that any statement in the group disturbs no variables later. Therefore, there are several possible definitions of a live variable [DUNS579].

1. A variable is live from the beginning of a procedure to the end of the procedure.
2. A variable is live at a particular statement only if it is referenced a certain number of statements before or after that statement.
3. A variable is live from its first to its last references within a procedure.

The first definition, while computationally simple, does not correspond to the idea of the live variable. According to this definition, both the variable “last” with a 8-statement life span (lines 17–24) and the variable “size” with a 22 statement life span (lines 14–35) can be considered alive throughout the procedure. The second definition might work, but there is no agreement on what a “certain number of statements” should be and no successful use has been reported.

The third definition meets the spirit of the live variable idea and is easy to compute algorithmically. In fact, a computer program (a software analyzer) can produce live variable counts for all statements in a program or procedure.

1	#include < stdio. h >
2	
3	void swap (int x [], int K)
4	{
5	int t;
6	t = x[K];
7	x[K] = x[K + 1];
8	x[K + 1] = t;
9	}
10	
11	void main ()
12	{
13	int i, j, last, size, continue, a[100], b[100];
14	scanf("% d", & size);
15	for (j = 1; j < = size; j + +)
16	scanf("%d %d", & a[j], & b[j]);
17	last = size;
18	continue = 1;
19	while(continue)
20	{
21	continue = 0;
22	last = last-1;
23	i = 1;
24	while (i < = last)
25	{
26	if (a[i] > a[i + 1])
27	{
28	continue = 1;
29	swap (a, i);
30	swap (b, i);
31	}
32	i = i + 1;
33	}
34	}
35	for (j = 1; j < = size; j + +)
36	printf("%d %d\n", a[j], b[j]);
37	}

Fig. 6.6: Bubble sort program

It is thus possible to define the average number of live variables (\overline{LV}), which is the sum of the count of live variables divided by the count of executable statements in a procedure. This is a complexity measure for data usage in a procedure or program. The live variables in the program in Fig. 6.6 appear in Fig. 6.7 the average live variables for this program is

$$\frac{124}{34} = 3.647.$$

<i>Line</i>	<i>Live Variables</i>	<i>Count</i>
4	—	0
5	—	0
6	<i>t,x,k</i>	3
7	<i>t,x,k</i>	3
8	<i>t,x,k</i>	3
9	—	0
10	—	0
11	—	0
12	—	0
13	—	0
14	<i>size</i>	1
15	<i>size,j</i>	2
16	<i>size,j,a,b</i>	4
17	<i>size,j,a,b,last</i>	5
18	<i>size,j,a,b,last,continue</i>	6
19	<i>size,j,a,b,last,continue</i>	6
20	<i>size,j,a,b,last,continue</i>	6
21	<i>size,j,a,b,last,continue</i>	6
22	<i>size,j,a,b,last,continue</i>	6
23	<i>size,j,a,b,last,continue,i</i>	7
24	<i>size,j,a,b,last,continue,i</i>	7
25	<i>size,j,a,b,continue,i</i>	6
26	<i>size,j,a,b,continue,i</i>	6
27	<i>size,j,a,b,continue,i</i>	6
28	<i>size,j,a,b,continue,i</i>	6
29	<i>size,j,a,b,i</i>	5
30	<i>size,j,a,b,i</i>	5

(Contd.)...

31	size, j, a, b, i	5
32	size, j, a, b, i	5
33	size, j, a, b	4
34	size, j, a, b	4
35	size, j, a, b	4
36	j, a, b	3
37	—	0

Fig. 6.7: Live variables for the program in Fig. 6.6.

As shown live variables depend on the order of statements in the source program, rather than the dynamic execution-time order in which they are encountered. A metric based on runtime order would be more precisely related to the life of the variable, but would be much more difficult to define algorithmically (especially in a non-structured programming language).

Variable spans: Two variables can be alive for the same number of statements, but their use in a program can be markedly different. For example, Fig. 6.8 lists all of the statements in a C program that refer to the variables “a” and “b”. Both variables are alive for the same 40 statements (21–60), but “a” is referred to three times while “b” is mentioned only once. A metric that captures some of the essence of how often a variable is used in a program is called the span (SP). This metric is the number of statements between two successive references of the same variable [ELSH76]. The span is related to the third definition of live variables. For a program that references a variable in n statements, there are n-1 spans for that variable. Thus, in Fig. 6.8 “a” has 4 spans and “b” has only 2. Intuitively this tells us that ‘a’ is being used more than ‘b’.

...	
21	scanf(" %d %d," & a, & b);
...	
32	x = a;
...	
45	y = a - b;
...	
53	z = a;
...	
60	printf(" %d %d," a, b);
...	

Fig. 6.8: Statements in ac program referring to variables a and b.

Furthermore, the size of a span indicates the number of statements that pass between successive uses of a variable. A large span can require the developer to remember during the construction process a variable that was last used in the program. In Fig. 6.8 “a” has 4 spans of 10, 12, 7, and 6, statements, while for “b” has 2 spans of 23 and 14 statements. It is simple to extend this metric to “average span size”, (\overline{SP}) in which case “a” has an average span size of 8.75 and ‘b’ has an average span size of 18.5.

Making program-wide metrics from intra-module metrics: Each of the metric discussed in this section is intended to be used within a module, as indicated. But it is possible to extend each one into an inter-module metric. For example if we want to characterize the average number of live variables for a program having modules, we can use this equation.

$$\overline{LV}_{\text{program}} = \frac{\sum_{i=1}^m \overline{LV}_i}{m}$$

where \overline{LV}_i is the average live variable metric computed from the i th module.

Furthermore, the average span size (\overline{SP}) for a program of n spans could be computed by using the equation.

$$\overline{SP}_{\text{program}} = \frac{\sum_{i=1}^n \overline{SP}_i}{n}$$

6.3.3 Program Weakness

A program consists of modules. Using the average number of live variables (\overline{LV}) and average life of variables (γ), the module weakness has been defined as [YOGE98]:

$$WM = \overline{LV} * \gamma$$

Average number of live variables and average life of variables can be found using some automated tools. Even most compilers and assemblers have an option to generate a cross-reference list, indicating the line number where a certain variable is declared and the line or lines where it is referenced. Such a list may be used to compute the value of LV and γ . Using these two values, weakness of a module can be computed. The weakness of the module can be used to estimate the testability and maintainability. If weakness of a module is more, testability will be better and vice versa. Weakness will also have effect on maintainability. A weaker module will be more difficult to maintain.

As we all know a program is normally a combination of various modules, hence program weakness can be a useful measure and is defined as:

$$WP = \frac{\left(\sum_{i=1}^m WM_i \right)}{m}$$

where, WM_i : weakness of i th module.

WP : weakness of the program

m : number of modules in the program.

Example 6.3

Consider a program for sorting and searching. The program sorts an array using selection sort and then search for an element in the sorted array. The program is given in Fig. 6.8. Generate cross-reference list for the program and also calculate $\bar{L}V$, γ , and WM for the program.

Solution

The given program is of 66 lines and has 11 variables. The variables are a , i , j , item, min, temp, low, high, mid, loc and option.

```
1  **** **** **** **** **** **** **** **** **** **** **** **** **** **** **** **** /  
2  ***** PROGRAM TO SORT AN ARRAY USING SELECTION SORT & THEN SEARCH  
***** /  
3  ***** FOR AN ELEMENT IN THE SORTED ARRAY ***** /  
4  **** **** **** **** **** **** **** **** **** **** **** **** **** **** /  
5  
6  #include <stdio.h>  
7  #define MAX 10  
8  
9  main ()  
10 {  
11     int a[MAX];  
12     int i,j,item,min,temp;  
13     int low=0,high,mid,loc;  
14     char option;  
15  
16     for (i=0;i<MAX;i++)  
17     {  
18         printf("Enter a[%d]:",i);  
19         scanf("%d",&a[i]);  
20     }  
21     /* selection sort */  
22     for (i=0;i<(MAX-1);i++)  
23     {  
24         min=i;  
25         for (j=i+1; j<MAX; j++)  
26         {  
27             if (a[min]>a[j])  
28             {  
29                 temp=a[min];  
30                 a[min]=a[j];  
31                 a[j]=temp;
```

(Contd.)...

```
32         }
33     }
34 }
35 printf("\n The Sorted Array:\n");
36 for (i=0; i<MAX;i++)
37     printf("\n a[%d]=%d",i,a[i]);
38 printf("\n Do you want to search any element in the array
39 (Y/N) :");
40 fflush(stdin);
41 scanf("%c",&option);
42 if (toupper(option)=='Y')
43 {
44     printf("\n Enter the item to be searched :");
45     scanf("%d", &item);
46     high=MAX;
47     mid=(int)(low+high)/2;
48     while ((low<=high)&&(item!=a[mid]))
49     {
50         if (item>a[mid])
51             low=mid+1;
52         else high=mid-1;
53         mid=(int) (low+high)/2;
54     }
55     if(low>high)
56     {
57         loc=0;
58         printf("\n No such item is present in the array\n");
59     }
60     if (item==a[mid])
61     {
62         loc=mid;
63         printf("\n The item %d is present at location %d in the sorted
64         array\n",item,loc);
65     }
66 }
```

Fig. 6.8: Sorting & searching program

Cross-Reference list of the program is given below:

<i>a</i>	11	18	19	27	27	29	30	30	31	37	47	49	59		
<i>i</i>	12	16	16	16	18	19	22	22	22	24	36	36	36	37	37
<i>j</i>	12	25	25	25	27	30	31								
item	12	44	47	49	59	62									
min	12	24	27	29	30										
temp	12	29	31												
low	13	46	47	50	52	54									
high	13	45	46	47	51	52	54								
mid	13	46	47	49	50	51	52	59	61						
loc	13	56	61	62											
option	14	40	41												

Live Variables per line are calculated as:

Line Number	Live Variables on the line	Count
13	low	1
14	low	1
15	low	1
16	low, <i>i</i>	2
17	low, <i>i</i>	2
18	low, <i>i</i> , <i>a</i>	3
19	low, <i>i</i> , <i>a</i>	3
20	low, <i>i</i> , <i>a</i>	3
22	low, <i>i</i> , <i>a</i>	3
23	low, <i>i</i> , <i>a</i>	3
24	low, <i>i</i> , <i>a</i> , min	4
25	low, <i>i</i> , <i>a</i> , min, <i>j</i>	5
26	low, <i>i</i> , <i>a</i> , min, <i>j</i>	5
27	low, <i>i</i> , <i>a</i> , min, <i>j</i>	5
28	low, <i>i</i> , <i>a</i> , min, <i>j</i>	5
29	low, <i>i</i> , <i>a</i> , min, <i>j</i> , temp	6
30	low, <i>i</i> , <i>a</i> , min, <i>j</i> , temp	6
31	low, <i>i</i> , <i>a</i> , <i>j</i> , temp	5
32	low, <i>i</i> , <i>a</i> ,	3
33	low, <i>i</i> , <i>a</i>	3
34	low, <i>i</i> , <i>a</i>	3
35	low, <i>i</i> , <i>a</i>	3
36	low, <i>i</i> , <i>a</i>	3

(Contd.)...

37	low, <i>i</i> , <i>a</i>	3
38	low, <i>a</i>	2
39	low, <i>a</i>	2
40	low, <i>a</i> , option	3
41	low, <i>a</i> , option	3
42	low, <i>a</i>	2
43	low, <i>a</i>	2
44	low, <i>a</i> , item	3
45	low, <i>a</i> , item, high	4
46	low, <i>a</i> , item, high, mid	5
47	low, <i>a</i> , item, high, mid	5
48	low, <i>a</i> , item, high, mid	5
49	low, <i>a</i> , item, high, mid	5
50	low, <i>a</i> , item, high, mid	5
51	low, <i>a</i> , item, high, mid	5
52	low, <i>a</i> , item, high, mid	5
53	low, <i>a</i> , item, high, mid	5
54	low, <i>a</i> , item, high, mid	5
55	<i>a</i> , item, mid	3
56	<i>a</i> , item, mid, loc	4
57	<i>a</i> , item, mid, loc	4
58	<i>a</i> , item, mid, loc	4
59	<i>a</i> , item, mid loc	4
60	item, mid, loc	3
61	item, mid, loc	3
62	item, loc	2
63		0
64		0
65		0
66		0
	Total	174

Thus Avg. number of Live Variables (\overline{LV}) = $\frac{\text{Sum of count of live variables}}{\text{Count of executable statements}}$

$$\overline{LV} = \frac{174}{53} = 3.28$$

$$\Rightarrow \overline{LV} = 3.28$$

$$\gamma = \frac{\text{Sum of count of live variables}}{\text{Total No. of variables}}$$

$$\Rightarrow \gamma = \frac{174}{11} = 15.8$$

\Rightarrow γ (i.e. Avg. life of variables) = 15.8

Module Weakness

$$WM = \overline{LV} \times \gamma$$

where WM is the module weakness

\overline{LV} is the Avg. no. of live variables

& γ is the Avg. life of variables

$$\Rightarrow WM = 3.28 \times 15.8 = 51.8$$

$$WM = 51.8$$

Example 6.4

Consider a program given in Fig. 6.9 that draws a circle using midpoint algorithm. Generate cross reference list of variables and also calculate average number of live variables (\overline{LV}), average life of variables (γ) and program weakness (WM).

Solution

There are 9 variables declared in the program. The variables are rad, p_0 , p_1 , x , y , x_c , y_c , d and m .

```
\\"To scan convert a circle using midPoint Algorithm
1. #include <iostream.h>
2. #include <conio.h>
3. #include <graphics.h>
4. void circle (int, int, int, int);
5. void main ()
6. {
7.     int rad;
8.     int p0, p1;
9.     int x, y;
10.    int xc, yc;
11.    int d, m;
12.    d = DETECT;
13.    initgraph (&d, &m, " ");
14.    setbkcolor(BLACK);
15.    clrscr ();
16.    cout <<" enter the radius of circle:" ;
17.    cin >> rad;
18.    cout <<"Enter the value of center co-ordinates:" ;
```

(Contd.)...

```
19.    cin >> xc >> yc;
20.    x = 0;
21.    y = rad;
22.    circle(xc, yc, x, y);
23.    p0 = 1 - rad;
24.    while (x < y)
25.    {
26.        if(p0 <0)
27.        {
28.            x++;
29.            p0 = p0 + 2*(x + 1) + 1;
30.            circle (x, y, xc, yc);
31.        }
32.        else
33.        {
34.            x++;
35.            y--;
36.            p0 = p0 + 2* (x - y) + 1;
37.            circle(x, y, xc, yc);
38.        }
39.    }
40.    getch();
41. }
42. void circle(int x, int y, int xc, int yc)
43. {
44.     putpixel(xc + x, yc + y, 4);
45.     putpixel(xc - x, yc + y, 4);
46.     putpixel(xc + x, yc - y, 4);
47.     putpixel(xc - x, yc - y, 4);
48.     putpixel(xc + y, yc + x, 4);
49.     putpixel(xc - y, yc + x, 4);
50.     putpixel(xc + y, yc - x, 4);
51.     putpixel(xc - y, yc - x, 4);
52. }
```

Fig. 6.9: Program for drawing a circle using midpoint algorithm

The cross reference list is given below:

Variable	Reference a Line Number
rad	7, 17, 21, 23,
p_0	8, 23, 26, 29, 29, 36, 36,
p_1	8
x	9, 20, 22, 24, 28, 29, 30, 34, 36, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
y	9, 21, 22, 24, 30, 35, 36, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
x_c	10, 19, 22, 30, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
y_c	10, 19, 22, 30, 37, 42, 44, 45, 46, 47, 48, 49, 50, 51
d	11, 12, 13
m	11, 13

Live Variables per line are given below:

Line Number	Live Variables	Count
12	d	1
13	d, m	2
14	-	0
15	-	0
16	-	0
17	rad	1
18	rad	1
19	rad, x_c, y_c	3
20	rad, x_c, y_c, x	4
21	rad, x_c, y_c, x, y	5
22	rad, x_c, y_c, x, y	5
23	rad, x_c, y_c, x, y, p_0	6
24	x_c, y_c, x, y, p_0	5
25	x_c, y_c, x, y, p_0	5
26	x_c, y_c, x, y, p_0	5
27	x_c, y_c, x, y, p_0	5
28	x_c, y_c, x, y, p_0	5
29	x_c, y_c, x, y, p_0	5
30	x_c, y_c, x, y, p_0	5
31	x_c, y_c, x, y, p_0	5
32	x_c, y_c, x, y, p_0	5
33	x_c, y_c, x, y, p_0	5
34	x_c, y_c, x, y, p_0	5
35	x_c, y_c, x, y, p_0	5
36	x_c, y_c, x, y, p_0	5

(Contd.)...

37	x_c, y_c, x, y	4
38	x_c, y_c, x, y	4
39	x_c, y_c, x, y	4
40	x_c, y_c, x, y	4
41	x_c, y_c, x, y	4
42	x_c, y_c, x, y	4
43	x_c, y_c, x, y	4
44	x_c, y_c, x, y	4
45	x_c, y_c, x, y	4
46	x_c, y_c, x, y	4
47	x_c, y_c, x, y	4
48	x_c, y_c, x, y	4
49	x_c, y_c, x, y	4
50	x_c, y_c, x, y	4
51	x_c, y_c, x, y	4
52		0
	Total	153

\overline{LV} = Average number of live variables

$$= \frac{\text{Sum of count of live variables}}{\text{Count of executable statements}}$$

$$= \frac{153}{41} = 3.73$$

γ = Average life of variables

$$= \frac{\text{Sum of count of live variables}}{\text{Number of unique variables}}$$

$$= \frac{153}{9} = 17$$

$$\begin{aligned} \text{Program weakness} &= \overline{LV} \times \gamma \\ &= 3.73 \times 17 = 63.41 \end{aligned}$$

6.3.4 The Sharing of Data Among Modules

As discussed earlier, a program normally contains several modules and share coupling among modules. However, it may be desirable to know the amount of data being shared among the modules.

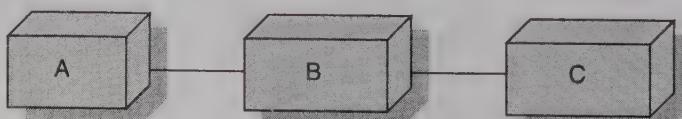


Fig. 6.10: Three modules from an imaginary program

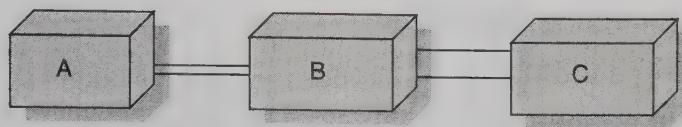


Fig. 6.11: "Pipes" of data shared among the modules

For example, Fig. 6.10 shows a schematic of a program consisting of the three subprograms A, B, and C. If we include information that represents the amount of data "passed" among the subprograms, we could envision pipes between the subprograms. The diameter of each pipe could represent the quantity or volume, of data sent from one subprogram to be used in the other. Fig. 6.11 shows the same three subprograms with pipes representing data shared between A and B and between B and C. Note that the A-B shared data is implicitly less than the B-C shared data. Fig. 6.12 shows a pictorial representation of the data shared between the main program of bubble and procedure swap both from Fig. 6.6 in bubble, the main program invokes the procedure swap in order to get it to swap the i th and $(i + 1)$ th members of the a-vector and the b-vector.

Here, we will introduce metrics that can be used for measuring this concept of sharing of data between modules. Keep in mind that the "bigger the pipe" in between any two modules, the more complex is their relationship. In theory, every module in a program is related to every other module.

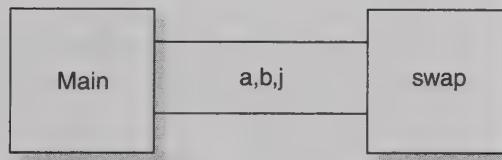


Fig. 6.12: The data shared in program bubble

If this were not true, the modules would not have been put into the same program. But, in practice, some modules may share no data directly with another, as shown by modules A and C in Fig. 6.11. The data structure employed in A should have little effect on module C, while the data structure employed in B is obviously important to C since they have such a large pipe between them.

The relationship between modules is simple if two variables are passed between them. However, a module with long list of parameters of different types, and some "global" data structure, which is shared, with several modules should be more difficult to construct or comprehend.

We assume that a global variable is one that is available to any and all modules in a program. Most programming languages allow the declaration of variables that can be accessed anywhere in the program. Contrast a global variable with a local variable, which is declared in a specific module, and whose name is unknown outside that module. A local variable is available to another module only when specifically mentioned in a parameter list passed to that module (in a procedural language), or when referred to in a module completely contained in the one where the local variable is declared (in a block-structured language). Global variables are not so limited, and are known and usable everywhere in the program.

6.4 INFORMATION FLOW METRICS

The other set of metrics we would like to consider are generally known as 'Information Flow' (IF) metrics. The basis of IF metrics is found upon the following premise. All but the simplest systems consist of components, and it is the work that these components do and how they are fitted together that influences the complexity of a system. If a component has to do numerous discrete tasks, it is said to lack 'cohesion'. If it passes information to, and/or accepts information from many other components within the system, it is said to be highly 'coupled'. Systems theory tells us that components that are highly coupled and that lack cohesion tend to be less reliable and less maintainable than those that are loosely coupled and that are cohesive. The following are the working definitions of the terms used above:

- Component : Any element identified by decomposing a (software) system into its constituent parts.
- Cohesion : The degree to which a component performs a single function.
- Coupling : The term used to describe the degree of linkage between one component to others in the same system.

This systems-view map to software systems is extremely easy to understand as most engineers today use, or are at least familiar with, top down design techniques that produce a hierarchical view of system components. Even the more modern 'middle out' design approaches produce this structured type of deliverable, and here again IF metrics can be used.

Information Flow metrics model the degree of cohesion and coupling for a particular system component. How that model is constructed can justifiably range from the simple to the complex. We intend to start with the most simple representation of IF metrics to illustrate the basic concepts, how to derive information using the metrics and how to use that information.

In terms of applying IF metrics to software systems, the pioneering work was done by Henry and Kafura [HENR81]. They looked at the UNIX operating system, and found a strong association between the IF metrics and the level of maintainability ascribed to components by developers. Other individuals who tried to apply these principles found difficulties in using the Henry and Kafura approach. Further work was done in the UK by Professor Darrell Ince and Martin Shepperd [INCE89], among others, which resulted in a more practical IF model. This work was complimented by Barbara Kitchenham [KITC90], who addressed the same problem, and who also presented a clear approach to the question of interpretation.

6.4.1 The Basic Information Flow Model

Information Flow metrics are applied to the Components of a system design. Fig. 6.13 shows a fragment of such a design, and for component 'A' we can define three measures, but remember that these are the simplest models of IF.

1. 'FAN IN' is simply a count of the number of other Components that can call, or pass control, to Component A.
2. 'FANOUT' is the number of Components that are called by Component A.
3. This is derived from the first two by using the following formula. We will call this measure the INFORMATION FLOW index of Component A, abbreviated as IF(A).

$$\text{IF}(A) = [\text{FAN IN}(A) \times \text{FAN OUT}(A)]^2$$

The formula includes a power component to 'model the non-linear nature of complexity' as most texts of IF metrics describe it. (The assumption is that if something is more complex than something else, then resultant is much more complex). Given that assumption, we could raise to a power three or four or whatever we want but, on the principle that the simpler the model the better, then two is a good enough choice. In our view, raising to two makes it easier, as will be seen, to pick out the potential bad guys [GOOD93].

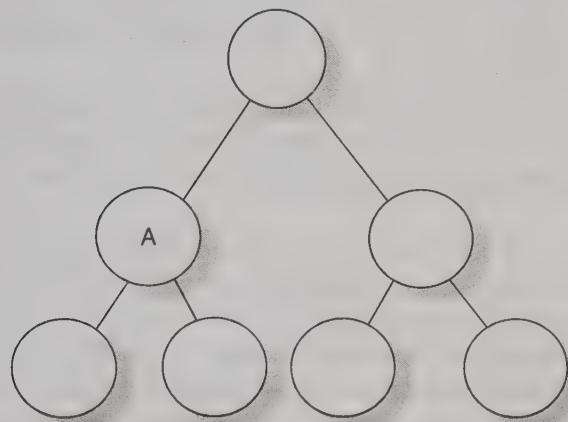


Fig. 6.13: Aspects of complexity

Given functional decomposition it will be seen that there is one additional attribute possessed by each Component, namely its level in the decomposition. The following is a step-by-step guide to deriving these most simple of IF metrics.

1. Note the level of each Component in the system design.
2. For each Component, count the number of calls to that Component — this is the FAN IN of that Component. Some organizations allow more than one Component at the highest level in the design, so for Components at the highest level which should have a FAN IN of zero, assign a FAN IN of one. Also note that a simple model of FAN IN can penalize reused Components.
3. For each Component, count the number of calls from that Component. For Components that call no other, assign a FAN OUT value of one.
4. Calculate the IF value for each Component using the above formula.
5. Sum the IF value for all Components within each level which is called as the LEVEL SUM.
6. Sum the IF values for the total system design which is called the SYSTEM SUM.
7. For each level, rank the Components in that level according to FAN IN, FAN OUT and IF values. Three histograms or line plots should be prepared for each level.
8. Plot the LEVEL SUM values for each level using a histogram or line plot.

This may sound like a great deal of work, but for most commercial systems, provided the documentation exists, this data can be derived and the analysis done within one engineering day. If the systems are larger, then it will obviously take longer, but remember that once done, it is very easy to keep up-to-date. Depending upon the environment it may even be possible to automate the calculations.

Having got the information, we now need to utilize it. It must be realized that, for IF metrics, there are no absolute values of good or bad. Information Flow metrics are relative indicators. This means that the value for one system may be higher than the other system, but this does not mean that one system is worse. Nor does a high metric value guarantee that a Component will be unreliable and un-maintainable. It is only that it will probably be less reliable and maintainable than its fellows.

The rub is that, in most systems, less reliable and less maintainable means that it is potentially going to cost significant amounts of money to fix and enhance. Potentially, it could even be a nightmare component.

A nightmare component is the one that the system administrator has nightmares about, because he or she knows that if anyone touches that component, the whole system is going to crash, and it will take weeks to fix because designers have already left and no one is available to guide.

So the strength of IF metrics is not in the numbers themselves, but in how the information is used. As a guide, 25 per cent of components with the highest scores for FAN IN, FAN OUT and IF values should be investigated. Now in practice it may well be that a certain number of components stick out like a sore thumb, especially on the IF values. If this group is more or less than the 25 per cent guide, then do not worry about it, concentrate on those that seem to be odd according to the metric values rather than following any 25 per cent rule slavishly.

High FAN IN values indicate components that lack cohesion. It may well be that the functions have not been broken out to a great enough degree. Basically, these Components are often called because they are doing more than one job.

High levels of FAN OUT also indicate a lack of cohesion or missed levels of abstraction. Here design was stopped before it was finished, and this is reflected in the high number of calls from the component. Generally speaking, FAN OUT appears to be a better indicator of problem components than FAN IN, but it is early days yet and we would not wish to discount FAN IN.

High IF values indicate highly coupled components. These Components need to be looked at in terms of FAN IN and FAN OUT to see how to reduce the complexity level. Sometimes a 'traffic centre' may be hit. This is a component where, for whatever reasons, there is a high IF value, but things cannot be improved. Switching components often exhibit this. Here there is a potential problem area which, it is also a large component, may be very error prone. If the complexity cannot be reduced, then at least make sure that component is thoroughly tested.

Looking at the LEVEL SUM plot of values, we should see a fairly smooth curve showing controlled growth in IF across the levels. Sudden increases in these values across levels can indicate a missed level of abstraction within the general design. For systems where the design has less than ten levels, then a simple count of components at each level seems to work equally well.

The final item of information flow metrics is the SYSTEM SUM value. This gives an overall complexity rating for the design in terms of IF metrics. Most presentations on this topic will say that this number can be used to assess alternative design proposals.

6.4.2 A More Sophisticated Information Flow Model

We have looked at the most simple form of IF metrics, but the original proposals put forward by Henry and Kafura [HENR81] were more sophisticated than the control flow-based variant discussed above. As mentioned earlier, Ince and Shepperd [INCE89] and Kitchenham [KITC90] have done a great deal of work to help in the practical application of Henry and Kafura's pioneering proposals, and it is a distillation of that work, that has been summarized by Goodman [GOOD93] into the more sophisticated IF model. It should, however, be realized that this is a model, and it will need to be tailored to one's organization's design mechanisms before it is used. Such a tailoring process should not take more than two days for counting rule derivation and documentation of these rules, provided a well-defined design notation is used together with a competent engineer who knows that notation.

The only difference between the simple and the sophisticated IF models lies in the definition of FAN IN and FAN OUT.

For a component A let:

a = the number of components that call A.

b = the number of parameters passed to A from components higher in the hierarchy;

c = the number of parameters passed to A from components lower in the hierarchy;

d = the number of data elements read by component A.

Then:

$$\text{FAN IN}(A) = a + b + c + d$$

Also let:

e = the number of components called by A;

f = the number of parameters passed from A to components higher in the hierarchy;

g = the number of parameters passed from A to components lower in the hierarchy;

h = the number of data elements written to by A.

Then:

$$\text{FAN OUT}(A) = e + f + g + h$$

Other than those changes to the basic definitions, the derivation, analysis and interpretation remain the same. It is advisable for any organization starting to apply IF metrics to build up confidence by using the simpler form. If these work, then leave it at that. If, and only if, the simpler form fails in environment, in other words we feel confident that no significant relationship exists between the simple measures and the levels of reliability and maintainability, then spend the effort to tailor and pilot the more sophisticated form.

It is encouraging to know that there have been a number of experimental validations of IF metrics that seem to support the claims made for them. Programming groups that have been introduced to IF metrics have been able to make use of them and they also report benefits in the area of design, quality control and system management. They seem to work, but there appears to be some reluctance in the industry as a whole to make use of IF metrics. Perhaps one reason is that managers feel they are a bit 'techie'. Perhaps others feel that they are not yet ready to use sophisticated techniques like IF metrics.

6.5. METRICS ANALYSIS

There is a wide range of techniques that we can use to assess internal attributes of the software product. All of these techniques produce data and, all this data can be expressed in numerical form. But collecting data should be seen as only the first stage of software assessment. We also have to analyze the data to make deductions about the quality of the software product and determine if it is sufficiently good to be released to customers.

The quantity of data we collect will often be quite large. Many textual, data structural and test coverage metrics are defined at the Component level. If we collect, say 20 metrics for each Component and we have 100 Components in a system, then we will have 2000 data points, excluding metrics defined at the subsystem and system levels. It would be difficult to imagine a software assessor simply gazing at pages of figures and rationally arriving at a pass/fail decision for that particular product. We need to use statistics to understand the numbers, to make deductions and then produce evidence to support those deductions. In many cases simply expressing the data in pie charts and histograms can reveal a great deal about the software. Nevertheless, if we wish to uncover the relationships between metrics to validate theories, methods like regression and correlation will be more appropriate. However, when applying any kind of statistics we need to be very careful. Software metrics data is often considered to be unusual in that it does not conform to the normally made assumptions on which many statistical tests and methods are based. However, this does not preclude the meaningful application of statistical techniques. We have many tests at our disposal, which can accommodate other distributional assumptions. It is therefore, important for the researcher to determine the specific statistical tests and methods needed for each analysis in turn. Furthermore, analyzing failure data needs a very particular type of statistics and there is a range of models, which are specifically used to predict future reliability. Nevertheless, there are limitations to the kind of predictions we can make about reliability [MART94].

6.5.1 Using Statistics for Assessment

The primary purpose of applying statistics to metrics data is to gain understanding. Therefore the choice of statistics we use will ultimately be determined by the audience for whom the statistics are intended. If we are providing an evaluation of a software product for project managers then generally we will want to use simple descriptive statistics. We can use simple graphs and tables to point out areas of high and low software quality and make comparisons with other projects or predefined target values.

We will also want to use the metrics data collected from different projects to define and refine the criteria on which the assessments are made. This means deciding the ranges of values that the metrics ought to have. Such an activity is essentially internal to the Quality Assurance department or test laboratory. In this case there is no need to restrict us to simple descriptive statistics, although these will still have a role. There are other more powerful techniques such as regression, correlation and multivariate techniques.

The advent of commercial statistical packages over the last few years has meant that people who have little or no knowledge of the underlying mathematics can readily use statistics. Many of the complex calculations that in the past had to be done manually or hard programmed into specific tools are no longer a concern. One particular facility provided by many such packages is the ability to generate graphs and diagrams automatically. Some packages are

compatible with word processors so that these diagrams can be 'cut and pasted' into assessment reports.

The fact that statistical packages are so easy to use gives rise to the danger of applying inappropriate techniques. We still need to understand the assumptions on which a particular technique is based. If these are ignored then the conclusions are likely to be erroneous. It is all too easy to produce plausible-looking diagrams or correlations that are, in reality, totally meaningless.

Statistics is a large body of knowledge and it would certainly not be possible to describe all the methods that would conceivably be used with software metrics. We do describe here a range of techniques that have been specifically used in the field. These are:

Summary statistics such as mean, median, maximum and minimum; *graphical representations* such as histograms, pie charts and box plots; Principal Components analysis to reduce the number of variables being analyzed; Regression and correlation techniques for uncovering relationships in the data; *Reliability* models for predicting future reliability — which are very different from the other techniques.

We want to see how each of these techniques can be used to interpret particular metrics data. But before we show how we can actually apply the statistics we explore why software metrics are different from other types of data on which statistics have traditionally been applied.

6.5.2 Problems with Metrics Data

Most statistical methods and tests make a number of assumptions about the data being analyzed. For example, the use of F-test in testing the significance of simple least -square regression (fitting a straight line to two variables) assumes that the data for both variables is at least interval and that the errors are approximately normally distributed. Both of these assumptions will often be false for many of the metrics we consider and so we should check carefully before applying any such technique.

Normal distribution

Many statistical tests are based on the assumption that the data under analysis is drawn from a normally distributed population. The frequency distribution for normal data has a bell-shaped curve as shown in Fig. 6.14.

Many types of physical measurement data have been shown to follow this kind of distribution and for this reason it underlies many techniques. However, this is often not the case for software metrics.

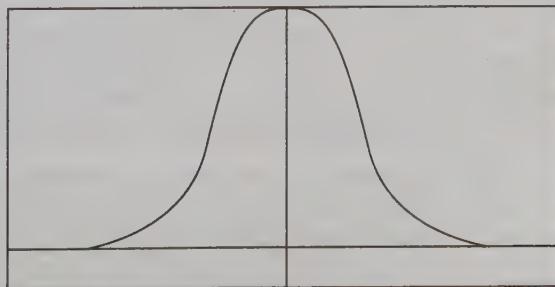


Fig. 6.14: The normal distribution.

This has implications when considering testing for equivalence of mean averages from different samples. When using the t-test for this purpose it is necessary that the sample variable be normally distributed. We can also circumvent this problem by using robust techniques. These robust techniques (often called non-parametric) require few distributional assumptions or perform just as well even when their assumptions are violated.

Outliers

An outlier is a data point, which is outside the normal range of values of a given population. In non-software data, outliers are often due to errors in measurement or are caused by systematic bias. For this reason they are frequently removed from the data set and subsequently ignored. In software metrics data, however, the outliers are often the most interesting data points. For example, if we are measuring component size, it is not uncommon to have a single software component, which is 10 to 20 times larger than any other component. It is precisely these large components, which may give rise to problems in maintainability and so they should certainly not be removed from the data set.

Measurement scale

The scale on which metrics are defined will generally determine which statistics can be meaningfully applied. Most physical measures are ratio, e.g., length, mass, voltage, pressure. Therefore people would be forgiven for assuming the same about software metrics. This is often not the case.

To decide which scale a particular metric is defined on, we need to go back to the actual definition of the metric and reason about the relationship imposed by the attribute being measured.

Multicollinearity

Many multivariate techniques such as multivariate regression require that the variables (*i.e.*, metrics) are independent of each other. Independent in this sense means that they do not correlate with one another. Unfortunately most static flow graph and textual metrics are correlated with size. It does not mean that these metrics all measure size-they measure many different attributes like number of decisions or nesting-but these attributes tend to be highly correlated with size and by implication with each other. This is because a component with a high number of decisions or a high level of nesting is also likely to be large. The same phenomenon of multicollinearity exists for coverage metrics, so branch coverage will be highly correlated with statement coverage, DDP coverage, etc.

The solution to this is to use either factor analysis (FA) or principal components analysis (PCA) to reduce the set of metrics to a smaller set of independent components. FA usually requires the data to be normally distributed, which, as we know, is rarely going to be true. PCA on the other hand is a robust technique because it relies on no distributional assumptions.

PCA is applied to data in order to reduce the number of measures used and to simplify correlation patterns. It attempts to solve these problems by reducing the dimensionality represented by these many related variables to a smaller set of principal components while retaining most of the variation from the original variables. These new principal components are uncorrelated and can act as substitutes for the original variables with little loss of information.

6.5.3 The Common Pool of Data

We often want to compare metrics from one software product with typical values observed in many other products. This requires creating a common pool of data from previous projects. If metrics are defined at the system level then we will have one value per software project. However, metrics defined at finer levels of granularity will contribute to the common pool in varying degrees. The pool of metric values defined at the component level will be influenced more by those software projects, which have a greater number of components. When establishing the common pool we need to be aware of three points.

1. The selection of projects should be representative and not all come from a single application domain or development styles. The exception is when a QA department with a fixed range of developments or a defined process uses the common pool.
2. No single very large project should be allowed to dominate the pool; this is less likely as the number of projects increases.
3. For some projects, certain metrics may not have been collected; we should ensure this is not a source of bias.

Only when these three points are satisfied can we be sure about making comparisons with the common pool.

6.5.4 A Pattern for Successful Applications

Successful applications of metrics abound but are not much talked about in the public literature. Mature metrics can help us to predict expected number of latent bugs, help us to decide how much testing is enough and how much design effort, cost, elapsed time, and all the rest we expect from metrics. Here's what it takes to have a success [BEIZ90].

1. *Any Metric Is Better Than None:* Use simplest possible metric like weight of program listings first and then implement “token counting” and “function counting” as the next step. Worry about the fancy metrics later.

2. *Automation Is Essential:* Any metrics project that relies on having the developers fill out long questionnaires or manually calculate metric values is doomed. They never work. If they work once, they won't the second time. If we've learned anything, it's that a metric whose calculation isn't fully automated isn't worth doing; it probably harms productivity and quality more than any benefit we can expect from metrics.

3. *Empiricism Is Better Than Theory:* Theory is at best a guide to what makes sense to include in a metrics project — there's no theory sufficiently sound today to warrant the use of a single, specific metric above others. Theory tells us what to put into the empirical pot. It's the empirical, statistical data that we must use for guidance.

4. *Use Multifactor Rather Than Single Metrics:* All successful metrics programs use a combination (typically linear) of several different metrics with weights calculated by regression analysis.

5. *Don't Confuse Productivity Metrics with Complexity Metrics:* Productivity is a characteristic of developers and testers. Complexity is a characteristic of programs. It's not always easy to tell them apart. Examples of productivity metrics incorrectly used in lieu of complexity metrics are: number of shots it takes to get a clean compilation, percentage of Components

that passed testing on the first attempt, number of test cases required to find the first bug. There's nothing wrong with using productivity metrics as an aid to project management, but that's a whole different story. Automated or not, a successful metrics program needs developer cooperation. If complexity and productivity are mixed up, be prepared for either or both to be sabotaged to uselessness.

6. Let Them Mature: It takes a lot of projects and a long time for metrics to mature to the point where they're trustworthy. If the typical project takes 12 months, then it takes ten to fifteen projects over a 2 to 3 year period before the metrics are any use at all.

7. Maintain Them: As design methods change, as testing gets better, and as QA functions to remove the old kind of bugs, the metrics based on that past history lose their utility as a predictor of anything. The weights given to metrics in predictor equations have to be revised and continually reevaluated to ensure that they continue to predict what they are intended to predict.

8. Let Them Die: Metrics wear out just like test suites do, for the same reason—the pesticide paradox. Actually, it's not that the metric itself wears out but that the importance we assign to the metric changes with time. We have seen projects go down the tubes because of worn-out metrics and the predictions based on them.

REFERENCES

- [AGGA94] Aggarwal K.K. & Yogesh Singh, "A Modified Approach for Software Science Measures", ACM SIGSOFT Software Engineering Notes, USA, July, 1994.
- [ALBR79] Albrecht A., "Measuring Application Development Productivity", Proc. IBM Application Development Symposium, Monterey, California Oct 14–17, 1979.
- [ALBR83] Albrecht A., & Gaffney J.E., "Software Function Source Lines of Code and Development Effort Prediction: A Software Science Validation", IEEE Trans. Software Engineering, SE-9 639–648, 1983.
- [BACH90] Bache & Monica, "Measures of Testability as a Basis for Quality Assurance, Software Engineering Journal, March, PP-86–92, 1990.
- [BAIL81] Bailey J.W., & Basili V.R., "A meta Model for Software Development Resource Expenditures", Proc. of the Int. Conf. on Software Engineering, 107–116, 1981.
- [BASI75] Basili V.R. & Turner A.J., "Iterative Enhancement: a Practical Technique for Software Development," IEEE Trans. On Software Engg., SE-1, 390–396, Dec. 1975.
- [BEIZ90] Boris Beizer, "Software Testing Techniques", Van Nostrand Reinhold International Co. Ltd., UK, 1990.
- [BOEH81] Boehm B., "Software Engineering Economics", Prentice-Hall, Englewood Cliffs, NJ. 1981.
- [BRUC92] Bruci I. Blum, "Software Engineering — A Holistic View", Oxford University Press, NY, 1992.
- [BULU73] Bulut N., "Invariant Properties of Algorithms", Ph.D. Thesis, Purdue University, August, 1973.
- [GHEZ94] Carlo Ghezzi et. al. "Software Engineering, PHI, 1994.
- [CARD87] Card D.N. & Agresti W.W., "Resolving Software Science Anomaly", Journal of Systems & Software, Vol.7, 29–35, 1987.

- [CHHA2K] Chhabra Jitender Kumar, Dinesh Chutani, Aggarwal K.K., Yogesh Singh, "Effect of Data Coupling on Program Weakness", International Conference on Quality, Reliability and IT at the Turn of the Millennium, New Delhi, Dec. 2000.
- [CHHA01] Chhabra Jitender Kumar, Aggarwal K.K., Yogesh Singh, "Computing Program Weakness using Module Coupling, ACM SIGSOFT Software Engineering Notes, Vol. 27, No. 1, January, 63–66, 2002.
- [CONT86] Conte S.K., Dunsmore H.E., Shen V.Y., "Software Engineering Metrics and Models", The Benjamin/Cummings Pub. C. Inc., California, USA, 1986.
- [DUNS79] Dunsmore H.E. & Gannon J.D., "Analysis of The Effects of Programming Factors on Programming Effort", Journal of systems and software, 141–153, 1980.
- [ELSH76] Elshoff J.L., "An Analysis of Some Commercial PI/1 Programs, IEEE Transactions on Software Engineering, SE-2, 113–120, June, 1976.
- [JAME78] Elshoff J.L., "An Inrestigation in to the Effects of the Counting Method Used on Software Science Measurements," ACM SIGPLAN Notices, Vol 13, 30–45, Feb, 1978.
- [FITZ78] Fitzsimmory A. & Love T., "A Review and Evaluation of Software Science", ACM Computing Surveys, Vol 10, March, 1978.
- [HALS77] Halstead M.H., "Elements of Software Science", New York, Elsevier North Holland, 1977.
- [HAME85] Hamer P. & Frewin G., "Software Metrics: A Critical Overview", Pergamon Infotech State of the art report 13 (2), 1985.
- [HENR81] Henry S. & Kafura D., "Software Structure Metrics Based on Information Flow", IEEE Trans. on Software Engineering SE-7, 5, 510–518, Sept 1981.
- [INCE89] Ince D., "Software Metrics: Measurement for Software Control and Assurance", New York: Elsevier, 1989.
- [JENS85] Jensen H.A. & Vairavan K., "An Experimental Study of Software Metrics for Real Time Software", IEEE Trans. on Software Engineering, 231–234, Feb, 1985.
- [KITC90] Kitchenham B., "Empirical Studies of Assumption Underlying Software Cost Estimation Models", Proc. of European COCOMO User Group, 1990.
- [MART94] Martin Neh, "Software Metrics for Product Assessment", McGraw Hill Book Co., UK, 1994.
- [MATS94] Matson E.M., et al., "Software Development Cost Estimation Using Function Points", IEEE Trans. on software Engineering", Vol 20, No.4. April, 1994.
- [MEHN86] Mehnadiralta B., & Grover P.S., "Measuring Computer Programs", Proc. of CSI Annual Convention, India, 1986.
- [GOOD93] Paul Goodman, "Practical Implementation of Software Metrics", McGraw Hill Book Company, UK, 1993.
- [RAMA88] Ramamurthy B. & Melton A., "A Synthesis of Software Science Measures and the Cyclomatic Number," IEEE Trans. on Software Engineering Vol. 14. No.8, August, 1988.
- [RUBE68] Rubey R.J. & R.D. Hartwick, "Quantitative Measurement of Program Quality", Proc. ACM Nat. Conf. PP 671–677, 1968.
- [SESH61] Sheshu S. & Recd M.B., "Linear Graphs & Electrical Networks", Addison Wesley, USA, 1961.
- [SHEN83] Shen V.Y., Conte S.D., and Dun Smore H.E." Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support", IEEE Trans. on Software Engg., Vol. SE-9 No.2., 1983,155–165, March, 1983.
- [YOGE95] Singh Yogesh, "Metrics and Design Techniques for Reliable Software", Ph.D Thesis, Kurukshetra University, Kurukshetra (India) , July, 1995.

- [YOG98] Singh Yogesh & Pradeep Bhatia, "Module Weakness — A New Measure", ACM SIGSOFT Software Engineering Notes, 81, July 1998.

[STEP95] Stephen Treble and Neil Douglas, "Sizing and Estimating Software in Practice", McGraw Hill Book Company, London, 1995.

[EJIO91] "Software Engineering with Formal Metrics", QED Information Sciences, Wellesley, Massachusetts 1991.

[STRO67] Stroud J.M., "The Fine Structure of Psychological Time", Annals of New York Academy of Science 138, 2, 623–631, 1967.

[THEB83] Thebaut S.M., "The Saturation Effect in Large Scale Software Development Its Impact and Control, Ph.D. Thesis, Department of Computer Science, Purdue University, West Lafayette, IN, May, 1983.

[TRAC88] Tracz W., "Software Reuse Emerging Technologies", IEEE Computer Society Press, Washington DC, 1988.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions.

- (c) Number of components dependent on component A
(d) None of the above.

6.9. 'FAN OUT' of a component A is defined as
(a) number of components related to component A
(b) number of components dependent on component A
(c) number of components that are called by component A
(d) none of the above.

6.10. Which is not a size metric?
(a) LOC
(b) Function count
(c) Program length
(d) Cyclomatic complexity.

6.11. Which one is not a measure of software science theory?
(a) Vocabulary
(b) Volume
(c) Level
(d) Logic

6.12. A human mind is capable of making how many number of elementary mental discriminations per second (*i.e.*, stroud number)?
(a) 5 to 20
(b) 20 to 40
(c) 1 to 10
(d) 40 to 80

6.13. Minimal implementation of any algorithm was given the following name by Halstead:
(a) Volume
(b) Potential volume
(c) Effective volume
(d) none of the above.

6.14. Program volume of a software product is
(a) $V = N \log_2 n$
(b) $V = (N/2) \log_2 n$
(c) $V = 2N \log_2 n$
(d) $V = N \log_2 n + 1$

6.15. Which one is the international standard for size measure?
(a) LOC
(b) Function count
(c) Program length
(d) None of the above.

EXERCISES

- 6.1.** Define software metrics. Why do we really need metrics in software?
 - 6.2.** Discuss the areas of applications of software metrics. What are the problems during implementation of metrics in any organisation ?
 - 6.3.** What are various categories of software metrics ? Discuss with the help of suitable examples.
 - 6.4.** Explain the Halstead theory of software science. Is it significant in today's scenario of component based software development ?
 - 6.5.** What is the importance of language level in Halstead theory of software science?
 - 6.6.** Give Halstead's software science measures for:

(i) Program Length	(ii) Program volume
(iii) Program level	(iv) Effort
(v) Language level.	
 - 6.7.** For a program with number of unique operators $\eta_1 = 20$ and number of unique operands $\eta_2 = 40$, Compute the following:

(i) Program volume	(ii) Effort and time
(iii) Program length	(iv) Program level.

- 6.8. Develop a small software tool that will perform a Halstead analysis on a programming language source code of your choice.
- 6.9. Write a program in C and also PASCAL for the calculation of the roots of a quadratic equation. Find out all software science metrics for both the programs. Compare the outcomes and comment on the efficiency and size of both the source codes.
- 6.10. How should a procedure identifier be considered, both when declared and when called? What about the identifier of a procedure that is passed as a parameter to another procedure?
- 6.11. It is interesting to examine how the ratio $(N-\hat{N})/N$ varies when a program is divided into parts. Actually, some experiments show that the partitioning of vocabularies due to program modularization maintains the stability of the ratio $(N-\hat{N})/N$. Two extreme situations may occur:
- η_1 and η_2 are the same for all parts.
 - η_1 and η_2 are partitioned into disjoint subsets by modularization.
- Compute the variations of \hat{N} for a program with $N = \hat{N} = 72$, $\eta_1 = 4$ and $\eta_2 = 16$ when the program is divided into two parts, under the two extreme assumptions.
- Warning:** It may be difficult to divide a program in such a way so as to satisfy the latter assumption. (At least one procedure definition & call must share an identifier.) For large values of the quantities involved, however, we can assume that, at least, η_1 and η_2 have small intersections with respect to their size [GHEZ94].
- 6.12. Define data structure metrics. How can we calculate amount of data in a program ?
- 6.13. Describe the concept of module weakness. Is it applicable to programs also ?
- 6.14. Write a program for the calculation of roots of a quadratic equation. Generate cross reference list for the program and also calculate \overline{LV} , γ and WM for this program.
- 6.15. Show that the value of SP at a particular statement is also the value of LV at that point.
- 6.16. Discuss the significance of data structure metrics during testing.
- 6.17. What are information flow metrics ? Explain the basic information flow model.
- 6.18. Discuss the problems with metrics data. Explain two methods for the analysis of such data.
- 6.19. Show why and how software metrics can improve the software process. Enumerate the effect of metrics on software productivity.
- 6.20. Why does lines of code (LOC) not measure software nesting and control structures?
- 6.21. Several researchers in software metrics concentrate on data structure to measure complexity. Is data structure a complexity or quality issue, or both?
- 6.22. List the benefits and disadvantages of using Library routines rather than writing own code.
- 6.23. Compare software science measures and function points as measures of complexity. Which do you think more useful as a predictor of how much particular software's development will cost?
- 6.24. Some experimental evidence suggests that the initial size estimate for a project affects the nature and results of the project. Consider two different managers charged with developing the same application. One estimates that the size of the application will be 50,000 lines, while the other estimates that it will be 100,000 lines. Discuss how these estimates affect the project throughout its life cycle.
- 6.25. Which one is the most appropriate size estimation technique and why ?

7

Software Reliability

Contents

7.1 Basic Concepts

- 7.1.1 What is Software Reliability ?
- 7.1.2 Software Reliability & Hardware Reliability
- 7.1.3 Failures & Faults
- 7.1.4 Environment
- 7.1.5 Uses of Reliability Studies

7.2 Software Quality

- 7.2.1 McCall Software Quality Model
- 7.2.2 Boehm Software Quality Model
- 7.2.3 ISO 9126

7.3 Software Reliability Models

- 7.3.1 Basic Execution Time Model
- 7.3.2 Logarithmic Poisson Execution Time Model
- 7.3.3 Calender Time Component
- 7.3.4 The Jelinski-Moranda Model
- 7.3.5 The Bug Seeding Model

7.4 Capability Maturity Model

- 7.4.1 Maturity Levels
- 7.4.2 Key Process Areas
- 7.4.3 Common Features

7.5 ISO 9000

- 7.5.1 Mapping of ISO 9001 to the CMM
- 7.5.2 Contrasting ISO 9001 and the CMM
- 7.5.3 Conclusion

Software Reliability

Software reliability. Does it exist ? Computer program instructions can not break or wear out. Hence predicting software reliability is a different concept as compared to predicting hardware reliability. Software becomes more reliable over time, instead of wearing out. It becomes obsolete as the environment for which it was developed changes. Hardware redundancy allows us to make a system reliable as we desire, if we use large number of components with given reliability. We do not have such techniques in software and we may not get such techniques in foreseeable future.

Initially, problems in programming were blamed to the severe constraints imposed by the hardware. However, hardware has now become more reliable, flexible and versatile, but the problems with programming have not decreased.

7.1 BASIC CONCEPTS

The term reliability is often misunderstood in the software field since software does not break or wear-out in the physical sense. It either works in a given environment or it does not. Hence traditional "bath tub" curve of hardware reliability is not applicable here. The "bath tub curve" is given in Fig. 7.1.

As indicated, there are three phases in the life of any hardware component i.e., burn-in, useful life & wear-out. In burn-in phase, failure rate is quite high initially, and it starts decreasing gradually as the time progresses. It may be due to initial testing in the premises of the organisation. During useful life period, failure rate is approximately constant. Failure rate increases in wear-out phase due to wearing out/aging of components. The best period is useful life period. The shape of this curve is like a "bath tub" and that is why it is known as bath tub curve.

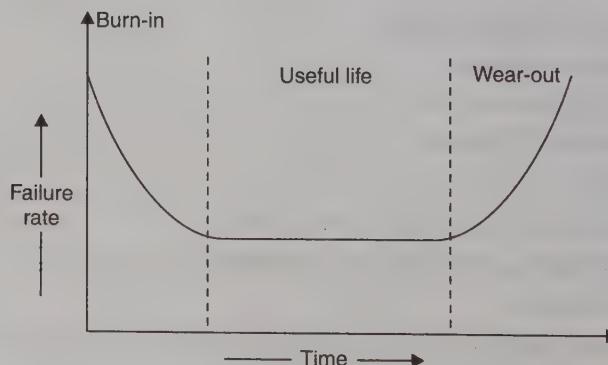


Fig. 7.1: Bath tub curve of hardware reliability.

We do not have wear out phase in software. The expected curve for software is given in Fig. 7.2.

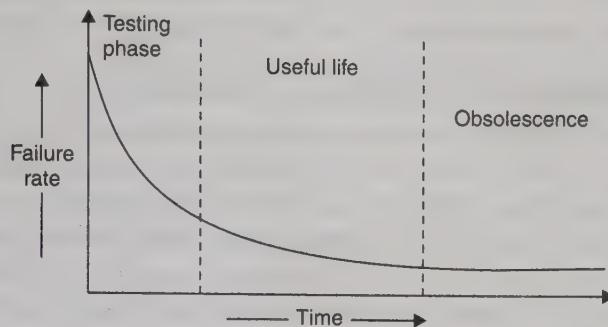


Fig. 7.2: Software reliability curve (failure rate versus time).

Software may be retired only if it becomes obsolete. Some of contributing factors are given below:

- change in environment
- change in infrastructure/technology
- major change in requirements
- increase in complexity
- extremely difficult to maintain
- deterioration in structure of the code
- slow execution speed
- poor graphical user interfaces.

7.1.1 What is Software Reliability?

According to Bev Littlewood [LITT79]: “Software reliability means operational reliability. Who cares how many bugs are in the program ? We should be concerned with their effect on its operations”.

As per IEEE standard [IEEE90]: “Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time”.

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the inputs are free of error [KOPE79]. Hence it is the probability that the software will work without failure for a specified period of time in a given environment. Here environment and time is fixed. Reliability is for fixed time under given environment or stated conditions. So, reliability value is always for a well defined domain.

The most acceptable definition of software reliability is: “It is the probability of a failure free operation of a program for a specified time in a specified environment [MUSA87]”.

For example, a time-sharing system may have a reliability of 0.95 for 10 hr when employed by the average user. This system, when executed for 10 hr, would operate without failure for

95 of these periods out of 100. As a result of the general way in which we defined failure, note that the concept of software reliability incorporates the notion of performance being satisfactory. For example, excessive response time at a given load level may be considered unsatisfactory so that a routine must be recorded in more efficient form.

7.1.2 Software Reliability and Hardware Reliability

The field of hardware reliability has been established for some time. Hence, one might ask how software reliability relates to it. In reality, the division between hardware and software reliability is somewhat artificial. Both may be defined in the same way. Therefore, one may combine hardware and software component reliabilities to get system reliability. Both depend on the environment. The source of failures in software is design faults, while the principal source in hardware has generally been physical deterioration. However, the concepts and theories developed for software reliability could really be applied to any design activity, including hardware design.

Once a software (design) defect is properly fixed, it is in general fixed for all time. Failure usually occurs only when a program (design) is exposed to an environment that it was not developed or tested for. Although manufacturing can affect the quality of physical components, the replication process for software (design) is trivial and can be performed to very high standards of quality. Since introduction and removal of design faults occurs during software development, software reliability may be expected to vary during this period.

The *design reliability* concept has not been applied to hardware to that extent. The probability of failure due to wear and other physical causes has usually been much greater than that due to an unrecognised design problem. It was possible to keep hardware design failures low because hardware was generally less complex logically than software. Hardware design failures had to be kept low because retrofitting of manufactured items in the field was very expensive. Awareness of the work that is going on in software reliability, plus a growing realisation of the importance of design faults, may now be having an effect on hardware reliability too. This growing awareness is strengthened by the parallels that people are starting to draw between software engineering and chip design.

A final characteristic of software reliability is that it tends to change continually during test periods. This happens either as new problems are introduced, when new code is written or when repair action removes problems that exist in the code. Hardware reliability may change during certain periods, such as initial burn-in or the end of useful life. However, it has a much greater tendency than software toward a constant value.

Despite the foregoing differences, we can develop software reliability theory in a way that is compatible with hardware reliability theory. Thus system reliability figures may be computed using standard hardware combinatorial techniques [SHOO86]. Hardware and software reliability share many similarities and some differences [LLOY77]. One must not err on the side of assuming that software always presents unique problems, but one must also be careful not to carry analogies too far.

7.1.3 Failures and Faults

What do we mean by the term software failure? It is the departure of the external results of program operation from requirements. So our *failure* is something dynamic. The program has

to be executing for a failure to occur. The term failure relates to the behaviour of the program. This very general definition of failure is deliberate. It can include such things as deficiency in performance attributes and excessive response time.

A *fault* is the defect in the program that, when executed under particular conditions, causes a failure. There can be different sets of conditions that cause failures, or the conditions can be repeated. Hence a fault can be the source of more than one failure. A fault is a property of the program rather than a property of its execution or behaviour. It is what we are really referring to in general when we use the term bug. A fault is created when a programmer makes an error. It's very important to make the failure-fault distinction!

Reliability quantities have usually been defined with respect to time, although it would be possible to define them with respect to other variables. We are concerned with three kinds of time. The *execution time* for a program is the time that is actually spent by a processor in executing the instructions of that program. The second kind of time is *calendar time*. It is the familiar time that we normally experience. Execution time is important, because it is now generally accepted that models based on execution time are superior. However, quantities must ultimately be related back to calendar time to be meaningful to engineers or managers. Sometimes the term *clock time* is used for a program. It represents the elapsed time from start to end of program execution on a running computer. It includes wait time and the execution time of other programs. Periods during which the computer is shut down are not counted. If computer utilisation by the program, which is the fraction of time the processor is executing the program, is constant, clock time will be proportional to execution time.

There are four general ways of characterising failure occurrences in time:

1. time of failure,
2. time interval between failures,
3. cumulative failures experienced upto a given time,
4. failures experienced in a time interval.

These are illustrated in Tables 7.1 and 7.2.

Note that all the foregoing four quantities are random variables. By *random* we mean that the values of the variables are not known with certainty. There are many possible values each associated with a probability of occurrence. For example, we don't really know when the next failure will occur. If we did, we would try to prevent or avoid it. We only know a set of possible times of failure.

There are at least two principal reasons for this randomness. First, the commission of errors by programmers, and hence the introduction of faults, is a very complex, unpredictable process. Hence the location of faults within the program are unknown. Second, the condition of what next? In addition, the relationship between program function requested and code path executed, although theoretically determinable, may not be so in practice because it is so complex. Since failures are dependent on the presence of a fault in the code and its execution in the execution of a program, these are generally unpredictable. For example, with a telephone switching system, how do you know what type of call will be made next in the context of certain machine states.

A third complicating element is thus introduced that argues for the randomness of the failure process.

Table 7.1: Time based failure specification

<i>Failure Number</i>	<i>Failure Time(sec)</i>	<i>Failure interval(sec)</i>
1	8	8
2	18	10
3	25	7
4	36	11
5	45	9
6	57	12
7	71	14
8	86	15
9	104	18
10	124	20
11	143	19
12	169	26
13	197	28
14	222	25
15	250	28

Table 7.2: Failure based failure specification

<i>Time (sec)</i>	<i>Cumulative Failures</i>	<i>Failures in interval (30 sec)</i>
30	3	3
60	6	3
90	8	2
120	9	1
150	11	2
180	12	1
210	13	1
240	14	1

Table 7.3 illustrates a typical probability distribution of failures that occurs within a time period of execution. Each possible value of the random variable of number of failures is given along with its associated probability. The probabilities, of course, add to 1. Note that here the random variable is discrete, as the number of failures must be an integer. Note that the most probable number of failures is 2 for $t = 1$ hr. The mean or average number of failures can be computed. We multiply each possible value by the probability it can occur and add all the products. The mean is 3.04 failures for $t = 1$ hour.

Table 7.3: Probability distribution at times t_A and t_B

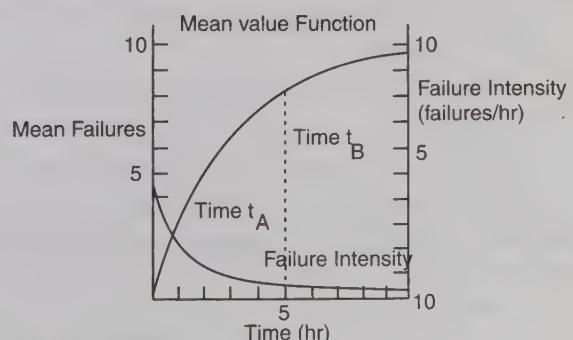
Value of random variable (failures in time period)	Probability	
	Elapsed time $t_A = 1 \text{ hr}$	Elapsed time $t_B = 5 \text{ hr}$
0	0.10	0.01
1	0.18	0.02
2	0.22	0.03
3	0.16	0.04
4	0.11	0.05
5	0.08	0.07
6	0.05	0.09
7	0.04	0.12
8	0.03	0.16
9	0.02	0.13
10	0.01	0.10
11	0	0.07
12	0	0.05
13	0	0.03
14	0	0.02
15	0	0.01
Mean failures	3.04	7.77

We will look at the time variation from two different viewpoints, the mean value function and the failure intensity function. The mean value function represents the average cumulative failures associated with each time point. The failure intensity function is the rate of change of the mean value function or the number of failures per unit time. For example, you might say 0.01 failure/hr or 1 failure/100 hr. Strictly speaking, the failure intensity is the derivative of the mean value function with respect to time, and is an instantaneous value.

A random process whose probability distribution varies with time is called non-homogeneous. Most failure processes during test fit this situation. Fig. 7.3 illustrates the mean value and the related failure intensity functions at time t_A and t_B . Note that the mean failures experienced increases from 3.04 to 7.77 between these two points, while the failure intensity decreases.

Failure behaviour is affected by two principal factors:

1. the number of faults in the software being executed,
2. the execution environment or the operational profile of execution.

**Fig. 7.3:** Mean value & failure intensity functions.

The number of faults in software is the difference between the number introduced and the number removed.

Faults are introduced when the code is being developed by programmers. They may introduce the faults during original design or when they are adding new features, making design changes, or repairing faults that have been identified. In general, only code that is new or modified results in faults introduction. Code that is inherited from another application does not usually introduce any appreciable number of faults, except possibly in the interfaces. It generally has been thoroughly debugged in the previous application. Note that the process of faults removal introduces some new faults because it involves modification or writing of new code.

Faults removal obviously can't occur unless you have some means of detecting the fault in the first place. Thus fault removal resulting from execution depends on the occurrence of the associated failure. Occurrence depends both on the length of time for which the software has been executing and on the execution environment or operational profile. When different functions are executed, different faults are encountered and the failures that are exhibited tend to be different; thus the environmental influence. We can often find faults without execution. They may be found through inspection, compiler diagnostics, design or code reviews, or code reading.

7.1.4 Environment

Let us scrutinise the term environment. The environment is described by the operational profile. We need to build up to the concept of the operational profile through several steps. It is possible to view the execution of a program as a single entity. The execution can last for months or even years for a real time system. However, it is more convenient to divide the execution into runs. The definition of run is somewhat arbitrary, but it is generally associated with some function that the program performs. Thus, it can conveniently describe the functional environment of the program. Runs that are identical repetitions of each other are said to form a run type. The proportion of runs of various types may vary, depending on the functional environment. Examples of a run type might be:

1. a particular transaction in an airline reservation system or a business data processing system,
2. a specific cycle in a closed loop control system (for example, in a chemical process industry),
3. a particular service performed by an operating system for a user.

During test, the term test case is sometimes used instead of run type.

We next need to understand the concept of the input variable. This is a variable that exists external to the program and is used by the program in executing its function. For an airline reservation, *destination* might be an input variable. One generally has a large quantity of input variables associated with the program, and each set of values of these variables characterise an input state.

In effect, the input state identifies the particular run type that you're making. Therefore, runs can always be classified by their input states. Again, taking the case of the airline reservation system, the input state might be characterised by particular values of origin,

destination, airline, day and flight number. The set of all possible input states is known as the input space.

Similarly, an output variable is a variable that exists external to a program and is set by it. An output state is a set of values of all output variables associated with a run of a program. In the airline reservation system, an output state might be the set of values of variables printed on the ticket and on different reports used in operating the airline. It can now be seen that a failure involves a departure of the output state from what it is expected to be.

The run types required of the program by the environment can be viewed as being selected randomly. Thus, we define the operational profile as the set of run types that the program can execute along with possibilities with which they will occur. In Fig. 7.4, we show two of many possible input states A and B, with their probabilities of occurrence. The part of the operational profile for just these two states is shown in Fig. 7.5. In reality, the number of possible input states is generally quite large. A realistic operational profile is illustrated in Fig. 7.6. Note that the input states have been located on the horizontal axis in order of the probabilities of their occurrence. They have been placed close together so that the operational profile would appear to be a continuous curve.

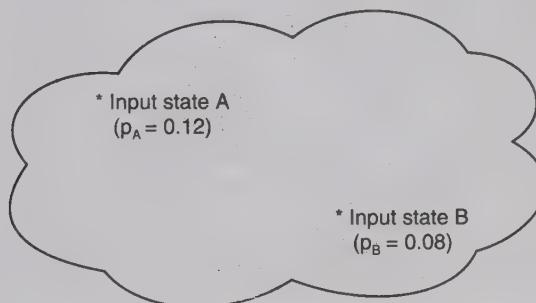


Fig. 7.4: Input Space

Probability of occurrence

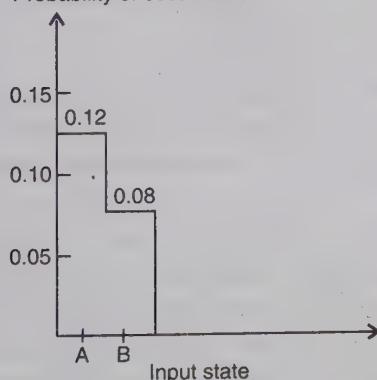


Fig. 7.5: Portion of operational profile

Probability of occurrence

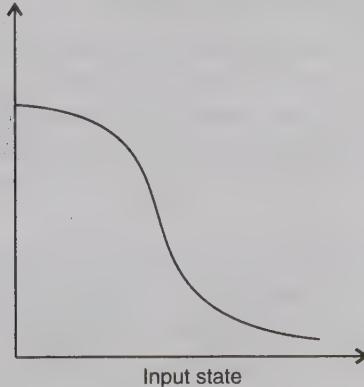


Fig. 7.6: Operational profile

Failure intensity is an alternative way of expressing reliability. We have discussed the example of the reliability of a particular system being 0.95 for 10 hr of time. An equivalent

statement could be the failure intensity of 0.05 failure/hr. Each specification has its advantages. The failure intensity statement is more economical, as you only have to give one number. However, the reliability statement is better suited to the combination of reliabilities of components to get system reliability. If the risk of failure at any point in time is of paramount concern, failure intensity may be the more appropriate measure. Such would be the case for a nuclear power plant. When proper operation of a system to accomplish some function with a time duration is required reliability specification is often best. An example would be a space flight to the moon. Fig. 7.7 shows how failure intensity and reliability typically vary during a test period, as faults are removed. Note that we define failure intensity, just like we do reliability with respect to a specified environment.

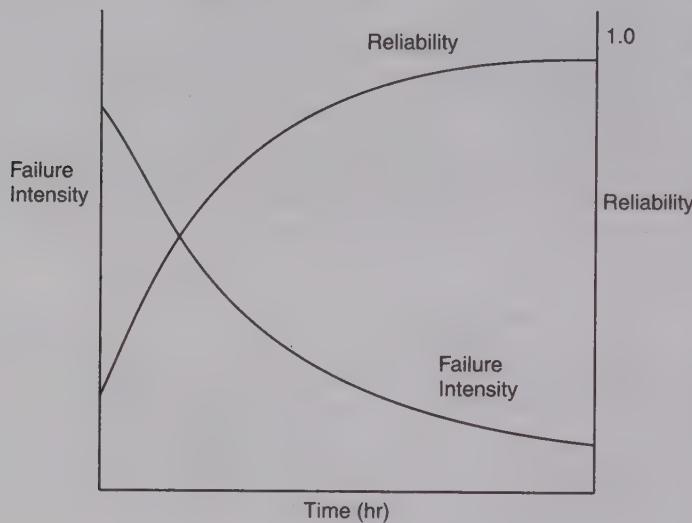


Fig. 7.7: Reliability and failure intensity

7.1.5 Uses of Reliability Studies

Pressures have been increasing for achieving a more finely tuned balance among product and process characteristics, including reliability. Trade-offs among product components with respect to reliability are also becoming increasingly important. Thus an important use of software reliability measurement is in system engineering. However, there are at least four other ways in which software reliability measures can be of great value to the software engineer, manager or user.

First, you can use software reliability measures to evaluate software engineering technology quantitatively. New techniques are continually being proposed for improving the process of developing software, but unfortunately they have been exposed to little quantitative evaluation. Because of the inability to distinguish between good and bad, new technology has often led to a general resistance to change that is counter productive. Software reliability measures offer the promise of establishing at least one criterion for evaluating the new technology. For example, you might run experiments to determine the decrease in failure intensity (failures per unit time) at the start of system test resulting from design reviews. A

quantitative evaluation such as this makes the benefits of good software engineering technology highly visible.

Second, software reliability measures offer you the possibility of evaluating development status during the test phases of a project. Methods such as intuition of designers or test team percent of tests completed, and successful execution of critical functional tests have been used to evaluate testing progress. None of these have been really satisfactory and some have been quite unsatisfactory. An objective reliability measure (such as failure intensity) established from test data provides a sound means of determining status. Reliability generally increases with the amount of testing. Thus, reliability can be closely linked with project schedules. Furthermore, the cost of testing is highly correlated with failure intensity improvement. Since two of the key process attributes that a manager must control are schedule and cost, reliability can be intimately tied in with project management.

Third, one can use software reliability measures to monitor the operational performance of software and to control new features added and design changes made to the software. The reliability of software usually decreases as a result of such changes. A reliability objective can be used to determine when, and perhaps how large, a change will be allowed. The objective would be based on user and other requirements. For example, a freeze on all changes not related to debugging can be imposed when the failure intensity rises above the performance objective.

Finally, a quantitative understanding of software quality and the various factors influencing it and affected by it enriches into the software product and the software development process. One is then much more capable of making informed decisions.

7.2 SOFTWARE QUALITY

Our objective of software engineering is to produce good quality maintainable software in time and within budget. Here quality is very important. What do we understand with the term "quality"? It is not easy to define quality. People understand quality, appreciate quality but may not be able to clearly express the same. It is like beauty which is very much in the eyes of the beholder.

Different people understand different meanings of quality like:

- conformance to requirements
- fitness for the purpose
- level of satisfaction

If a product is meeting its requirements, we may say it is a good quality product. We expect that requirements are clearly stated and can not be misunderstood. Everything is measured with respect to requirements and if it matches, product is a quality product. If a car is designed with a maximum speed of 150 km/hour and if it fails to achieve in the field, then it is not meeting the requirements. If two cars are designed with different style, performance & economy and both are up to standards set for them, then both are quality cars.

When user uses the product and finds the product fit for its purpose, he/she feels that product is of good quality. If product is meeting user requirements, a feeling of satisfaction may emerge and this satisfaction is nothing but the satisfaction for quality.

In a broad sense, the users views of quality must deal with the product's ease of installation, operational efficiency, and convenience. If the product is easy to handle and we comfortably remember how to use it, then customer satisfaction level may increases.

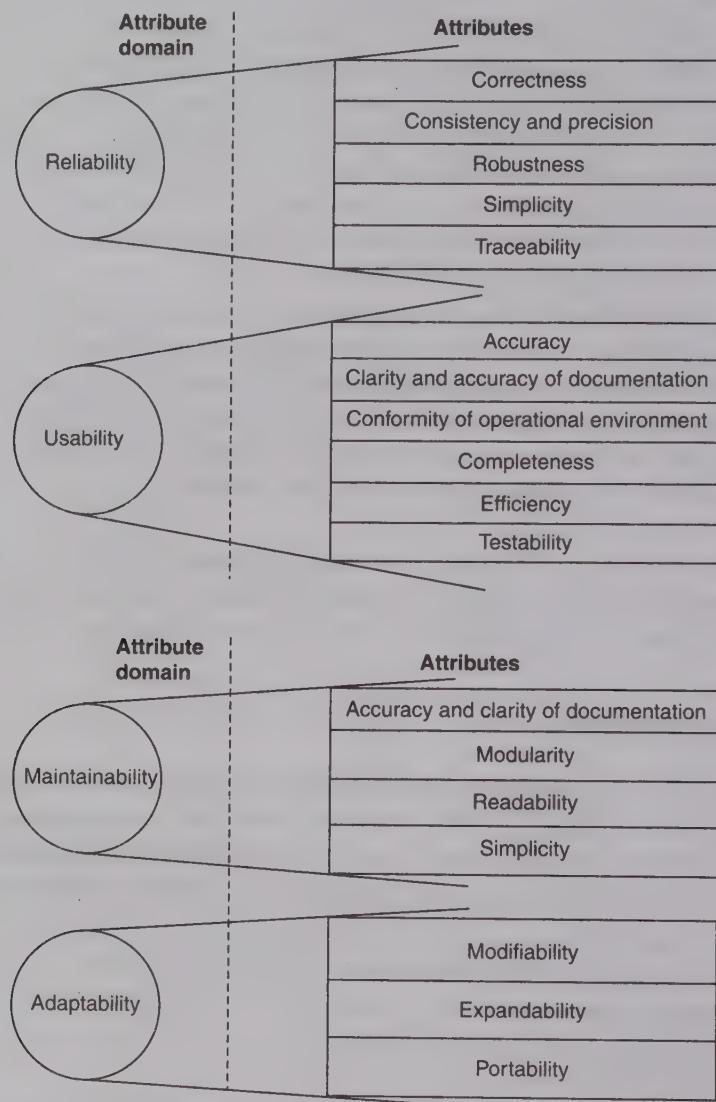


Fig. 7.8: Software quality attributes

Quality has many characteristics and some are related to each other. In software, the quality is commonly recognised as "lack of bugs" in the program. If a software has too many functional defects, then, it is not meeting its basic requirement of functionality. This is usually expressed in two ways:

- (i) **Defect rate:** number of defects per million lines of source code, per function point or any other unit.

(ii) **Reliability:** generally measured as number of failures per 't' hours of operation, mean time to failure or probability of failure free operation in a specified time under specified environment.

When we deal with software quality, a list of attributes is required to be defined that are appropriate for software. There are four attribute domains [DUNN90], which should be defined. These are usually the ones most entrusted by the customer:

- Reliability
- Usability
- Maintainability
- Adaptability

These four attribute-domains can be divided into attributes that are more commonly understood by the software community and are given in Fig. 7.8. The details of software quality attributes are given in Table 7.4.

Table 7.4: Software quality attributes

1.	Reliability	The extent to which a software performs its intended functions without failure.
2.	Correctness	The extent to which a software meets its specifications.
3.	Consistency & precision	The extent to which a software is consistent and give results with precision.
4.	Robustness	The extent to which a software tolerates the unexpected problems.
5.	Simplicity	The extent to which a 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 & Accuracy of documentation	The extent to which documents are clearly & accurately written.
10.	Conformity of operational environment	The extent to which a software is in conformity of operational environment.
11.	Completeness	The extent to which a software has specified functions.
12.	Efficiency	The amount of computing resources and code required by software to perform a function.
13.	Testability	The effort required to test a software to ensure that it performs its intended functions.

(Contd....)

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 a software is readable in order to understand.
17.	Adaptability	The extent to which a software is adaptable to new platforms & technologies.
18.	Modifiability	The effort required to modify a software during maintenance phase.
19.	Expandability	The extent to which a software is expandable without undesirable side effects.
20.	Portability	The effort required to transfer a program from one platform to another platform.

The attribute domain and attributes are also named as the factor and criteria. There are many models of software quality and some are discussed here.

7.2.1 McCall Software Quality Model

McCall et. al [MCCA77] model of software quality was introduced in 1977 and many quality factors were incorporated. The model distinguishes between two levels of quality attributes. Higher level quality attributes are known as quality factors. These are external attributes and can be measured directly. The second level of quality attributes is named as quality criteria. Quality criteria can be measured either subjectively or objectively. The software quality factors are organised in three product quality factors as shown in Fig. 7.9.

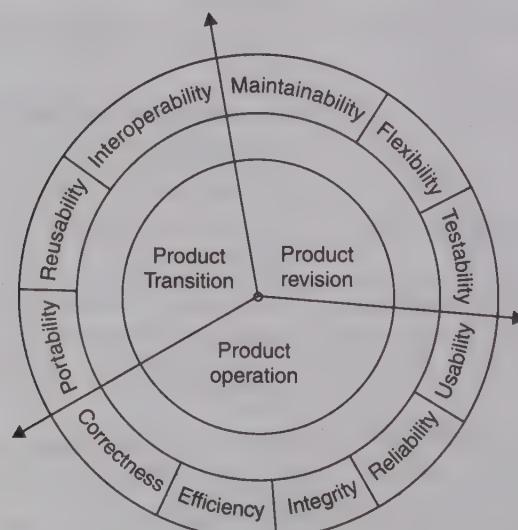


Fig. 7.9: Software quality factors

(i) **Product Operation:** Here, factors which are related to the operation of a product are combined. The factors are:

- Correctness
- Efficiency
- Integrity
- Reliability
- Usability.

These five factors are related to operational performance, convenience, ease of usage and its correctness. These factors play a very significant role in building customer's satisfaction.

(ii) **Product Revision:** The factors which are required for testing & maintenance are combined and are given below:

- Maintainability
- Flexibility
- Testability.

These factors pertain to the testing & maintainability of software. They give us idea about ease of maintenance, flexibility and testing effort. Hence, they are combined under the umbrella of product revision.

(iii) **Product Transition:** We may have to transfer a product from one platform to another platform or from one technology to another technology. The factors related to such a transfer are combined and are given below:

- Portability
- Reusability
- Interoperability.

Most of the quality factors are explained in Table 7.4. The remaining factors are given in Table 7.5.

Table 7.5: Remaining quality factors (others are in Table 7.4)

Sr. No.	Quality Factor	Purpose
1.	Integrity	The extent to which access to software or data by the unauthorised 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.

Quality criteria

The second level of quality attributes are termed as quality criteria. We have eleven quality factors and each quality factor has many second level of quality attributes which are shown in Fig. 7.10. Second level attributes are internal attributes. However, users and managers are interested in the higher level external quality attributes. For example, we may not directly

measure the reliability of a software system. We may however, directly measure the number of defects encountered so far. This direct measure can be used to obtain insight into the reliability of the system. This involves a theory of how the number of defects encountered relates to reliability, which can be ascertained on good grounds. For most other aspects of quality though, the relation between the attributes that can be measured directly and the external attributes we are interested in is less obvious, to say the least [VLIE02]. The relationship between quality factors and quality criteria are given in Table 7.5(a). The definitions & details of these quality criteria are discussed in Table 7.5(b).

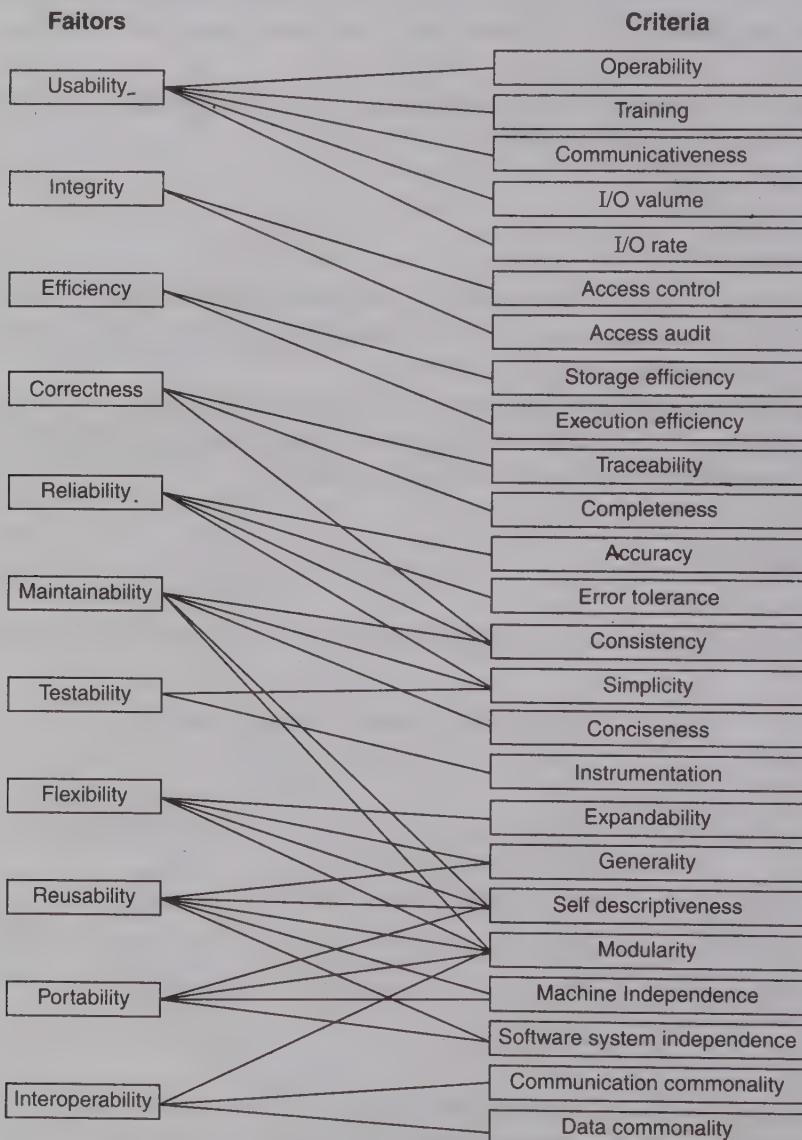


Fig. 7.10: McCall's quality model

Table 7.5(a): Relation between quality factors and quality criteria

Sr. No.	Quality Criteria	Usability	Integrity	Efficiency	Correctness	Reliability	Maintainability	Testability	Flexibility	Reusability	Portability	Interoperability
1.	Operability	x										
2.	Training	x										
3.	Communicativeness	x										
4.	I/O volume	x										
5.	I/O rate	x										
6.	Access control		x									
7.	Access Audit		x									
8.	Storage efficiency			x								
9.	Execution Efficiency			x								
10.	Traceability				x							
11.	Completeness				x							
12.	Accuracy					x						
13.	Error tolerance					x						
14.	Consistency				x	x	x					
15.	Simplicity					x	x	x				
16.	Conciseness						x					
17.	Instrumentation							x				
18.	Expandability								x			
19.	Generality								x	x		
20.	Self-descriptiveness						x	x	x	x		
21.	Modularity						x	x	x	x	x	x
22.	Machine independence									x	x	
23.	S/W system independence									x	x	
24.	Communication commonality											x
25.	Data commonality											x

Table 7.5(b): Software quality criteria

Sr. No.	Quality criteria	Definition / Purpose
1.	Operability	The ease of operation of the software
2.	Training	The ease with which new users can use the system
3.	Communicativeness	The ease with which inputs and outputs can be assimilated.

(Contd....)

Sr. No.	Quality Criteria	Definition / Purpose
4.	I/O volume	It is related to the I/O volume.
5.	I/O rate	It is the indication of I/O rate.
6.	Access control	The provisions for control and protection of the software and data.
7.	Access audit	The ease with which software and data can be checked for compliance with standards or other requirements.
8.	Storage efficiency	The run-time storage requirements of the software.
9.	Execution efficiency	The run-time efficiency of the software.
10.	Traceability	The ability to link software components to requirements.
11.	Completeness	The degree to which a full implementation of the required functionality has been achieved.
12.	Accuracy	The precision of computations and output.
13.	Error tolerance	The degree to which continuity of operation is ensured under adverse conditions.
14.	Consistency	The use of uniform design and implementation techniques and notations throughout a project.
15.	Simplicity	The ease with which the software can be understood.
16.	Conciseness	The compactness of the source code, in terms of lines of code.
17.	Instrumentation	The degree to which the software provides for measurements of its use or identification of errors.
18.	Expandability	The degree to which storage requirements or software functions can be expanded.
19.	Generability	The breadth of the potential application of software components.
20.	Self-descriptiveness	The degree to which the documents are self explanatory.
21.	Modularity	The provision of highly independent modules.
22.	Machine independence	The degree to which software is dependent on its associated hardware.
23.	Software system independence	The degree to which software is independent of its environment.
24.	Communication commonality	The degree to which standard protocols and interfaces are used.
25.	Data commonality	The use of standard data representations.

It is not easy to measure many quality factors. We may have to apply software metrics, if possible, to measure such factors. The quality factors are not independent, but may overlap. Some factors may impact others in a positive sense, while others may do so negatively. We

may have to study & understand such relationships very carefully. A subjective assessment of some criteria can be obtained by giving a rating on a scale from, say, 0(extremely bad) to 10(extremely good). Such a subjective metric is difficult to use. Different people assessing the same criterion are likely to give different ratings. This renders a proper quality assessment almost impossible.

There are other methods to assessing quality like decomposing criterion into objectively measurable properties of the system. It is also difficult to measure every property objectively but aim is to define correctly & measure effectively.

7.2.2 Boehm Software Quality Model

Boehm introduced his quality model in 1978 [BOEH78]. He has defined three levels for quality attributes which are given in Fig. 7.11. These levels are primary uses, intermediate constructs and primitive constructs. Intermediate constructs are similar to McCall's quality factors and primitive constructs are similar to quality criteria. Hence Boehm's model is similar to McCall's in that it presents a hierarchy of characteristics, each of which contributes to overall quality. Boehm's notion of successful software includes the needs and expectations of users, as does McCall's, however, it also includes characteristics of hardware performance that are missing in McCall's model [PFLE02].

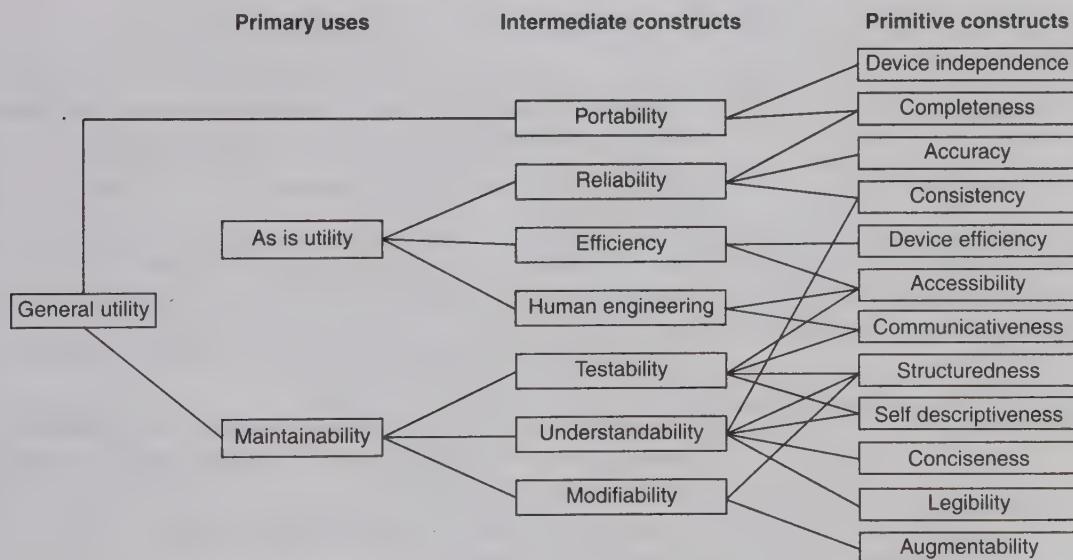


Fig. 7.11: The Boehm software quality model.

The users must find the system easy to use. Thus human engineering aspect can sometime be the most critical. A system may be very good from performance point of view, but if user cannot understand how to use it, the system is a failure.

The Boehm's model asserts that quality software is software that satisfies the needs of the users. It reflects an understanding of quality where the software [PFLE02]:

- does what the user wants it to do
- uses resources correctly and efficiently

- is easy for the user to learn & use
- is well designed, well coded, easily tested and maintained.

7.2.3 ISO 9126

The software engineering community was in the search of a single model to standardise the quality factors since 1980. The advantage of such a universal model is quite obvious; it makes easier to compare one product with another product. The result was ISO 9126 in 1992 a hierarchical model with six major attributes contributing to quality. These attributes are:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability.

The standard claims that these six factors are comprehensive; that is, any component of software quality can be described in terms of some aspect of one or more of the six factors.

Table 7.6: Software quality characteristics and attributes—The ISO 9126 view

<i>Characteristic/Attribute</i>	<i>Short Description of the Characteristics and the concerns Addressed by Attributes</i>
Functionality	Characteristics relating to achievement of the basic purpose for which the software is being engineered
• Suitability	The presence and appropriateness of a set of functions for specified tasks
• Accuracy	The provision of right or agreed results or effects
• Interoperability	Software's ability to interact with specified systems
• Security	Ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.
Reliability	Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time
• Maturity	Attributes of software that bear on the frequency of failure by faults in the software
• Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
• Recoverability	Capability and effort needed to reestablish level of performance and recover affected data after possible failure.
Usability	Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated implied set of users.

(Contd.)...

<i>Characteristic/Attribute</i>	<i>Short Description of the Characteristics and the Concerns Addressed by Attributes</i>
• Understandability	The effort required for a user to recognize the logical concept and its applicability
• Learnability	The effort required for a user to learn its application, operation, input, and output
• Operability	The ease of operation and control by users.
Efficiency	Characteristic related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions
• Time behaviour	The speed of response and processing times and throughput rates in performing its function
• Resource behaviour	The amount of resources used and the duration of such use in performing its function
Maintainability	Characteristics related to the effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functional specifications
• Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified
• Changeability	The effort needed for modification, fault removal or for environmental change
• Stability	The risk of unexpected effect of modifications
• Testability	The effort needed for validating the modified software.
Portability	Characteristics related to the ability to transfer the software from one organization or hardware or software environment to another
• Adaptability	The opportunity for its adaptation to different specified environments
• Installability	The effort needed to install the software in a specified environment
• Conformance	The extent to which it adheres to standards or conventions relating to portability
• Replaceability	The opportunity and effort of using it in the place of other software in a particular environment.

The hierarchical model is given in Fig. 7.12. The software quality attributes are explained in Table 7.6.

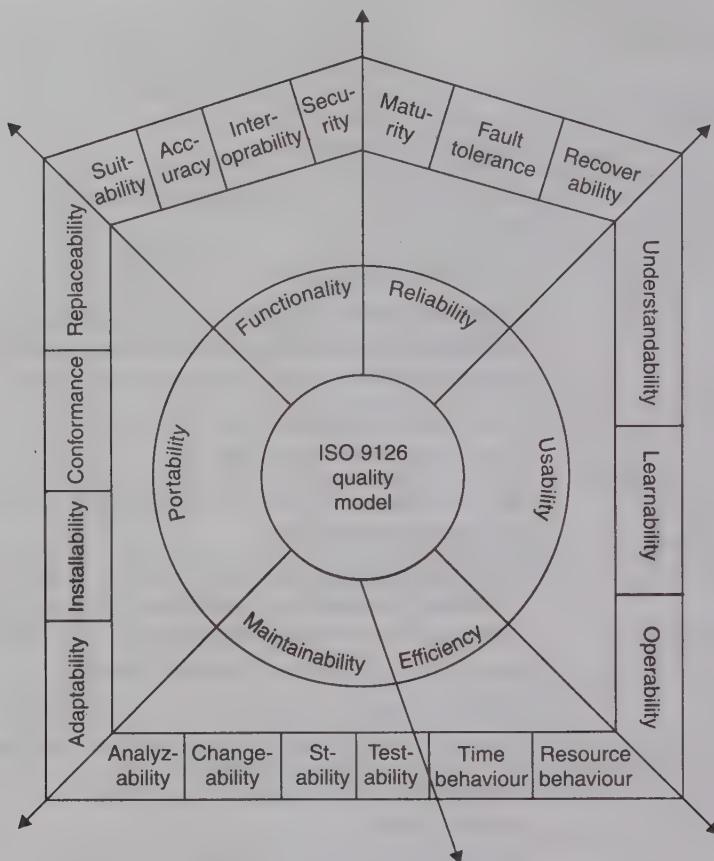


Fig. 7.12: ISO 9126 quality model

One major difference between ISO model and those of McCall and Boehm is that the ISO hierarchy is strict. Each characteristic is related only to one attribute. More-over, outer attributes are related to user view of the software rather than to an internal, developer view.

These models are helpful in articulating just what it is that we value in the software we build and use. But there are a number of difficulties in the direct application of any of the above models. We have defined quality as “meeting requirements”, and it is not possible from practical point of view to define one generic model, which can fit into all types of application domain.

These models take for granted that all high level quality attributes are independent of each other. Based on this assumption, they have decomposed higher level quality factors into lower level ones independently [NIHA01]. Thus, design of a generic model is not easy due to various attributes of quality. Sometimes, even, quality is difficult to express, although easy to feel & experience. These problems also make it difficult for us to determine how much a given model is effective & complete.

7.3 SOFTWARE RELIABILITY MODELS

To model software reliability one must first consider the principal factors that affect it: fault introduction, fault removal, and the environment. Fault introduction depends primarily on

the characteristics of the developed code (code created or modified for the application) and development process characteristics, which include software engineering technologies and tools used and level of experience of personnel. Note that code can be developed to add features or remove faults. Fault removal depends upon time, operational profile, and the quality of repair activity. The environment directly depends on the operational profile. Since some of the foregoing factors are probabilistic in nature and operate over time, software reliability models are generally formulated in terms of the random processes. The models are distinguished from each other in general terms by the nature of the variation of the random process with time.

A software reliability model specifies the general form of the dependence of the failure process on the factors mentioned. We have assumed that it is, by definition, time based (this is not to say that non-time-based models may not provide useful insights). The possibilities for different mathematical forms to describe the failure process are almost limitless.

As with any emerging discipline, software reliability has produced its share of models. Software reliability models are propounded to assess reliability of software from either specified parameters which are assumed to be known or from software-error generated data.

The past few years have seen the introduction of a number of different software reliability models. These models have been developed in response to the urgent need of software engineers, system engineers, and managers to quantify the concept of software reliability. In the reliability models, our emphasis is on failures rather than faults. The failures occur during execution of the program. Hence notion of time plays an important role. As we know, reliability is the probability that the program will not fail during a certain period of time under stated conditions.

Normally, we deal with calendar time. We may like to know the probability that a given system will not fail in one week time period. But models are generally based on execution time. The execution time is the time spent by the machine actually executing the program. Reliability models based on execution time yield better results than those based on calendar time. In many cases, a posterior translation of execution time to calendar time is possible. To emphasize this distinction, execution time will be denoted by τ and calendar time by t .

When the software fails, we try to locate and repair the fault that caused this failure. In particular, this situation arises during the testing phase of the life cycle. Most of the reliability models are applicable for system testing, when the individual modules have been integrated into one system.

During system testing, failure behaviour may not follow a constant pattern and may change over time, since faults detected are subsequently repaired. As we know, a stochastic process whose probability distribution changes over time is called non-homogeneous [VLIE02]. The variation in time between successive failures can be described in terms of a function $\mu(\tau)$ which denotes the average number of failures upto time τ . Alternatively, we may define:

$$\lambda(\tau) = \text{failure intensity function (Average number of failures per unit of time at time } \tau)$$

The reliability of a program increases through fault correction and hence the failure intensity decreases.

7.3.1 Basic Execution Time Model

The model was developed by J.D. MUSA [MUSA79] in 1979 and is based on execution time. It is assumed that failures may occur according to a non-homogeneous Poisson Process (NHPP).

Real world events may be described using Poisson processes. Examples of Poisson processes are:

- number of telephone calls expected in a given period of time
- expected number of road accidents in a given period of time
- expected number of persons visiting in a shopping mall in a given period of time.

In this case processes are non-homogeneous, since failure intensity changes as a function of time.

In this model, the decrease in failure intensity, as a function of the number of failures observed, is constant and is given as:

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{V_0}\right) \quad (7.1)$$

where λ_0 : Initial failure intensity at the start of execution.

V_0 : Number of failures experienced, if program is executed for infinite time period.

μ : Average or expected number of failures experienced at a given point in time

The relationship between failure intensity (λ) and mean failures experienced (μ) is given in Fig. 7.13.

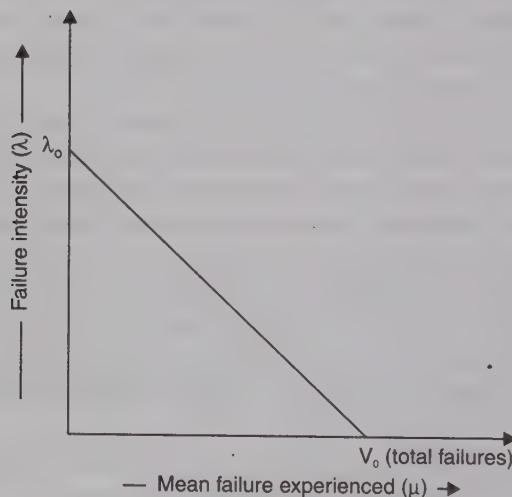


Fig. 7.13: Failure intensity λ as a function of μ for basic model

The slope of the failure intensity can be obtained by finding its first derivative and is given as:

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} \quad (7.2)$$

This model implies a uniform operational profile. If all input classes are selected equally often, the various faults have an equal probability of manifesting themselves. The correction of any of those faults then contributes an equal decrease in the failure intensity. The negative sign shows that there is a negative slope meaning thereby a decrementing trend in failure intensity.

The relationship between execution time (τ) and mean failures experienced (μ) is shown in Fig. 7.14.

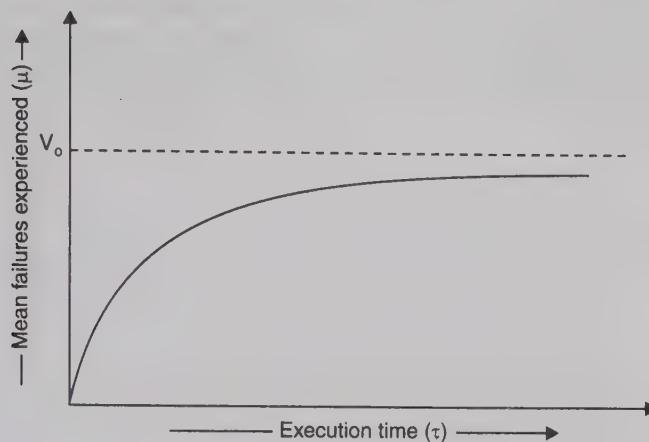


Fig. 7.14: Relationship between τ & μ for basic model

For a derivation of this relationship, equation 7.1 can be written as:

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 \left(1 - \frac{\mu(\tau)}{V_0} \right)$$

The above equation can be solved for $\mu(\tau)$ and results in:

$$\mu(\tau) = V_0 \left(1 - \exp \left(\frac{-\lambda_0 \tau}{V_0} \right) \right) \quad (7.3)$$

The failure intensity as a function of execution time is shown in Fig. 7.15.

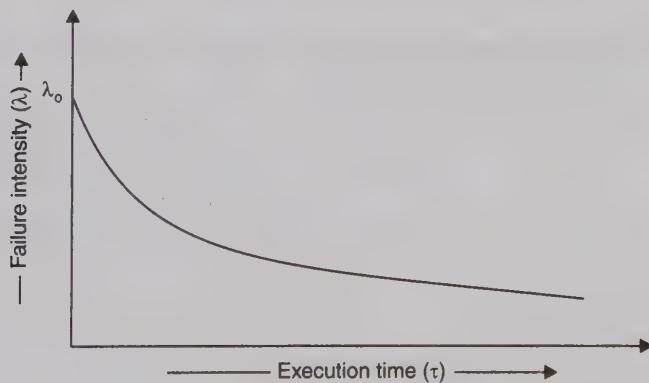


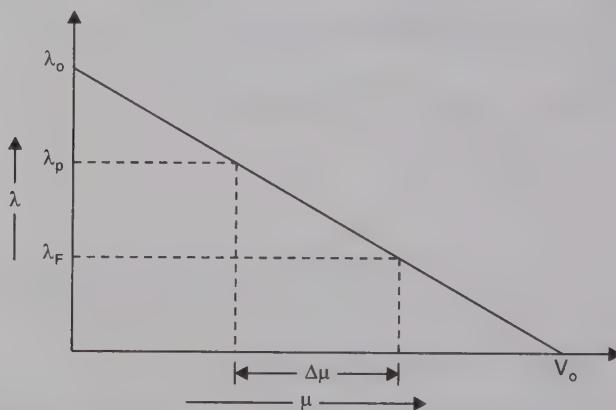
Fig. 7.15: Failure intensity versus execution time for basic model

The relationship is useful for determining the present failure intensity at any given value of execution time. This relationship can just be derived by differentiating equation 7.3 and results in:

$$\lambda(\tau) = \lambda_0 \exp \left(\frac{-\lambda_0 \tau}{V_0} \right) \quad (7.4)$$

Derived quantities

Assume that we have chosen a failure intensity objective for the software product. Assume, some failures have been corrected through removing the associated faults. Then, we can use the objective and present failure intensity to determine the additional failures that must be experienced to reach that objective. The process is explained in Fig. 7.16.



λ_0 : Initial failure intensity

λ_p : Present failure intensity

λ_F : Failure intensity objective

$\Delta\mu$: Expected number of additional failures to be experienced to reach failure intensity objective.

Fig. 7.16: Additional failures required to be experienced to reach the objective

$\Delta\mu$ can be derived in mathematical form as

$$\Delta\mu = \frac{V_0}{\lambda_0} (\lambda_p - \lambda_F) \quad (7.5)$$

Similarly, we can determine the additional execution time $\Delta\tau$ required to reach the failure intensity objective. This is shown in Fig. 7.17.

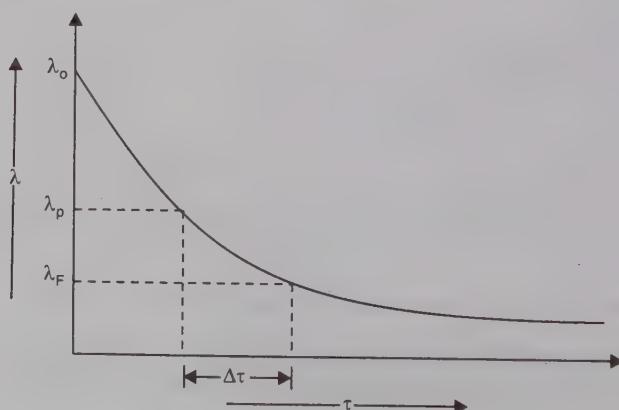


Fig. 7.17: Addition time required to reach the objective

This can be derived in mathematical form as:

$$\Delta\tau = \frac{V_0}{\lambda_0} \ln\left(\frac{\lambda_P}{\lambda_F}\right) \quad (7.6)$$

Hence, $\Delta\mu$ & $\Delta\tau$ give us the idea about failure correction workload. Both are used in making estimates of the additional calendar time required to reach the failure intensity objective.

Example 7.1

Assume that a program will experience 200 failures in infinite time. It has now experienced 100. The initial failure intensity was 20 failures/CPU hr.

- (i) Determine the current failure intensity.
- (ii) Find the decrement of failure intensity per failure.
- (iii) Calculate the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.
- (iv) Compute additional failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr.

Use the basic execution time model for the above mentioned calculations.

Solution

Here $V_0 = 200$ failures
 $\mu = 100$ failures
 $\lambda_0 = 20$ failures/CPU hr.

- (i) Current failure intensity:

$$\begin{aligned}\lambda(\mu) &= \lambda_0 \left(1 - \frac{\mu}{V_0}\right) \\ &= 20 \left(1 - \frac{100}{200}\right) = 20(1 - 0.5) = 10 \text{ failures/CPU hr.}\end{aligned}$$

- (ii) Decrement of failure intensity per failure can be calculated as:

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} = -\frac{20}{200} = -0.1/\text{CPU hr.}$$

- (iii) (a) Failures experienced & failure intensity after 20 CPU hr:

$$\begin{aligned}\mu(\tau) &= V_0 \left(1 - \exp\left(-\frac{\lambda_0 \tau}{V_0}\right)\right) \\ &= 200 \left(1 - \exp\left(-\frac{20 \times 20}{200}\right)\right) = 200(1 - \exp(-2)) \\ &= 200(1 - 0.1353) \approx 173 \text{ failures.}\end{aligned}$$

$$\begin{aligned}\lambda(\tau) &= \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right) \\ &= 20 \exp\left(\frac{-20 \times 20}{200}\right) = 20 \exp(-2) = 2.71 \text{ failures/CPU hr.}\end{aligned}$$

(b) Failures experienced & failure intensity after 100 CPU hr:

$$\begin{aligned}\mu(\tau) &= V_0 \left(1 - \exp \left(\frac{-\lambda_0 \tau}{V_0} \right) \right) \\ &= 200 \left(1 - \exp \left(\frac{-20 \times 100}{200} \right) \right) \approx 200 \text{ failures (almost)} \\ \lambda(\tau) &= \lambda_0 \exp \left(\frac{-\lambda_0 \tau}{V_0} \right) \\ &= 20 \exp \left(\frac{-20 \times 100}{200} \right) = 0.000908 \text{ failures/CPU hr.}\end{aligned}$$

(iv) Additional failures ($\Delta\mu$) required to reach the failure intensity objective of 5 failures/CPU hr.

$$\Delta\mu = \left(\frac{V_0}{\lambda_0} \right) (\lambda_P - \lambda_F) = \left(\frac{200}{20} \right) (10 - 5) = 50 \text{ failures}$$

Additional execution time required to reach failure intensity objective of 5 failures/CPU hr.

$$\begin{aligned}\Delta\tau &= \left(\frac{V_0}{\lambda_0} \right) \ln \left(\frac{\lambda_P}{\lambda_F} \right) \\ &= \frac{200}{20} \ln \left(\frac{10}{5} \right) = 6.93 \text{ CPU hr.}\end{aligned}$$

7.3.2 Logarithmic Poisson Execution Time Model

This model is also developed by Musa et. al. [MUSA79]. The failure intensity function is different here as compared to Basic model. In this case, failure intensity function (decrement per failure) decreases exponentially whereas it is constant for basic model.

The failure intensity function is given as:

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu) \quad (7.7)$$

where θ is called the failure intensity decay parameter. The relationship between failure intensity (λ) and mean failures experienced (μ) is shown in Fig. 7.18.

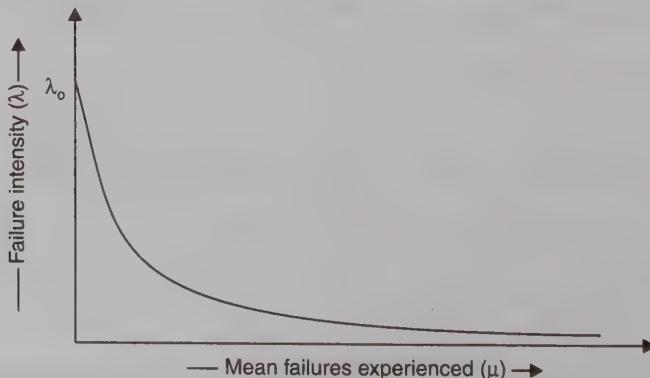


Fig. 7.18: Relationship between μ & λ

The 'θ' represents the relative change of failure intensity per failure experienced. The slope of failure intensity function is:

$$\begin{aligned}\frac{d\lambda}{d\mu} &= -\lambda_0 \theta \exp(-\mu\theta) \\ \frac{d\lambda}{d\mu} &= -\theta\lambda\end{aligned}\quad (7.8)$$

The relationship between execution time and mean failures experienced is given in Fig. 7.19.

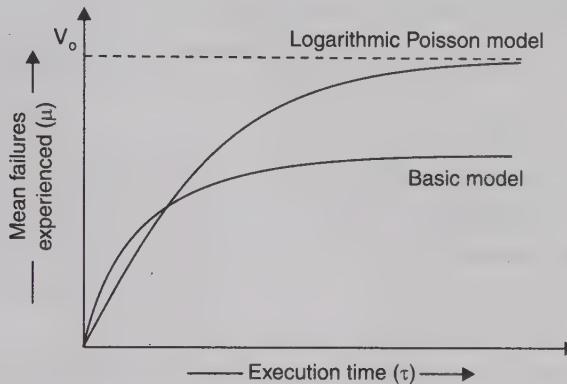


Fig. 7.19: Relationship between τ & μ

The expected number of failures for this model is always infinite at infinite time. The relation for number of failures is given by:

$$\mu(\tau) = \frac{1}{\theta} \ln(\lambda_0 \theta \tau + 1) \quad (7.9)$$

In we observe Fig. 7.15 and 7.18, it is clear that the failure intensity of the logarithmic Poisson execution time model drops more rapidly initially than that of the basic model and later it drops more slowly.

The expression for failure intensity is given as:

$$\lambda(\tau) = \lambda_0 / (\lambda_0 \theta \tau + 1) \quad (7.10)$$

The relations for the additional number of failures and additional execution time in this model are:

$$\Delta\mu = \frac{1}{\theta} \ln \left(\frac{\lambda_P}{\lambda_F} \right) \quad (7.11)$$

$$\text{and } \Delta\tau = \frac{1}{\theta} \left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P} \right] \quad (7.12)$$

where λ_P = Present failure intensity

λ_F : Failure intensity objective

Hence, at larger values of execution time, the logarithmic Poisson model will have larger values of failure intensity than the basic model.

Example 7.2

Assume that the initial failure intensity is 20 failures/CPU hr. The failure intensity decay parameter is 0.02/failures. We have experienced 100 failures up to this time.

- (i) Determine the current failure intensity.
- (ii) Calculate the decrement of failure intensity per failure.
- (iii) Find the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.
- (iv) Compute the additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.

Use Logarithmic Poisson execution time model for the above mentioned calculations.

Solution

$$\lambda_0 = 20 \text{ failures/CPU hr.}$$

$$\mu = 100 \text{ failures}$$

$$\theta = 0.02/\text{failures}$$

- (i) Current failure intensity:

$$\begin{aligned}\lambda(\mu) &= \lambda_0 \exp(-\theta\mu) \\ &= 20 \exp(-0.02 \times 100) \\ &= 2.7 \text{ failures/CPU hr.}\end{aligned}$$

- (ii) Decrement of failure intensity per failure can be calculated as:

$$\begin{aligned}\frac{d\lambda}{d\mu} &= -\theta\lambda \\ &= -.02 \times 2.7 = -.054/\text{CPU hr.}\end{aligned}$$

- (iii) (a) Failures experienced & failure intensity after 20 CPU hr:

$$\begin{aligned}\mu(\tau) &= \frac{1}{\theta} \ln(\lambda_0\theta\tau + 1) \\ &= \frac{1}{0.02} \ln(20 \times 0.02 \times 20 + 1) = 109 \text{ failures} \\ \lambda(\tau) &= \lambda_0/(\lambda_0\theta\tau + 1) \\ &= (20)/(20 \times .02 \times 20 + 1) = 2.22 \text{ failures/CPU hr.}\end{aligned}$$

- (b) Failures experienced & failure intensity after 100 CPU hr:

$$\begin{aligned}\mu(\tau) &= \frac{1}{\theta} \ln(\lambda_0\theta\tau + 1) \\ &= \frac{1}{0.02} \ln(20 \times .02 \times 100 + 1) = 186 \text{ failures} \\ \lambda(\tau) &= \lambda_0/(\lambda_0\theta\tau + 1) \\ &= (20)/(20 \times .02 \times 100 + 1) = 0.4878 \text{ failures/CPU hr.}\end{aligned}$$

- (iv) Additional failures ($\Delta\mu$) required to reach the failure intensity objective of 02 failures/CPU hr.

$$\Delta\mu = \frac{1}{\theta} \ln \frac{\lambda_P}{\lambda_F} = \frac{1}{0.02} \ln \left(\frac{2.7}{2} \right) = 15 \text{ failures.}$$

$$\Delta\tau = \frac{1}{\theta} \left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P} \right] = \frac{1}{0.02} \left[\frac{1}{2} - \frac{1}{2.7} \right] = 6.5 \text{ CPU hr.}$$

Example 7.3

The following parameters for basic and logarithmic Poisson models are given:

<i>Basic execution time model</i>	<i>Logarithmic Poisson execution time model</i>
$\lambda_0 = 10 \text{ failures/CPU hr.}$	$\lambda_0 = 30 \text{ failures/CPU hr.}$
$V_0 = 100 \text{ failures}$	$\theta = 0.025/\text{failure}$

- (a) Determine the additional failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr for both models.
- (b) Repeat this for an objective function of 0.5 failure/CPU hr. Assume that we start with the initial failure intensity only.

Solution

- (a) (i) **Basic execution time model**

$$\begin{aligned}\Delta\mu &= \frac{V_0}{\lambda_0} (\lambda_P - \lambda_F) \\ &= \frac{100}{10} (10 - 5) = 50 \text{ failures.}\end{aligned}$$

λ_P (Present failure intensity) in this case is same as λ_0 (initial failure intensity).

$$\begin{aligned}\text{Now, } \Delta\tau &= \frac{V_0}{\lambda_0} \ln \left(\frac{\lambda_P}{\lambda_F} \right) \\ &= \frac{100}{10} \ln \left(\frac{10}{5} \right) = 6.93 \text{ CPU hr.}\end{aligned}$$

- (ii) **Logarithmic execution time model**

$$\begin{aligned}\Delta\mu &= \frac{1}{\theta} \ln \left(\frac{\lambda_P}{\lambda_F} \right) \\ &= \frac{1}{0.025} \ln \left(\frac{30}{5} \right) = 71.67 \text{ failures} \\ \Delta\tau &= \frac{1}{\theta} \left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P} \right) \\ &= \frac{1}{0.025} \left(\frac{1}{5} - \frac{1}{30} \right) = 6.66 \text{ CPU hr.}\end{aligned}$$

Logarithmic model has calculated more failures in almost some duration of execution time initially.

(b) Failure intensity objective (λ_F) = 0.5 failures/CPU hr.

(i) **Basic execution time model**

$$\begin{aligned}\Delta\mu &= \frac{V_0}{\lambda_0} (\lambda_P - \lambda_F) \\ &= \frac{100}{10} (10 - 0.5) = 95 \text{ failures}\end{aligned}$$

$$\begin{aligned}\Delta\tau &= \frac{V_0}{\lambda_0} \ln\left(\frac{\lambda_P}{\lambda_F}\right) \\ &= \frac{100}{10} \ln\left(\frac{10}{0.5}\right) = 30 \text{ CPU hr.}\end{aligned}$$

(ii) **Logarithmic execution time model**

$$\begin{aligned}\Delta\mu &= \frac{1}{\theta} \ln\left(\frac{\lambda_P}{\lambda_F}\right) \\ &= \frac{1}{0.025} \ln\left(\frac{30}{0.5}\right) = 164 \text{ failures} \\ \Delta\tau &= \frac{1}{\theta} \left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P} \right) \\ &= \frac{1}{0.025} \left(\frac{1}{0.5} - \frac{1}{30} \right) = 78.66 \text{ CPU hr.}\end{aligned}$$

Note that as failure intensity objectives get smaller, the $\Delta\mu$ and $\Delta\tau$ required to reach the objectives become substantially larger for logarithmic model than the basic model.

7.3.3 Calendar Time Component

The calendar time component relates execution time and calendar time by determining the calendar time to execution time ratio at any given point in time. The ratio is based on the constraints that are involved in applying resources to a project. To obtain calendar time, one integrates this ratio with respect to execution time. The calendar time component is of greatest significance during phases where the software is being tested and repaired. During this period one can predict the dates at which various failure intensity objectives will be met. The calendar time component exists during periods in which repair is not occurring and failure intensity is constant. However, it reduces in that case to a constant ratio between calendar time and execution time.

In test, the rate of testing at any time is constrained by the failure identification or test team personnel, the failure correction or debugging personnel, or the computer time available. The quantities of these resources available to a project are usually more or less established in its early stages. Increases are generally not feasible during the system test phase because of the long lead times required for training and computer procurement. At any given value of execution time, one of these resources will be limiting. The limiting resource will determine the rate at which execution time can be spent per unit calendar time. A test phase may consist of from one to three periods, each characterized by a different limiting resource.

The following is a common scenario. At the start of testing one identifies a large number of failures separated by short time intervals. Testing must be stopped from time to time to let the people who are fixing the faults keep up with the load. As testing progresses, the intervals between failures become longer and longer. The time of the failure correction personnel is no longer completely filled with failure correction work. The test team becomes the bottleneck. The effort required to run tests and analyse the results is occupying all their time. That paces the amount of testing done each day. Finally, at even longer intervals, the capacity of the computing facilities becomes limiting. This resource then determines how much testing is accomplished.

The calendar time component is based on a debugging process model. This model takes into account:

1. resources used in operating the program for a given execution time and processing an associated quantity of failures.
2. resource quantities available, and
3. the degree to which a resource can be utilised (due to bottlenecks) during the period in which it is limiting.

Table 7.7 will help in visualising these different aspects of the resources, and the parameters that result.

Resource usage

Resource usage is linearly proportional to execution time and mean failures experienced. Let x_r be the usage of resource r . Then

$$x_r = \theta_r \tau + \mu_r \mu \quad (7.13)$$

Note that θ_r is the resource usage per CPU hr. It is nonzero for failure identification personnel (θ_I) and computer time (θ_c). The quantity μ_r is the resource usage per failure. Be careful not to confuse it with mean failures experienced μ . It is nonzero for failure identification personnel (μ_I), failure correction personnel (μ_f), and computer time (μ_c).

Table 7.7: Calendar time component resources and parameters

Resource	Usage parameters requirements per		Planned parameters	
	CPU hr	Failure	Quantities available	Utilisation
Failure identification personnel	θ_I	μ_I	P_I	1
Failure correction personnel	0	μ_f	P_f	ρ_f
Computer time	θ_c	μ_c	P_c	ρ_c

Hence, to be more precise, we have

$$\begin{aligned} X_C &= \mu_c \Delta\mu + \theta_c \Delta\tau && \text{(for computer time)} \\ X_f &= \mu_f \Delta\mu && \text{(for failure correction)} \\ X_I &= \mu_I \Delta\mu + \theta_I \Delta\tau && \text{(for failure identification)} \end{aligned}$$

For failure correction (unlike identification), resources required are dependent only on the mean failures experienced. However, computer time is used in both identification and correction of failures. Hence, computer time used will usually depend on both the amount of execution time and the number of failures.

Note that since failures experienced is a function of execution time, resource usage is actually a function of execution time only. The intermediate step of thinking in terms of failures experienced and execution time is useful in gaining physical insight into what is happening.

Computer time required per unit execution time will normally be greater than 1. In addition to the execution time for the program under test, additional time will be required for the execution of such support programs as test drivers, recording routines, and data reduction packages.

Consider the change in resource usage per unit of execution time. It can be obtained by differentiating the equation 7.13 with respect to execution time.

We obtain

$$\frac{dx_r}{d\tau} = \theta_r + \mu_r \lambda \quad (7.14)$$

Since the failure intensity decreases with testing, the effort used per hour of execution time tends to decrease with testing. It approaches the execution time coefficient of resource usage asymptotically as execution time increases.

Calendar time to execution time relationship

Resource quantities and utilizations are assumed to be constant for the period over which the model is being applied. This is a reasonable assumption, as increases are usually not feasible [BROO75].

The instantaneous ratio of calendar time to execution time can be obtained by dividing the resource usage rate of the limiting resource by the constant quantity of resources available that can be utilized. Let t be calendar time. Then

$$\frac{dt}{d\tau} = (1/P_r \rho_r) \frac{dx_r}{d\tau} \quad (7.15)$$

The quantity P_r represents resource available. Note that ρ_r is the utilization. The above ratio must be computed separately for each resource-limited period. Since x_r is a function of τ , we now have a relationship between t and τ in each resource limited period.

The form of the instantaneous calendar time to execution time ratio for any given limiting resource and either model is shown in Fig. 7.20. It is readily obtained from Equations (7.14) and (7.15) as

$$\frac{dt}{d\tau} = (\theta_r + \mu_r \lambda) / P_r \rho_r \quad (7.16)$$

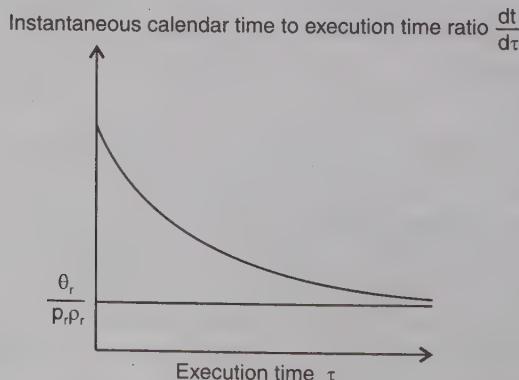


Fig. 7.20: Instantaneous calendar time to execution time ratio

The shape of this curve will parallel that of the failure intensity. The curve approaches an asymptote of $(\theta_r/P_r\rho_r)$. Note that the asymptote is 0 for the failure correction personnel resource. At any given time, the maximum of the ratios for the three limiting resources actually determines the rate at which calendar time is expanded; this is illustrated in Fig. 7.21. The maximum is plotted as a solid curve. When the curve for a resource is not maximum (not limiting), it is plotted thin. Note the transaction points FI and IC. Here, the calendar time to execution time ratios of two resources are equal and the limiting resource changes. The point FC is a potential but not true transition point. Neither resource F nor resource C is limiting near this point.

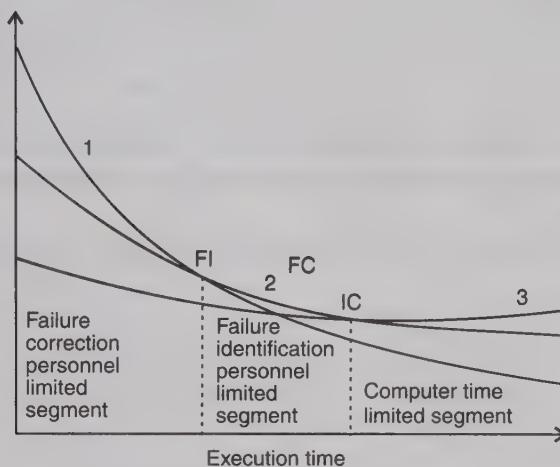


Fig. 7.21: Calendar time to execution time ratio for different limiting resources.

The calendar time component allows you to estimate the calendar time in days required to meet the failure intensity objective. The value of this interval is particularly useful to software managers and engineers. One may determine it from the additional execution time and additional number of failures needed to meet the objective that we found for the execution time component. Second, one now determines the date on which the failure intensity objective will be achieved. This is a simple variant of the first quantity that takes account of things like weekends and holidays. However, it is useful quantity because it speaks in terms managers and engineers understand.

Example 7.4

A team runs test cases for 10 CPU hrs and identifies 25 failures. The effort required per hour of execution time is 5 person hr. Each failure requires 2 hr. on an average to verify and determine its nature. Calculate the failure identification effort required.

Solution

As we know, resource usage is:

$$X_r = \theta_r \tau + \mu_r \mu$$

$$\text{Here } \theta_r = 15 \text{ person hr.,} \quad \mu = 25 \text{ failures}$$

$$\tau = 10 \text{ CPU hrs.,} \quad \mu_r = 2 \text{ hrs./failure}$$

Hence,

$$\begin{aligned} X_r &= 5(10) + 2(25) \\ &= 50 + 50 = 100 \text{ person hr.} \end{aligned}$$

Example 7.5

Initial failure intensity (λ_0) for a given software is 20 failures/CPU hr. The failure intensity objective (λ_F) of 1 failure/CPU hr. is to be achieved. Assume the following resource usage parameters.

Resource Usage	Per hour	Per failure
Failure identification effort	2 Person hr.	1 Person hr.
Failure Correction effort	0	5 person hr.
Computer time	1.5 CPU hr.	1 CPU hr.

- (a) What resources must be expended to achieve the reliability improvement ? Use the logarithmic Poisson execution time model with a failure intensity decay parameter of 0.025/failure.
- (b) If the failure intensity objective is cut to half, what is the effect on requirement of resources ?

Solution

$$\begin{aligned} (a) \quad \Delta\mu &= \frac{1}{\theta} \ln \left(\frac{\lambda_P}{\lambda_F} \right) \\ &= \frac{1}{0.025} \ln \left(\frac{20}{1} \right) = 119 \text{ failures.} \\ \Delta\tau &= \frac{1}{\theta} \left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P} \right) \\ &= \frac{1}{0.025} \left(\frac{1}{1} - \frac{1}{20} \right) = \frac{1}{0.025} (1 - 0.05) = 38 \text{ CPU hrs.} \end{aligned}$$

Hence $X_I = \mu_I \Delta\mu + \theta_I \Delta\tau$
 $= 1(119) + 2(38) = 195 \text{ Person hrs.}$

$$\begin{aligned} X_F &= \mu_F \Delta\mu \\ &= 5(119) = 595 \text{ person hr.} \end{aligned}$$

$$\begin{aligned} X_C &= \mu_c \Delta\mu + \theta_c \Delta\tau \\ &= 1(119) + (1.5)(38) = 176 \text{ CPU hr.} \end{aligned}$$

(b) Now $\lambda_F = 0.5 \text{ failures/CPU hr.}$

$$\Delta\mu = \frac{1}{0.025} \ln \left(\frac{20}{0.5} \right) \approx 148 \text{ failures}$$

$$\Delta\tau = \frac{1}{0.025} \left(\frac{1}{0.5} - \frac{1}{20} \right) = 78 \text{ CPU hrs.}$$

So,

$$X_I = 1(148) + 2(78) = 304 \text{ Person hrs.}$$

$$X_F = 5(148) = 740 \text{ person hrs.}$$

$$X_C = 1(148) + (1.5)(78) = 265 \text{ CPU hrs.}$$

Hence, if we cut failure intensity objective to half, resources requirements are not doubled but they are some what less. Note that $\Delta\tau$ is approximately doubled but increases logarithmically. Thus, the resources increase will be between a logarithmic increase and a linear increase for changes in failure intensity objective.

Example 7.6

A program is expected to have 500 faults. It is also assumed that one fault may lead to one failure only. The initial failure intensity was 2 failures/CPU hr. The program was to be released with a failure intensity objective of 5 failures/100 CPU hr. Calculate the number of failures experienced before release.

Solution

The number of failures experienced during testing can be calculated using the equation mentioned below:

$$\Delta\mu = \frac{V_0}{\lambda_0} (\lambda_P - \lambda_F)$$

Here $V_0 = 500$ because one fault leads to one failure.
 $\lambda_0 = 2$ failures/CPU hr.
 $\lambda_F = 5$ failures/100 CPU hr.
 $= 0.05$ failures/CPU hr.

So $\Delta\mu = \frac{500}{2} (2 - 0.05)$
 $= 487$ failures

Hence 13 faults are expected to remain at the release instant of the software.

7.3.4 The Jelinski-Moranda Model

The Jelinski-Moranda model [JELI72] is the earliest and probably the best-known reliability model. It proposed a failure intensity function in the form of

$$\lambda(t) = \phi(N - i + 1) \quad (7.17)$$

where

ϕ = Constant of proportionality

N = Total number of errors present

i = number of errors found by time interval t_i .

This model assumes that all failures have the same failure rate. It means that failure rate is a step function and there will be an improvement in reliability after fixing a fault. So, every failure contributes equally to the overall reliability.

Here, failure intensity is directly proportional to the number of remaining errors in a program. The relation between time & failure intensity is shown in Fig. 7.22.

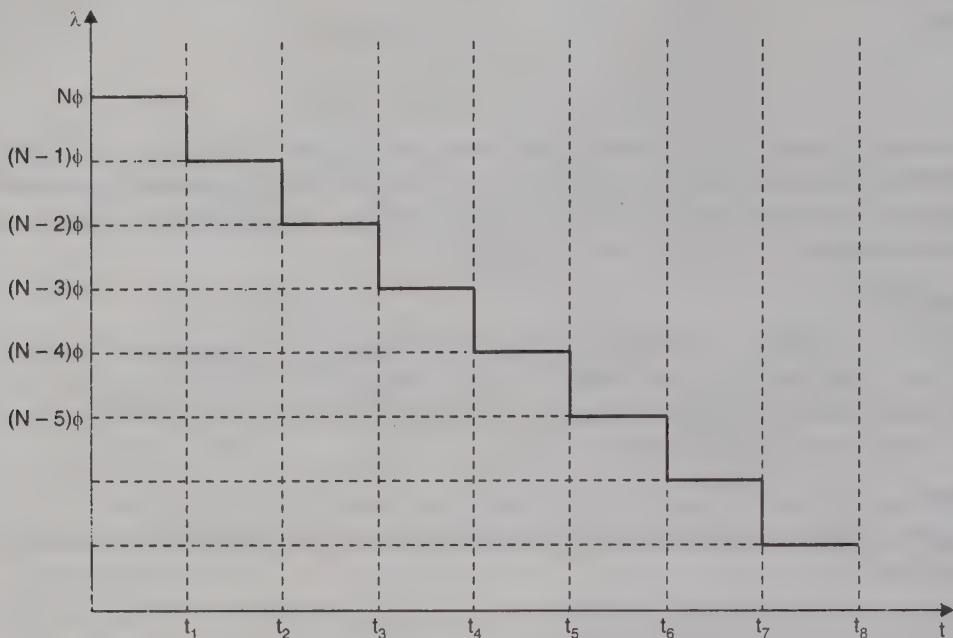


Fig. 7.22: Relation between t & λ

The time interval $t_1, t_2 \dots t_k$ may vary in duration depending upon the occurrence of a failure.

Between $(i - 1)$ th and i th failure, failure intensity function is $(N - i + 1)\phi$.

Example 7.7

There are 100 errors estimated to be present in a program. We have experienced 60 errors. Use Jelinski-Moranda model to calculate failure intensity with a given value of $\phi = 0.03$. What will be failure intensity after the experience of 80 errors ?

Solution

$$N = 100 \text{ errors}$$

$$i = 60 \text{ failures}$$

$$\phi = 0.03$$

We know

$$\begin{aligned}\lambda(t) &= \phi(N - i + 1) \\ &= 0.03(100 - 60 + 1) \\ &= 1.23 \text{ failures/CPU hr.}\end{aligned}$$

After 80 failures

$$\begin{aligned}\lambda(t) &= 0.03(100 - 80 + 1) \\ &= 0.63 \text{ failures/CPU hr.}\end{aligned}$$

Hence, there is continuous decrease in the failure intensity as the number of failures experienced increases.

7.3.5 The Bug Seeding Model

The so-called bug seeding model is an outgrowth of a technique used to estimate the number of animals in a wild life population or fish in a pond [FELL57]. The technique is best illustrated by discussing the estimation of the number of a specific species of fish, say, bass, N , in a small pond which contains no other type of fish. We begin by procuring a suitable number of bass, N_t , from a fish hatchery and tag each one with a means of identification which will remain reliably attached for the length of the measurement period. The N_t tagged fish are then added to the N original fish and allowed to mix and disperse. After an appropriate number of days a sample is fished from the lake (by hook or net) and separated into n_t tagged bass and n untagged bass. If we assume that there was no difference in the dispersion or ease of catching of the tagged and untagged fish, then we can set up the following equation, which equals the proportions of tagged fish in the fished sample to the original fraction seeded.

$$\frac{N_t}{N + N_t} = \frac{n_t}{n + n_t} \quad (7.18)$$

We may solve the above equation for N in terms of the known quantities N_t , n , and n_t yielding the following:

$$\hat{N} = \frac{n}{n_t} N_t \quad (7.19)$$

By direct analogy we may consider N to be the unknown number of bugs in the program at the start of debugging and $N_s = N_t$ to be the number of seeded bugs (unknown to the debugger). After γ months of debugging, the bugs which have been removed are examined by someone (other than the debugger) who has a list of the seeded bugs. This tester classifies them as n_s which come from the seeded group and n which were not seeded. Direct substitution yields

$$N = \frac{n}{n_s} N_s \quad (7.20)$$

The possibility of seeding bugs in a program and of using this technique to measure the initial bug content was first suggested by Mills [MILL72]. The results of the early experiments in this area were inconclusive for two reasons. First of all, it was difficult to make up realistic bugs. Consider the analogous fish problem of matching ages of the bass, not to mention the realistic situation where one has, say, three types of fish—bass, perch, and pickerel. Now one must also match the unknown ratios in seeding the tagged fish unless all types of fish are equally easy to catch. Also, are hatchery-bred fish as easy to catch as pond-bred fish? The second problem with the early seeding experiments was that the measuring technique of bug seeding was used to evaluate the effectiveness of coder reading (as opposed to machine testing) as a means of debugging programs. Thus, the results of the two experiments somewhat clouded a crisp evaluation of either.

A different approach was suggested by Hyman [HYMA73] which circumvented the problem of seeding bugs. He proposed that one employ two (or more) independent debuggers to work on the same program initially. Suppose that it is estimated that debugging will take 4 months and that debugger number 1 is assigned to the program for 4 months (or the duration of the job). Debugger number 2 is assigned to the job for only one or two months at the beginning of the project. The two debuggers work independently, and after a few weeks the results of their efforts are evaluated by a third analyst, who estimates the number of program bugs N .

The estimates are repeated every few weeks, and when the third analyst is satisfied that the value N is sufficiently well estimated, the results of debugger number 2's work are given to debugger number 1 and debugger number 2 is reassigned. Now, reasonable estimate of the total number of bugs in the program, and knowing the number of bugs already removed, by subtraction we obtain the number of remaining bugs. In addition, only a portion of debugger number 2's findings duplicate debugger number 1's. Thus, debugger number 1 is able to rapidly incorporate much of debugger number 2's work, thereby producing almost a step change in the number of bugs found. In most cases the benefits should far outweigh the costs—one or two extra man-months of debugging time (the cost may be minor, zero, or negative if debugger number 2 really finds many independent errors which shorten debugger number 1's efforts). We now discuss a detailed development of the estimation formulas.

In order to develop our two-debugger estimation procedure, we begin with the following notation:

γ = development time in months; interval 0 to γ_1 is the measurement period

B_0 = number of bugs in program at $\gamma = 0$

B_1 = number of bugs found in program by debugger number 1 upto time γ_1

b_0 = number of bugs which debugger number 2 finds upto time γ_1 which are common, i.e., in set B_1

b_1 = number of bugs which programmer number 2 finds upto time γ_1 which are independent, i.e., not in set B_1

$B_2 = b_0 + b_1$ = number of bugs found in program by debugger number 2 upto time γ_1

If we really believe that the bugs we would seed in a bug seeding experiment are identical to the indigenous bugs, then instead of seeding, we could merely locate a certain number of bugs and tag them (identify them). Of course the problem here is that much work must transpire in order to identify a group of bugs. Since it should not matter which set of bugs we choose as the tagged set (as long as they are representative), we should be able to treat the identified set as if they were a tagged set of bugs. In order to proceed, we must make some fundamental assumptions, which we will state as hypotheses:

Bug Characteristics unchanged as debugging proceeds: When a large program is debugged, the bugs found during the first several weeks (months) are representative of the total bug population.

Independent debugging results in similar programs: When two independent debuggers work on a large program, the evolution of the program is such that the differences between their two versions are small enough that they can be neglected.

Common bugs versus representative: When two independent debuggers work on a large program, the bugs which they find in common are representative of the total population.

Assuming that the above hypotheses are valid, we can treat the quantity B_1 as if it were the selected group N s. Similarly, b_c becomes n_s and B_2 becomes n . Substituting these quantities solving for the original number of bugs $B_0 = N$ yields

$$\hat{B}_0 = \frac{B_2}{b_c} B_1 \quad (7.21)$$

7.4 CAPABILITY MATURITY MODEL

The capability maturity model (CMM) is not a software life cycle model. Instead, it is a strategy for improving the software process, irrespective of the actual life cycle model used. The CMM was developed by Software Engineering Institute (SEI) of Carnegie-Mellon University in 1986.

CMM is used to judge the maturity of the software processes of an organization and to identify the key practices that are required to increase the maturity of these processes. The CMM is organized into five maturity levels as shown in Fig. 7.23 [PAUL94, SCHAF96].

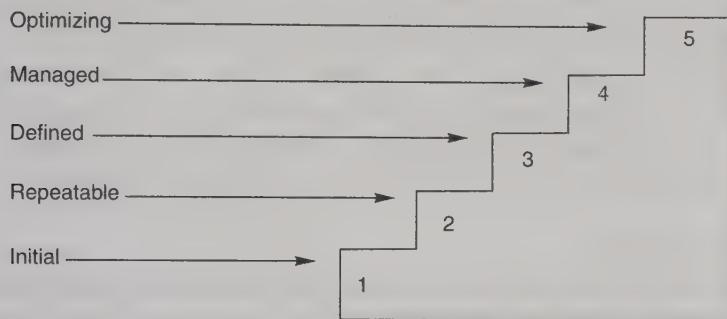


Fig. 7.23: Maturity levels of CMM.

7.4.1 Maturity Levels

1. Initial (Maturity Level 1)

At this, the lowest level, there are essentially no sound software engineering management practices in place in the organization. Instead, everything is done on an adhoc basis. If one specific project happens to be staffed by a competent manager and a good software development team, then that project may be successful. However, the usual pattern is time and cost overruns caused by a lack of sound management in general, and planning in particular. As a result, most activities are responses to crisis, rather than preplanned tasks. In maturity level 1 organizations, the software process is unpredictable, because it depends totally on the current staff; as the staff changes, so does the process. As a consequence, it is impossible to predict, with any accuracy, the important items such as the time it will take to develop a product or the cost of that product. It is unfortunate fact that the vast majority of software organizations all over the world are level 1 organizations.

2. Repeatable (Maturity Level 2)

At this level, policies for managing a software project and procedures to implement those policies are established. Planning and managing new projects is based on experience with similar projects. An objective in achieving level 2 is to institutionalize effective management processes for software projects, which allow organizations to repeat successful practices developed on earlier projects, although the specific processes implemented by the projects may differ. An effective process can be characterized as practiced, documented, enforced, trained, measured, and amenable to be proved.

Instead of functioning in crisis mode as in level 1, managers identify problems as they arise and take immediate corrective action to prevent them from becoming crisis. The key point is that, without measurement it is impossible to detect problems before they get out of hand. Measurements may include the careful tracking of costs, schedules and functionality. Also, measurements taken during one project can be used to draw up realistic duration and cost schedules for future projects.

Projects in level 2 organizations have installed basic software management controls. Realistic project commitments are based on the results observed on previous projects and on the requirements of the current projects. Software project standards are defined, and the organization ensures they are faithfully followed. The software project team works with its subcontractors, if any, to establish a strong customer-supplier relationship.

The software process capability of level 2 organizations can be summarized as disciplined because planning and tracking of the software project is stable and earlier successes can be repeated. The project's process is under the effective control of a project management system, following realistic plans based on the performance of previous projects.

3. Defined (Maturity Level 3)

At this level, the standard process for developing and maintaining software across the organization is documented, including both software engineering and management processes. This standard process is referred to throughout the CMM as the organization's standard software process. Processes established at level 3 are used (and changed, as appropriate) to help the software managers and technical staff to perform more effectively. The organization exploits effective software engineering practices while standardizing its processes. An organization wide training program is implemented to ensure that the staff and managers have the knowledge and skills required to fulfill their assigned roles.

Projects tailor the organization's standard software process, to develop their own defined software process, which accounts for the unique characteristics of the project. This tailored process is referred to, in the CMM as the project's defined software process. Defined software contains a coherent, integrated set of well-defined software engineering and management processes. Because the software process is well defined, management has good control over technical progress of all projects.

The software process capability of level 3 organizations can be summarized as "standard" and "constituent" because both software engineering and management activities are stable and repeatable. Within established product lines, cost, schedule, and functionality are under control, and software quality is tracked. This process capability is based on a common, organization-wide understanding of the activities, roles, and responsibilities in a defined software process.

4. Managed (Maturity Level 4)

At this level, the organization sets quantitative quality goals for both software products and processes. Productivity and quality are measured for important software process activities across all projects as part of an organizational measurement program. An organization-wide software process database is used to collect and analyze the data available from the project's defined software processes. Software processes are instrumented with well-defined and

consistent measurements at level 4. These measurements establish the quantitative foundation for evaluating the project's software processes and products. Projects achieve control over their products and processes by narrowing the variation in their process performance to fall within acceptable quantitative boundaries.

Meaningful variations in process performance can be distinguished from random variation (noise), particularly within established product lines. The risks involved in moving up the learning curve of a new application domain are known and carefully managed. One measure could be number of faults detected per 1000 lines of code. A corresponding objective is to reduce this quantity (number of faults) over time.

The software process capability at level 4 organizations can be summarized as "predictable" because the process is measured and operates within measurable limits. This level of process capability allows an organization to predict trends of process and product quality within quantitative bounds of these limits. When these limits are exceeded, action is taken to correct the situation. Software products are of predictably high quality.

5. Optimizing (Maturity Level 5)

At this level, the entire organization is focused on continuous process improvement. The organizations have the means to identify weaknesses and strengthen the process proactively, with the goal of preventing the occurrence of defects. Data of the effectiveness of the software process is used to perform cost benefit analysis of new technologies and proposes changes to the organization's software process. Innovations that exploit the best software engineering practices are identified and transferred throughout the organization.

Software project teams in level 5 organizations "analyze defects to determine their causes". Software processes are evaluated to prevent known types of defects from recurring, and lessons learned are disseminated to other projects.

The software process capability of level 5 organizations can be characterized as "continuously improving" because level 5 organizations are continuously striving to improve the range of their process capability, thereby improving the process performance of their projects. Improvement occurs both by incremental advancements in the existing process and by innovations using new technologies and methods.

These five maturity models are summarized in Fig. 7.24 [SCHA96]

<i>Maturity Level</i>	<i>Characterization</i>
Initial	Adhoc Process
Repeatable	Basic Project Management
Defined	Process Definition
Managed	Process Measurement
Optimizing	Process Control

Fig. 7.24: The five levels of CMM

Experience with the capability maturity model has shown that advancing a complete maturity level usually takes from 18 months to 3 years, but moving from level 1 to level 2

sometimes takes 3 or even 5 years. This is a reflection of how difficult it is to instill a methodical approach in an organization that up to now has functioned on a purely adhoc and reactive basis.

7.4.2 Key Process Areas

Except for level 1, each maturity level is decomposed into several key process areas that indicate the areas (KPAs) an organization should focus on to improve its software process. Key process areas identify the issues that must be addressed to achieve a maturity level. Each key process area identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for enhancing process capability. The key process areas and their purposes are listed below. The name of each key process area is followed by its two-letter abbreviation [PAUL94, PAUL95, WIEG98].

By definition there are no key process areas for level 1.

The key process areas at level 2 focus on the software project's concerns related to establishing basic project management controls, as summarized below:

Requirements Management (RM)	Establish a common relationship between the customer requirements and the developers in order to understand the requirements of the project.
Software Project Planning (PP)	Establish reasonable plans for performing the software engineering and for managing the software project.
Software Project Tracking and Oversight (PT)	Establish adequate visibility into actual progress so that management can take effective actions when the software project's performance deviates significantly from the software plans.
Software Subcontract Management (SM)	Select qualified software subcontractors and manage them effectively.
Software Quality Assurance (QA)	Provide management with appropriate visibility into the process being used by the software project and of the products being built.
Software Configuration Management (CM)	Establish and maintain the integrity of the products of the software project throughout the project's software life cycle.
The key process areas at level 3 address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects, as summarized below:	
Organization Process Focus (PF)	Establish the organizational responsibility for software process activities that improve the organization's overall software process capability.
Organization Process Definition (PD)	Develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for cumulative, long-term benefits to the organization.

Training Program (TP)	Develop the skills and knowledge of individuals so that they can perform their roles effectively and efficiently.
Integrated Software Management (IM)	Integrate the software engineering and management activities into a coherent, defined software process that is tailored from the organization's standard software process and related process assets.
Software Product Engineering (PE)	Consistently perform a well-defined engineering process that integrates all the software engineering activities to produce correct, consistent software products effectively and efficiently.
Inter group Coordination (IC)	Establish a means for the software engineering group to participate actively with the other engineering groups so the project is better able to satisfy the customer's needs effectively and efficiently.
Peer Reviews (PR)	Remove defects from the software work products early and efficiently. An important corollary effect is to develop a better understanding of the software work products and of the defects that can be prevented.

The key process areas of level 4 focuses on establishing a quantitative understanding of both the software process and the software work products being built, as summarized below:

Quantitative Process Management (QP)	Control the process performance of the software project quantitatively.
Software Quality Management (QM)	Develop a quantitative understanding of the quality of the project's software products and achieve specific quality goals.

The key process areas at level 5 cover the issues that both the organization and the projects must address to implement continuous and measurable software process improvement, as summarized below:

Defect Prevention (DP)	Identify the causes of defects and prevent them from recurring.
Technology Change Management (TM)	Identify beneficial new technologies (i.e., tools, methods, and processes) and transfer them into the organization in an orderly manner.
Process Change Management (PC)	Continually improve the software processes used in the organization with the intent of improving software quality, increasing productivity, and decreasing the cycle time for product development.

7.4.3 Common Features

For convenience, common features organize each of the key process areas. The common features are attributes that indicate whether the implementation and institutionalization of a key process area are effective, repeatable, and lasting. The five common features, followed by their two-letter abbreviations, are listed below:

Commitment to Perform (CO)	Describes the actions the organizations must take to ensure that the process is established and will endure. It includes practices on policy and leadership.
Ability to Perform (AB)	Describes the preconditions that must exist in the project or organization to implement the software process competently. It includes practices on resources, organizational structure, training, and tools.
Activities Performed (AC)	Describes the role and procedures necessary to implement a key process area. It includes practices on plans, procedures, work performed, tracking, and corrective action.
Measurement and Analysis (ME)	Describes the need to measure the process and analyze the measurements. It includes examples of measurements.
Verifying Implementation (VE)	Describes the steps to ensure that the activities are performed in compliance with the process that has been established. It includes practices on management reviews and audits.

There is another reason for the growing importance of the CMM. Many software organizations have stipulated that any software development organization that wishes to be a subcontractor must conform to CMM level 3 or higher. Thus, there is pressure for organizations to improve the maturity of their software processes.

7.5 ISO 9000

As explained in the previous section, the SEI capability maturity model initiative is an attempt to improve software quality by improving the process by which software is developed. The term maturity is essentially a measure of the goodness of the process itself. A different attempt to improve software quality is based on International Standards Organization (ISO) 9000-Series standards.

The standard is important, as it is becoming the main focus in which customers can judge the competence of a software developer. It has been adopted for use by over 130 countries. One of the problems with ISO- 9000 series standard is that it is not industry-specific. It is expressed in general terms, and can be interpreted by the developers to diverse products such as ball bearings, hair dryers, automobiles, televisions as well as software [INCE94].

ISO-9000 series of standards is a set of documents dealing with quality systems that can be used for quality assurance purposes. ISO-9000 series is not just software standard. It is a series of five related standards that are applicable to a wide variety of industrial activities, including design/development, production, installation, and servicing. Within the ISO 9000 Series, standard ISO 9001 for quality system is the standard that is most applicable to software development.

Because of the broadness of ISO 9001, ISO has published specific guidelines to assist in applying ISO 9001 to software namely ISO 9000-3.

There is significant room for interpretation in using ISO 9001 in the software world. ISO 9000-3 is a guide to interpret ISO 9001, yet many-to-many relationships between their clauses raises the suspicion that liberties have been taken in creating this guidance.

7.5.1 Mapping ISO 9001 to the CMM

There are 20 clauses in ISO 9001, which are summarized and compared to the practices in the CMM in this section [PAUL94].

1. Management responsibility

ISO 9001 requires that the quality policy be defined, documented, understood, implemented, and maintained; that responsibilities and authorities for all personnel specifying, achieving, and monitoring quality be defined; and that in-house verification resources be defined, trained, and funded. A designated manager ensures that the quality program is implemented and maintained.

In the CMM, management responsibility for quality policy and verification activities is primarily addressed in Software Quality Assurance, although software project planning and software project tracking and oversight also include activities that identify responsibility for performing all project roles.

2. Quality system

ISO 9001 requires that a documented quality system, including procedures and instructions, be established. ISO 9000-3 characterizes this quality system as an integrated process throughout the entire life cycle.

Quality system activities are primarily addressed in the CMM in Software Quality Assurance. The procedures that would be used are distributed throughout the key process areas in the various Activities Performed practices.

The specific procedures & standards that a software project would use are specified in the software development plan described in Software Project Planning. Compliance with these standards and procedures is assured in Software Quality Assurance and by the auditing practices in the Verifying Implementation common feature.

3. Contract review

ISO 9001 requires that contracts be reviewed to determine whether the requirements are adequately defined, agreed with the bid, and can be implemented.

Review of the customer requirements, as allocated to software, is described in the CMM in Requirements Management. The software organization (supplier) ensures that the system requirements allocated to software are documented and reviewed and that missing or ambiguous requirements are clarified. Since the CMM is constrained to the Software perspective, the customer requirements as a whole are beyond the scope of this key process area.

4. Design control

ISO 9001 requires that procedures to control and verify the design be established. This includes planning design activities, identifying inputs and outputs, verifying the design, and controlling design changes.

In the CMM, the life cycle activities of requirements analysis, design, code, and test are described in Software Product Engineering. Planning these activities is described in Software Project Planning. Software Project Tracking and Oversight describes control of these life cycle activities, and Software Configuration Management describes configuration management of software work products generated by these activities.

5. Document control

ISO 9001 requires that the distribution and modification of documents be controlled.

In the CMM, the configuration management practices characterizing document control are described in Software Configuration Management. The specific procedures, standards, and other documents that may be placed under configuration management in the CMM are distributed throughout the key process areas in the various Activities Performed practices.

6. Purchasing

ISO 9001 requires that purchased products conform to their specified requirements. This includes the assessment of potential subcontractors and verification of purchased products.

In the CMM, this is addressed in Software Subcontract Management.

7. Purchaser-supplied product

ISO 9001 requires that any purchaser-supplied material be verified and maintained.

Integrated Software Management is the only practice in the CMM describing the use of purchased software. It does so in the context of identifying off-the-shelf or reusable software as part of planning. Integration of off-the-shelf and reusable software is one of the areas where the CMM is weak.

8. Product identification and traceability

ISO 9001 requires that the product be identified and traceable during all stages of production, delivery, and installation. The CMM covers this clause primarily in Software Configuration Management.

9. Process control

ISO 9001 requires that production processes be defined and planned. This includes carrying out production under controlled conditions, according to documented instructions. Special processes that cannot be fully verified are continuously monitored and controlled.

The procedures defining the software production process in the CMM are distributed throughout the key process areas in the various Activities Performed practices (see section 7.4.3). Specific procedures and standards that would be used are specified in the software development plan.

10. Inspection and testing

ISO 9001 requires that incoming materials be inspected or certified before use and that in-process inspection and testing be performed. Final inspection and testing are performed prior to release of finished product. Records of inspection and testing are kept.

The issues surrounding the inspection of incoming material have already been discussed in clause 4.7 of ISO-9001. The CMM describes testing in Software Product Engineering. In-process inspections in the software development are addressed in Peer Reviews.

11. Inspection, measuring, and test equipment

ISO 9001 requires that equipment used to demonstrate conformance be controlled, calibrated, and maintained. When test hardware or software is used, it is checked before use and rechecked at prescribed intervals.

This clause is generically addressed in the CMM under the testing practices in Software Product Engineering.

12. Inspection and test status

ISO 9001 requires that the status of inspections and tests be maintained for items as they progress through various processing steps.

This clause is addressed in the CMM by the testing practices in Software Product Engineering and on problem reporting and configuration status, respectively, in Software Configuration Management.

13. Control of nonconforming product

ISO 9001 requires that nonconforming product be controlled to prevent inadvertent use or installation.

Design, implementation, testing, and validation are addressed in Software Product Engineering. Software Configuration Management addresses the status of configuration items, which would include the status of items that contain known defects not yet fixed. Installation is not addressed in the CMM.

14. Corrective action

ISO 9001 requires that the causes of nonconforming product be identified. Potential causes of nonconforming product are eliminated; procedures are changed resulting from corrective action.

The software development group should look at field defects, analyze why they occurred, and take corrective action. This would typically occur through software updates and patches distributed to the customers of the software. Under this interpretation, an appropriate mapping of this clause would be problem reporting, followed with controlled maintenance of baseline work products. Problem reporting is described in Software Configuration Management in the CMM.

15. Handling, storage, packaging, and delivery

ISO 9001 requires that procedures for handling, storage, packaging, and delivery be established and maintained. Replication, delivery and installation are not covered in the CMM.

16. Quality records

ISO 9001 requires that quality records be collected, maintained, and dispositioned. The practices defining the quality records to be maintained in the CMM are distributed throughout the key process areas in the various Activities Performed practices.

17. Internal quality audits

ISO 9001 requires that audits be planned and performed. The results of audits are communicated to management, and any deficiencies found are corrected.

The auditing process is described in Software Quality Assurance. Specific audits in the CMM are called out in the auditing practices of the Verifying Implementation common feature.

18. Training

ISO 9001 requires that training needs be identified and that training be provided, since selected tasks may require qualified personnel. Records of training are maintained.

Specific training needs in the CMM are identified in the training and orientation practices in the Ability to Perform common feature.

19. Servicing

ISO 9001 requires that servicing activities be performed as specified. Although the CMM intends to apply in both the software development and maintenance environments, the practices in the CMM do not directly address the unique aspects that characterize the maintenance environment. Maintenance is embedded throughout the practices of the CMM, and they must be appropriately interpreted in the development or maintenance contexts. Maintenance is not, therefore, a separate process in the CMM.

20. Statistical techniques

ISO 9001 states that appropriate, adequate statistical techniques are identified and should be used to verify the acceptability of process capability and product characteristics.

The practices describing measurement in the CMM are distributed throughout the key process areas. Product measurement is typically incorporated into the various Activities Performed practices (see section 7.4.3), and process measurement is described in the Measurement and Analysis common feature.

7.5.2 Contrasting ISO 9001 and the CMM

Clearly there is a strong correlation between ISO 9001 and the CMM, although some issues in ISO 9001 are not covered in the CMM, and some issues in the CMM are not addressed in ISO 9001.

The biggest difference, however, between these two documents is the emphasis of the CMM on continuous process improvement. ISO 9001 addresses the minimum criteria for an acceptable quality system. It should also be noted that the CMM focuses strictly on software, while ISO 9001 has a much broader scope: hardware, software, processed materials, and services [MARQ91].

The biggest similarity is that for both the CMM and ISO 9001, the bottom line is “Say what you do; do what you say.” The fundamental premise of ISO 9001 is that every important process should be documented and every deliverable should have its quality checked through a quality control activity. ISO 9001 requires documentation that contains instructions or guidance on what should be done or how it should be done. The CMM shares this emphasis on processes that are documented and practiced as documented. Phrases such as conducted

"according to a documented procedure" and following "a written organizational policy" characterize the key process areas in the CMM.

The CMM also emphasizes the need to record information for later use in the process and for improvement of the process. This is equivalent to the quality records of ISO 9001 that document whether or not the required quality is achieved and whether or not the quality system operates effectively [TICK92].

7.5.3 Conclusion

Although there are specific issues that are not adequately addressed in the CMM, in general the concerns of ISO 9001 are encompassed by the CMM. The converse is less true. ISO 9001 describes the minimum criteria for an adequate quality management system rather than process improvement, although future revisions of ISO 9001 may address this concern. The differences are sufficient to make a route mapping impractical, but the similarities provide a high degree of overlap.

Should software process improvement be based on the CMM, with perhaps some extensions for ISO 9001 specific concerns, or should the improvement effort focus on certification concerns? A market may require ISO 9001 certification, and level 1 organization would certainly profit from addressing the concerns of ISO 9001. It is also true that addressing the concerns of the CMM would help organizations prepare for an ISO 9001 audit. Although either document could be used to structure a process improvement program, the more detailed guidance and greater breadth provided to software organizations by the CMM suggest that it is the better choice (perhaps a biased answer).

In any case, building competitive advantage should be focused on improvement, not on achieving a score, whether the score is a maturity level or a certificate. We would advocate addressing the larger context encompassed by the CMM, but even then there is a need to address the still larger business context, as exemplified by Total Quality Management.

REFERENCES

- [AVIZ77] Avizienis A. & Chen L., "On the Implementation of N-version Programming for Software Fault Tolerance During Program Execution", Proceedings of COMPSAC 77, Chicago, Nov.1977.
- [BELL91] Belli F., and Jedrzejowicz P., "An Approach to the Reliability Optimization of Software with Redundancy", IEEE Trans. on Software Engineering, Vol.17, No.3, March 1991.
- [BOEH78] Boehm B.W., et. al, "Characteristics of Software Quality", Amsterdam, North Holland, 1978.
- [BROO75] Brooks, F.P., Jr., *The Mythical Man Month*, Addison Wesley, Reading, MA, 1975.
- [DUNN90] Dunn R., "Software Quality: Concepts and Plans", Prentice Hall, 1990.
- [ECKH 91] Eckhardt D.E., et al., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability", IEEE Trans. on Software Engineering , Vol. 17, No.7, July 1991.
- [FELL57] Feller, W., "An Introduction to Probability Theory and its Applications", 2nd ed., Vol.1, Wiley, New York, 1957.

- [HYMA73] Hyman, Mort: "Private Communication", 1973.
- [INCE94] Ince D., "ISO 9001 & Software Quality Assurance", McGraw Hill Book Co. 1994.
- [JELI72] Jelinski Z. & Moranda P.B., "Software Reliability Research", in Statistical Computer Performance Evaluation, Academic Press, NY, PP465-84, 1972.
- [JELI71] Jelinski Z. and Moranda P.B., "Software Reliability Research in Statistical Computer Performance Evaluation", ed. W. Freiberger, New York: Academic Press, pp.465–484, 1971.
- [JONE 86] Jones, C., Programming Productivity, McGraw-Hill, N.Y., 1986.
- [LLOY77] Lloyd, D.K., and M.Lipow, Reliability Management, Methods, and Mathematics, Redondo Beach, CA , 1977
- [MARQ91] Marquardt D., et. al., "Vision 2000: The Strategy for the ISO 9000 Series Standards in the 90s", ASQC Quality Progress, Vol. 24, No. 5, 25—31, May 1991.
- [MACA76] McCabe T.J., "A Complexity Measure", IEEE Trans on Software Engineering, Vol.2, No.6, pp.308–320, Dec.1976.
- [MILL72] Mills, Harlan D., "Mathematical Foundations of Structured Programming", IBM Federal Systems Division Document FSC72-6012, Gaithersburg, Md., February 1972.
- [MUSA75] Musa J.D., "A Theory of Software Reliability & its Applications", IEEE Trans. on Software Engg., SE 1(3), pp.312–327, Sept.1975.
- [MUSA79] Musa J.D., "Validity of the Execution Time Theory of Software Reliability" , IEEE Trans. on Reliability, R-28 (3), PP. 181–191, August 1979.
- [MUSA79] Musa J.D., "Software Reliability Data", Report Available from Data and Analysis Center for Software, Rome Air Development center, Rome, N.Y., 1979.
- [MUSA79] Musa J.D., "Validity of the Execution Time Theory for Software Reliability", IEEE Trans. on Reliability R-28(3), pp.181–191, Aug.1979
- [MUSA80] Musa J.D., "Software Reliability measurements", Journal of Systems & Software, 1(3), pp.223–241, 1980
- [MUSA87] Musa J.D., A.Iannino, K. Okumoto, "Software Reliability Measurement, Prediction & Application", McGraw Hill Book Company, NY., pp.183–185, 1987.
- [MUSA87] Musa J.D., et. al., "Engineering and Managing Software with Reliability Measures", McGraw-Hill, 1987.
- [MCCA77] McCall J.A., et. al., "Factors in Software Quality", Vol. 1, 2 and 3, AD/A-049-014/015/055, springfield, VA: National Technical Information Service, 1977.
- [PAUL94] Paulk M.C., "A Comparision of ISO 9001 and the CMM for Software", Technical Report CMU/SEI-94-TR-12, ESC-TR-94-12, SEI, Carnegie Mellon University, USA, 1994.
- [PAUL95] Paulk M.C. et. al., "The Capability Maturity Model: Guidelines for Improving the Software Process", Reading, MA, Addison-Wesley, 1995.
- [PFLE02] Pfleeger S.L., "Software Engineering", 2nd Edition, Pearson Edution Asia, 2002.
- [RAND75] Randill B., "System Structure for Software Fault Tolerance", IEEE Trans. on Software Engineering, Vol. SE-1, June 1975.
- [SCHA96] Schach S., "Classical & Object Oriented Software Engineering", IRWIN, USA, 1996.
- [VLIE02] Vliet H.V., "Software Engineering", John Wiley & Sons, 2002.
- [WIEG98] Wiegers K.E., "Molding the CMM to Your Organisation", Software Development Magazine, May, 1998.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions.

- 7.23. Failure intensity function of Logarithmic Poisson execution model is given as
(a) $\lambda(\mu) = \lambda_0 \ln(-\theta\mu)$ (b) $\lambda(\mu) = \lambda_0 \exp(\theta\mu)$
(c) $\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$ (d) $\lambda(\mu) = \lambda_0 \log(-\theta\mu)$.
- 7.24. In Logarithmic Poisson execution model, 'θ' is known as
(a) Failure intensity function parameter (b) Failure intensity decay parameter
(c) Failure intensity measurement (d) Failure intensity increment parameter.
- 7.25. In Jelinski-Moranda model, failure intensity is defined as
(a) $\lambda(t) = \phi(N - i + 1)$ (b) $\lambda(t) = \phi(N + i + 1)$
(c) $\lambda(t) = \phi(N + i - 1)$ (d) $\lambda(t) = \phi(N - i - 1)$.
- 7.26. CMM level 1 has
(a) 6 KPAs (b) 2 KPAs
(c) 0 KPAs (d) None of the above.
- 7.27. MTBF stands for
(a) Mean time between failures (b) Maximum time between failures
(c) Minimum time between failures (d) Many time between failures.
- 7.28. CMM model is a technique to
(a) Improve the software process (b) Automatically develop the software
(c) Test the software (d) All of the above.
- 7.29. Total number of maturing levels in CMM are
(a) 1 (b) 3
(c) 5 (d) 7.
- 7.30. Reliability of a software is dependent on number of errors
(a) removed (b) remaining
(c) both (a) & (b) (d) None of the above.
- 7.31. Reliability of software is usually estimated at
(a) Analysis phase (b) Design phase
(c) Coding phase (d) Testing phase.
- 7.32. CMM stands for
(a) Capacity maturity model (b) Capability maturity model
(c) Cost management model (d) Comprehensive maintenance model.
- 7.33. Which level of CMM is for basic project management ?
(a) Initial (b) Repeatable
(c) Defined (d) Managed.
- 7.34. Which level of CMM is for process control ?
(a) Initial (b) Repeatable
(c) Defined (d) Optimizing.
- 7.35. Which level of CMM is for process management ?
(a) Initial (b) Defined
(c) Managed (d) Optimizing.
- 7.36. CMM was developed at
(a) Harvard University (b) Cambridge University
(c) Carnegie Mellon University (d) Maryland University.

EXERCISES

- 7.1. What is software reliability? Does it exist?
- 7.2. Explain the significance of bath tube curve of reliability with the help of a diagram.
- 7.3. Compare hardware reliability with software reliability.
- 7.4. What is software failure? How is it related with a fault?
- 7.5. Discuss the various ways of characterising failure occurrences with respect to time.
- 7.6. Describe the following terms:
 - (i) Operational profile
 - (ii) Input space
 - (iii) MTBF
 - (iv) MTTF
 - (v) Failure intensity.
- 7.7. What are uses of reliability studies? How can one use software reliability measures to monitor the operational performance of software?
- 7.8. What is software quality? Discuss software quality attributes.
- 7.9. What do you mean by software quality standards? Illustrate their essence as well as benefits.
- 7.10. Describe the McCall software quality model. How many product quality factors are defined and why?
- 7.11. Discuss the relationship between quality factors and quality criteria in McCall's software quality model.
- 7.12. Explain the Boehm software quality model with the help of a block diagram.
- 7.13. What is ISO9126? What are the quality characteristics and attributes?
- 7.14. Compare the ISO9126 with McCall software quality model and highlight few advantages of ISO9126.
- 7.15. Discuss the basic model of software reliability. How can $\Delta\mu$ and $\Delta\tau$ be calculated?
- 7.16. Assume that the initial failure intensity is 6 failures/CPU hr. The failure intensity decay parameter is 0.02/failure. We assume that 45 failures have been experienced. Calculate the current failure intensity.
- 7.17. Explain the basic & logarithmic Poisson model and their significance in reliability studies.
- 7.18. Assume that a program will experience 150 failures in infinite time. It has now experienced 80. The initial failure intensity was 10 failures/CPU hr.
 - (i) Determine the current failure intensity.
 - (ii) Calculate the failures experienced and failure intensity after 25 and 40 CPU hrs. of execution.
 - (iii) Compute additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.
- Use the basic execution time model for the above mentioned calculations.
- 7.19. Write a short note on Logarithmic Poisson Execution time model. How can we calculate $\Delta\mu$ & $\Delta\tau$?
- 7.20. Assume that the initial failure intensity is 10 failures/CPU hr. The failure intensity decay parameter is 0.03/failure. We have experienced 75 failures upto this time. Find the failures experienced and failure intensity after 25 and 50 CPU hrs. of execution.

- 7.21. The following parameters for basic and logarithmic Poisson models are given:

<i>Basic execution time model</i>	<i>Logarithmic Poisson execution time model</i>
$\lambda_0 = 5 \text{ failures/CPU hr}$	$\lambda_0 = 25 \text{ failures/CPU hr}$
$V_0 = 125 \text{ failures}$	$\theta = 0.3/\text{failure}$

Determine the additional failures and additional execution time required to reach the failure intensity objective of 0.1 failure/CPU hr. for both models.

- 7.22. Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss them.
- 7.23. Discuss the calendar time component model. Establish the relationship between calendar time to execution time.
- 7.24. A program is expected to have 250 faults. It is also assumed that one fault may lead to one failure. The initial failure intensity is 5 failures/CPU hr. The program is released with a failure intensity objective of 4 failures/10 CPU hr. Calculate the number of failures experienced before release.
- 7.25. Explain the Jelinski-Moranda model of reliability theory. What is the relation between ' t ' and ' λ '?
- 7.26. Describe the Mill's bug seeding model. Discuss few advantages of this model over other reliability models.
- 7.27. Explain how the CMM encourages continuous improvement of the software process.
- 7.28. Discuss various key process areas of CMM at various maturity levels.
- 7.29. Construct a table that correlates key process areas (KPAs) in the CMM with ISO9000.
- 7.30. Discuss the 20 clauses of ISO9001 and compare with the practices in the CMM.
- 7.31. List the difference of CMM and ISO9001. Why is it suggested that CMM is the better choice than ISO9001 ?
- 7.32. Explain the significance of software reliability engineering. Discuss the advantages of using any software standard for software development ?
- 7.33. What are various key process areas at defined level in CMM? Describe activities associated with one key process area.
- 7.34. Discuss main requirements of ISO9001 and compare it with SEI capability maturity model.
- 7.35. Discuss the relative merits of ISO9001 certification and the SEI CMM based evaluation. Point out some of the shortcomings of the ISO 9001 certification process as applied to the software industry.

8

Software Testing

Contents

8.1 Testing Process

- 8.1.1 What is Testing ?
- 8.1.2 Why should We Test ?
- 8.1.3 Who should Do theTesting ?
- 8.1.4 What should We Test ?

8.2 Some Terminologies

- 8.2.1 Error, Mistake, Bug, Fault and Failure
- 8.2.2 Test, Test Case and Test Suite
- 8.2.3 Verification and Validation
- 8.2.4 Alpha, Beta and Acceptance Testing

8.3 Functional Testing

- 8.3.1 Boundary Value Analysis
- 8.3.2 Equivalence Class Testing
- 8.3.3 Decision Table Based Testing
- 8.3.4 Cause Effect Graphing Technique
- 8.3.5 Special Value Testing

8.4 Structural Testing

- 8.4.1 Path Testing
- 8.4.2 Cyclomatic Complexity
- 8.4.3 Graph Matrices
- 8.4.4 Data Flow Testing
- 8.4.5 Mutation Testing

8.5 Levels of Testing

- 8.5.1 Unit Testing
- 8.5.2 Integration Testing
- 8.5.3 System Testing

8.6 Debugging

- 8.6.1 Debugging Techniques
- 8.6.2 Debugging Approaches
- 8.6.3 Debugging Tools

8.7 Testing Tools

- 8.7.1 Static Testing Tools
- 8.7.2 Dynamic Testing Tools
- 8.7.3 Characteristics of Modern Tools

Software testing is the process of testing the software product. Effective software testing will contribute to the delivery of higher quality software products, more satisfied users, lower maintenance costs, more accurate, and reliable results. However, ineffective testing will lead to the opposite results; low quality products, unhappy users, increased maintenance costs, unreliable and inaccurate results. Hence, software testing is necessary and important activity of software development process. It is a very expensive process and consumes one-third to one-half of the cost of a typical development project. It is partly intuitive but largely systematic. Good testing involves much more than just running the program a few times to see whether it works. Thorough analysis of a program helps us to test more systematically and more effectively.

8.1 TESTING PROCESS

Software testing is a specialised discipline requiring unique skills. Software testing should not be intuitive as far as possible and we must learn how to do it systematically. Naive managers erroneously think that any developer can test software. Some may feel that if we can program, then we can test. What is great in it [TAMR03]? This may not be the right way of thinking.

Software is everywhere. However, it is written by people-so it is not perfect. We have seen many software failures like Intel Pentium Floating Point Division bug of 1994, NASA Mars Polar Lander of 1999, Patriot Missile Defense system of 1991, Y2K Problem etc. [PATT01].

8.1.1 What Is Testing?

Many people understand many definitions of testing. Few of them are given below:

1. Testing is the process of demonstrating that errors are not present.
2. The purpose of testing is to show that a program performs its intended functions correctly.
3. Testing is the process of establishing confidence that a program does what it is supposed to do.

These definitions are incorrect. They describe almost the opposite of what testing should be viewed as. Forgetting the definitions for the moment, consider that when we want to test a program, we want to add some value to the program. Adding value means raising the quality or reliability of the program. Raising the reliability of the program means finding and removing errors. Hence, we should not test a program to show that it works; rather we should start with the assumption that the program contains errors and then test the program to find as many of the errors as possible. Thus, a more appropriate definition is:

“Testing is the process of executing a program with the intent of finding errors”

Human beings are normally goal oriented. Thus, establishing the proper goal has an important psychological effect. If our goal is to demonstrate that a program has no errors, then we shall subconsciously steer towards this goal; that is , we will tend to select those inputs that have a low probability of causing the program to fail. On the other hand, if our goal is to demonstrate that a program has errors, our inputs selection will have a higher probability of finding errors. The second approach will add more value to the program than the first one. Thus, testing can not show the absence of errors, it can only show that errors are present [DAHL72].

According to most appropriate definition, there is a fundamental entity “errors are present within the software under test”. This cannot be the aim of software designers. They must have designed the software with the aim of producing it with zero errors. Therefore, whole effort of software engineering activities is to design methods and tools to eliminate errors at source. Software testing is becoming increasingly important in the earlier part of the software life cycle, aiming to discover errors before they are deeply embedded within systems. It is to be hoped that one-day software engineering will become refined to the degree that software testing will be fully integrated within each phase of software life cycle. After all, engineers building bridges do not need to test their products to destruction to predict the breaking point of their constructs. For the moment, in software, this is the only practical method open to us.

In software testing we are facing a major dilemma. On the one hand we wish to design the software product that has zero errors while on the other hand we must remain firm in our belief that any software product under testing certainly has errors, which need to be unearthed.

8.1.2 Why should We Test?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved. No one would think of allowing automatic pilot software into service without the most rigorous testing. In so-called life critical systems, economics must not be the prime consideration while deciding whether a product should be released to a customer.

In most systems, however, it is the cost factor, which plays a major role. It is both the driving force and the limiting factor as well. In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal. The most damaging errors are those, which are not discovered during the testing process and therefore remain when the system ‘goes live’. In commercial systems it is often difficult to estimate the costs of errors. For example, in a banking system, the potential cost of even a minor software error could be enormous. The consequential cost of lost business (which may never be recovered) can be beyond calculations.

It is not possible to test the software for all possible combinations of input cases. No software would ever be released by its creators if they were asked to certify that it was totally free of all errors. Testing therefore continues to the point where it is considered that the costs of the testing processes significantly outweigh the returns. Hence, when to release the software in the market, is a very important decision.

8.1.3 Who should Do the Testing?

The testing requires the developers to find errors from their software. It is very difficult for software developer to point out errors from own creations. Beizer [BEIZ90] explains this situation effectively when he states, "There is a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if everyone used structured programming, top-down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs, so goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test design amount to an admission of failure, which instils a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For not achieving inhuman perfection? For not distinguishing between what another developer thinks and what he says? For not being telepathic? For not solving human communication problems that have been kicked around by philosophers and theologians for 40 centuries?"

Many organisations have made a distinction between development and testing phase by making different people responsible for each phase. This has an additional advantage. Faced with the opportunity of testing someone else's software, our professional pride will demand that we achieve success. Success in testing is finding errors. We will therefore strive to reveal any errors present in the software. In other words, our ego would have been harnessed to the testing process, in a very positive way, in a way, which would be virtually impossible, were we testing our own software [NORM89]. Therefore, most of the times, testing persons are different from development persons for the overall benefit of the system. Developers provide guidelines during testing, however, whole responsibility is owned by testing persons.

8.1.4 What should We Test?

We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are $2^8 \times 2^8$. If only one second is required to execute one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.

Another dimension is to execute all possible paths of the program. A program path can be traced through the code from the start of the program to program termination. Two paths differ if the program executes different statements in each, or executes the same statements but in different order. A program may have many paths. Myers has explained this problem with a simple example [MYER79] where he used a loop and few IF statements as shown in Fig. 8.1.

The number of paths in the example of Fig. 8.1 are 10^{14} or 100 trillions. It is computed from $5^{20} + 5^{19} + 5^{18} + \dots + 5^1$; where 5 is the number of paths through the loop body. If only 5 minutes are required to test one path, it may take approximately one billion years to execute every path.

The point which we would like to highlight is that complete or exhaustive testing is just not possible. Exhaustive testing requires every statement in the program and every possible path combination to be executed at least once. So our objective is not possible to be achieved and we may have to settle for something less than that of complete testing.

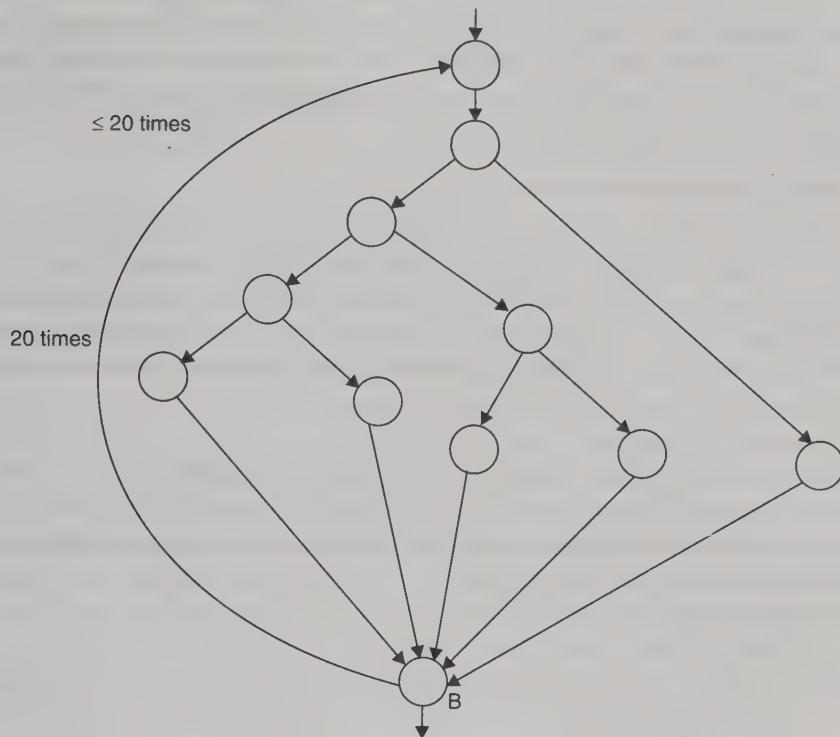


Fig. 8.1: Control flow graph [MYER79].

We may like to test those areas where probability of getting a fault is maximum. Such critical and sensitive areas are not easy to identify. Organisations should develop strategies and policies for choosing effective testing techniques rather than leaving this to arbitrary judgements of the development team.

8.2 SOME TERMINOLOGIES

Some terminologies are confusing and used interchangeably in literature and books. Institute of Electronics and Electrical Engineers (IEEE), USA has developed some standards which are discussed below:

8.2.1 Error, Mistake, Bug, Fault and Failure

People make errors. A good synonym is mistake. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors. When developers make mistakes while coding, we call these mistakes “bugs”. Errors propagate from one phase to another with higher severity. A requirement error may be magnified during design, and amplified still more during coding. If it could not be detected prior to release, it may have serious implications in the field.

An error may lead to one or more faults. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault. If fault is in source code, we call it a bug.

A failure occurs when a fault executes. It is the departure of the output of program from the expected output. Hence failure is dynamic. 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 how it has been exercised.

8.2.2 Test, Test Case and Test Suite

Test and Test case terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description. Inputs are of two types: pre conditions (circumstances that hold prior to test case execution) and the actual inputs that are identified by some testing methods. Expected outputs are also of two types: post conditions and actual outputs. Every test case will have an identification.

During testing, we set necessary preconditions, give required inputs to program, and compare the observed output with expected output to know the outcome of a test case. If expected and observed outputs are different, then, there is a failure and it must be recorded properly in order to identify the cause of failure. If both are same, then, there is no failure and program behaved in the expected manner. A good test case has a high probability of finding an error. The test case designer's main objective is to identify good test cases. The template for a typical test case is given in Fig. 8.2.

<i>Test Case ID:</i>	
<i>Section-I (Before Execution)</i>	<i>Section-II (After Execution)</i>
Purpose:	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional):
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig. 8.2: Test case template

Test cases are valuable and useful—at least as valuable as source code. They need to be developed, reviewed, used, managed, and saved.

The set of test cases is called a test suite. We may have a test suite of all possible test case. We may have a test suite of effective/good test cases. Hence any combination of test cases may generate a test suite.

8.2.3 Verification and Validation

Verification and validation are often used interchangeably but have different meanings. Verification is the process of confirming that software meets its specification.

However, Validation is the process of confirming that software meets the customer's requirements. Both the definitions may seem to be similar. One is related to specification and

other is related to customer's requirements. We should not assume that our specifications are always correct. We should verify the specifications and validate the final product. In general, we may say;

Verification: Checking the software with respect to specifications.

Validation: Checking the software with respect to customer's expectations.

Many times customer's requirements are not being translated in SRS correctly. Sometimes expectations are vague or unrealistic. Sometimes poor understanding of expectations due to communication gap and understanding levels may lead to incorrect specifications. Our objective is to reduce this gap before the finalisation of SRS.

If there is a gap at SRS level that will only be known during validation activities. The verification and Validation activities are combined under a broad term known as software testing.

8.2.4 Alpha, Beta and Acceptance Testing

It is not possible to predict the every usage of the software by the customer. Customer may try with strange inputs, combination of inputs and so many other things. Some output may be very clear from the developer's perspective, but customer may not understand and finally may not appreciate it. In order to avoid or minimise such situations, customer involvement is required before delivering the final product. The above mentioned three terms are related to customer's involvement in testing but have different meanings.

Acceptance testing

This term is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user/customer and may range from adhoc tests to well planned systematic series of tests. Acceptance testing may be conducted for few weeks or months. The discovered errors will be fixed and better quality software will be delivered to the customer.

Alpha and beta testing

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers. Hence formal acceptance testing is not possible in such cases. However, some potential customers are identified to get their views about the product. The alpha tests are conducted at the developer's site by a customer. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

The beta tests are conducted by the customers/end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer. Customers are expected to report failures, if any, to the company. After receiving such failure reports, developers modify the code and fix the bug and prepare the product for final release.

Most of the companies are following this practice firstly, they send the beta release of their product for few months. Many potential customers will use the product and may send their views about the product. Some may encounter with failure situations and may report to the company. Hence, company gets the feedback of many potential customers. The best part is

that the reputation of the company is not at stake even if many failure situations are encountered.

8.3 FUNCTIONAL TESTING

As discussed earlier, complete testing is not at all possible. Thus, we may like to reduce this incompleteness as much as possible. Probably the poorest methodology is random input testing. In random input testing, some subset of all input values are selected randomly. In terms of probability of detecting errors, a randomly selected collection of test cases has little chance of being an optimal, or close to optimal, subset. What we are looking for is a set of thought processes that allow us to select a set of data more intelligently.

One way to examine this issue is to explore a strategy where testing is based on the functionality of the program and is known as functional testing. Thus, functional testing refers to testing, which involves only observation of the output for certain input values. There is no attempt to analyse the code, which produces the output. We ignore the internal structure of the code. Therefore, functional testing is also referred to as black box testing in which contents of the black box are not known. Functionality of the black box is understood completely in terms of its inputs and outputs as shown in Fig. 8.3. Here, we are interested in functionality rather than internal structure of the code. Many times we operate more effectively with black box knowledge. For example, most people successfully operate automobiles with only black box knowledge.

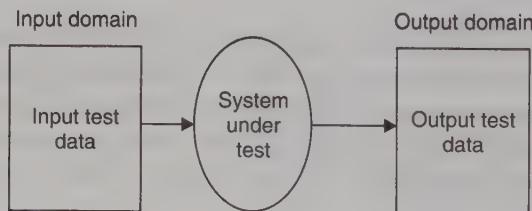


Fig. 8.3: Black box testing.

There are a number of strategies or techniques that can be used to design test cases which have been found to be very successful in detecting errors.

8.3.1 Boundary Value Analysis

Experience shows that test cases that are close to boundary conditions have a higher chances of detecting an error. Here boundary condition means, an input value may be on the boundary, just below the boundary (upper side) or just above the boundary (lower side). Suppose, we have an input variable x with a range from 1 to 100. The boundary values are 1, 2, 99 and 100.

Consider a program with two input variables x and y . These input variables have specified boundaries as:

$$a \leq x \leq b$$

$$c \leq y \leq d$$

Hence both the inputs x and y are bounded by two intervals $[a, b]$ and $[c, d]$ respectively. For input x , we may design test cases with values a and b , just above a and also just below b .

Similarly for input y , we may have values c and d , just above c and also just below d . These test cases will have more chances to detect an error [JORG95]. The input domain for our program is shown in Fig. 8.4. Any point within the inner rectangle is a legitimate input to the program.

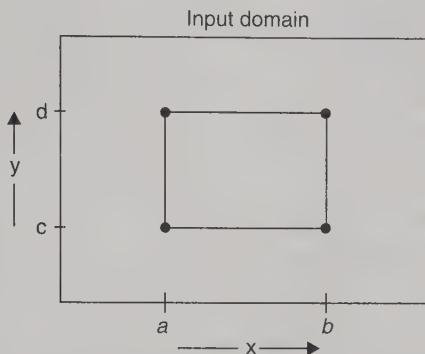


Fig. 8.4: Input domain for program having two input variables.

The basic idea of boundary value analysis is to use input variable values at their minimum, just above minimum, a nominal value, just below their maximum, and at their maximum.

Here, we have an assumption of reliability theory known as “single fault” assumption. This says that failures are rarely the result of the simultaneous occurrence of two (or more) faults. Thus, boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values and letting that variable assume its extreme values. The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200, 100), (200, 101), (200, 200), (200, 299), (200, 300), (100, 200), (101, 200), (299, 200) and (300, 200). This input domain is shown in Fig. 8.5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yields $4n + 1$ test cases.

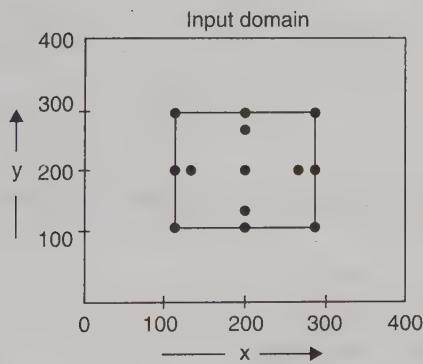


Fig. 8.5: Input domain of two variables x and y with boundaries [1, 200] each.

Example 8.1

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a, b, c) and values may be from interval $[0, 100]$. The program output may have one of the following words:

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Solution

Quadratic equation will be of type:

$$ax^2 + bx + c = 0$$

Roots are real if $(b^2 - 4ac) > 0$

Roots are imaginary if $(b^2 - 4ac) < 0$

Roots are equal if $(b^2 - 4ac) = 0$

Equation is not quadratic if $a = 0$

The boundary value test cases are:

Test Case	a	b	c	Expected output
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Example 8.2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$$1 \leq \text{month} \leq 12$$

$$1 \leq \text{day} \leq 31$$

$$1900 \leq \text{year} \leq 2025$$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Solution

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

As we know, with single fault assumption theory, $4n + 1$ test cases can be designed and which are equal to 13. The boundary value test cases are:

Test case	Month	Day	Year	Expected output
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Example 8.3

Consider a simple program to classify a triangle. Its input is a triple of positive integers (say x , y , z) and the data type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle]

Design the boundary value test cases.

Solution

The boundary value test cases are shown below:

Test case	x	y	z	Expected output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles

(Contd.)...

Test case	x	y	z	Expected output
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle

Robustness testing

It is nothing but the extension of boundary value analysis. Here, we would like to see, what happens when the extreme values are exceeded with a value slightly greater than the maximum and a value slightly less than minimum. It means, we want to go outside the legitimate boundary of input domain. This type of testing is quite common in electric and electronic circuits. This extended form of boundary value analysis is called robustness testing and shown in Fig. 8.6.

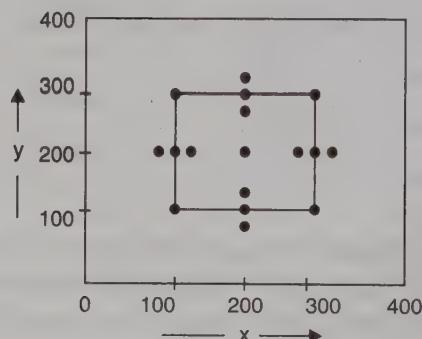


Fig. 8.6: Robustness test cases for two variables x and y with range [1, 200] each.

There are four additional test cases which are outside the legitimate input domain. Hence total test cases in robustness testing are $6n + 1$, where n is the number of input variables. So, 13 test cases are:

$$(200, 99), (200, 100), (200, 101), (200, 200), (200, 299), (200, 300), \\ (200, 301), (99, 200), (100, 200), (101, 200), (299, 200), (300, 200), (301, 200).$$

Worst-case testing

If we reject “single fault” assumption theory of reliability and may like to see what happens when more than one variable has an extreme value. In electronic circuits analysis, this is called “worst case analysis”. It is more thorough in the sense that boundary value test cases are a proper subset of worst case test cases. It requires more effort. Worst case testing for a function of n variables generates 5^n test cases as opposed to $4n + 1$ test cases for boundary value analysis. Our two variable example will have $5^2 = 25$ test cases and are given in Table 8.1.

Table 8.1: Worst case test inputs for two variable example.

Test case number	Inputs		Test case number	Inputs	
	x	y		x	y
1	100	100	14	200	299
2	100	101	15	200	300
3	100	200	16	299	100
4	100	299	17	299	101
5	100	300	18	299	200
6	101	100	19	299	299
7	101	101	20	299	300
8	101	200	21	300	100
9	101	299	22	300	101
10	101	300	23	300	200
11	200	100	24	300	299
12	200	101	25	300	300
13	200	200	—		

Boundary value analysis works well for the programs with independent input values. Here input values should be truly independent. This technique does not make sense for boolean variables where extreme values are TRUE and FALSE, but no clear choice is available for other values like nominal, just above boundary and just below boundary.

Example 8.4

Consider the program for the determination of nature of roots of a quadratic equation as explained in Example 8.1. Design the Robust test cases and worst test cases for this program.

Solution

As we know, robust test cases are $6n + 1$. Hence, in 3 variable input cases total number of test cases are 19 as given below:

Test case	a	b	c	Expected output
1	-1	50	50	Invalid Input
2	0	50	50	Not quadratic Equation
3	1	50	50	Real Roots
4	50	50	50	Imaginary Roots
5	99	50	50	Imaginary Roots
6	100	50	50	Imaginary Roots
7	101	50	50	Invalid Input

(Contd.)...

<i>Test case</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>Expected output</i>
8	50	-1	50	Invalid Input
9	50	0	50	Imaginary Roots
10	50	1	50	Imaginary Roots
11	50	99	50	Imaginary Roots
12	50	100	50	Equal Roots
13	50	101	50	Invalid Input
14	50	50	-1	Invalid Input
15	50	50	0	Real Roots
16	50	50	1	Real Roots
17	50	50	99	Imaginary Roots
18	50	50	100	Imaginary Roots
19	50	50	101	Invalid Input

In case of worst test case total test cases are 5^n . Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

<i>Test case</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>Expected output</i>
1	0	0	0	Not Quadratic
2	0	0	1	Not Quadratic
3	0	0	50	Not Quadratic
4	0	0	99	Not Quadratic
5	0	0	100	Not Quadratic
6	0	1	0	Not Quadratic
7	0	1	1	Not Quadratic
8	0	1	50	Not Quadratic
9	0	1	99	Not Quadratic
10	0	1	100	Not Quadratic
11	0	50	0	Not Quadratic
12	0	50	1	Not Quadratic
13	0	50	50	Not Quadratic
14	0	50	99	Not Quadratic
15	0	50	100	Not Quadratic
16	0	99	0	Not Quadratic
17	0	99	1	Not Quadratic
18	0	99	50	Not Quadratic

(Contd.)...

<i>Test case</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>Expected output</i>
19	0	99	99	Not Quadratic
20	0	99	100	Not Quadratic
21	0	100	0	Not Quadratic
22	0	100	1	Not Quadratic
23	0	100	50	Not Quadratic
24	0	100	99	Not Quadratic
25	0	100	100	Not Quadratic
26	1	0	0	Equal Roots
27	1	0	1	Imaginary
28	1	0	50	Imaginary
29	1	0	99	Imaginary
30	1	0	100	Imaginary
31	1	1	0	Real Roots
32	1	1	1	Imaginary
33	1	1	50	Imaginary
34	1	1	99	Imaginary
35	1	1	100	Imaginary
36	1	50	0	Real Roots
37	1	50	1	Real Roots
38	1	50	50	Real Roots
39	1	50	99	Real Roots
40	1	50	100	Real Roots
41	1	99	0	Real Roots
42	1	99	1	Real Roots
43	1	99	50	Real Roots
44	1	99	99	Real Roots
45	1	99	100	Real Roots
46	1	100	0	Real Roots
47	1	100	1	Real Roots
48	1	100	50	Real Roots
49	1	100	99	Real Roots
50	1	100	100	Real Roots
51	50	0	0	Equal Roots
52	50	0	1	Imaginary
53	50	0	50	Imaginary

(Contd.)...

<i>Test case</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>Expected output</i>
54	50	0	99	Imaginary
55	50	0	100	Imaginary
56	50	1	0	Real Roots
57	50	1	1	Imaginary
58	50	1	50	Imaginary
59	50	1	99	Imaginary
60	50	1	100	Imaginary
61	50	50	0	Real Roots
62	50	50	1	Real Roots
63	50	50	50	Imaginary
64	50	50	99	Imaginary
65	50	50	100	Imaginary
66	50	99	0	Real Root
67	50	99	1	Real Root
68	50	99	50	Imaginary
69	50	99	99	Imaginary
70	50	99	100	Imaginary
71	50	100	0	Real Roots
72	50	100	1	Real Roots
73	50	100	50	Equal Roots
74	50	100	99	Imaginary
75	50	100	100	Imaginary
76	99	0	0	Equal Roots
77	99	0	1	Imaginary
78	99	0	50	Imaginary
79	99	0	99	Imaginary
80	99	0	100	Imaginary
81	99	1	0	Real Roots
82	99	1	1	Imaginary
83	99	1	50	Imaginary
84	99	1	99	Imaginary
85	99	1	100	Imaginary
86	99	50	0	Real Roots
87	99	50	1	Real Roots
88	99	50	50	Imaginary
89	99	50	99	Imaginary

<i>Test case</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>Expected output</i>
90	99	50	100	Imaginary
91	99	99	0	Real Roots
92	99	99	1	Real Roots
93	99	99	50	Imaginary Roots
94	99	99	99	Imaginary
95	99	99	100	Imaginary
96	99	100	0	Real Roots
97	99	100	1	Real Roots
98	99	100	50	Imaginary
99	99	100	99	Imaginary
100	99	100	100	Imaginary
101	100	0	0	Equal Roots
102	100	0	1	Imaginary
103	100	0	50	Imaginary
104	100	0	99	Imaginary
105	100	0	100	Imaginary
106	100	1	0	Real Roots
107	100	1	1	Imaginary
108	100	1	50	Imaginary
109	100	1	99	Imaginary
110	100	1	100	Imaginary
111	100	50	0	Real Roots
112	100	50	1	Real Roots
113	100	50	50	Imaginary
114	100	50	99	Imaginary
115	100	50	100	Imaginary
116	100	99	0	Real Roots
117	100	99	1	Real Roots
118	100	99	50	Imaginary
119	100	99	99	Imaginary
120	100	99	100	Imaginary
121	100	100	0	Real Roots
122	100	100	1	Real Roots
123	100	100	50	Imaginary
124	100	100	99	Imaginary
125	100	100	100	Imaginary

Example 8.5

Consider the program for the determination of previous date in a calendar as explained in Example 8.2. Design the robust and worst test cases for this program.

Solution

Robust test cases are $6n + 1$. Hence total 19 robust test cases are designed and are given below:

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
1	6	15	1899	Invalid date (outside range)
2	6	15	1900	14 June, 1900
3	6	15	1901	14 June, 1901
4	6	15	1962	14 June, 1962
5	6	15	2024	14 June, 2024
6	6	15	2025	14 June, 2025
7	6	15	2026	Invalid date (outside range)
8	6	0	1962	Invalid date
9	6	1	1962	31 May, 1962
10	6	2	1962	1 June, 1962
11	6	30	1962	29 June, 1962
12	6	31	1962	Invalid date
13	6	32	1962	Invalid date
14	0	15	1962	Invalid date
15	1	15	1962	14 January, 1962
16	2	15	1962	14 February, 1962
17	11	15	1962	14 November, 1962
18	12	15	1962	14 December, 1962
19	13	15	1962	Invalid date

Worst test cases are 5^n and n is 3. Hence 125 test cases are generated and are given below:

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
1	1	1	1900	31 December, 1899
2	1	1	1901	31 December, 1900
3	1	1	1962	31 December, 1961
4	1	1	2024	31 December, 2023
5	1	1	2025	31 December, 2024
6	1	2	1900	1 January, 1900
7	1	2	1901	1 January, 1901

(Contd.)...

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
8	1	2	1962	1 January, 1962
9	1	2	2024	1 January, 2024
10	1	2	2025	1 January, 2025
11	1	15	1900	14 January, 1900
12	1	15	1901	14 January, 1901
13	1	15	1962	14 January, 1962
14	1	15	2024	14 January, 2024
15	1	15	2025	14 January, 2025
16	1	30	1900	29 January, 1900
17	1	30	1901	29 January, 1901
18	1	30	1962	29 January, 1962
19	1	30	2024	29 January, 2024
20	1	30	2025	29 January, 2025
21	1	31	1900	30 January, 1900
22	1	31	1901	30 January, 1901
23	1	31	1962	30 January, 1962
24	1	31	2024	30 January, 2024
25	1	31	2025	30 January, 2025
26	2	1	1900	31 January, 1900
27	2	1	1901	31 January, 1901
28	2	1	1962	31 January, 1962
29	2	1	2024	31 January, 2024
30	2	1	2025	31 January, 2025
31	2	2	1900	1 February, 1900
32	2	2	1901	1 February, 1901
33	2	2	1962	1 February, 1962
34	2	2	2024	1 February, 2024
35	2	2	2025	1 February, 2025
36	2	15	1900	14 February, 1900
37	2	15	1901	14 February, 1901
38	2	15	1962	14 February, 1962
39	2	15	2024	14 February, 2024
40	2	15	2025	14 February, 2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date

(Contd.)...

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	31 May, 1900
52	6	1	1901	31 May, 1901
53	6	1	1962	31 May, 1962
54	6	1	2024	31 May, 2024
55	6	1	2025	31 May, 2025
56	6	2	1900	1 June, 1900
57	6	2	1901	1 June, 1901
58	6	2	1962	1 June, 1962
59	6	2	2024	1 June, 2024
60	6	2	2025	1 June, 2025
61	6	15	1900	14 June, 1900
62	6	15	1901	14 June, 1901
63	6	15	1962	14 June, 1962
64	6	15	2024	14 June, 2024
65	6	15	2025	14 June, 2025
66	6	30	1900	29 June, 1900
67	6	30	1901	29 June, 1901
68	6	30	1962	29 June, 1962
69	6	30	2024	29 June, 2024
70	6	30	2025	29 June, 2025
71	6	31	1900	Invalid date
72	6	31	1901	Invalid date
73	6	31	1962	Invalid date
74	6	31	2024	Invalid date
75	6	31	2025	Invalid date
76	11	1	1900	31 October, 1900
77	11	1	1901	31 October, 1901
78	11	1	1962	31 October, 1962

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
79	11	1	2024	31 October, 2024
80	11	1	2025	31 October, 2025
81	11	2	1900	1 November, 1900
82	11	2	1901	1 November, 1901
83	11	2	1962	1 November, 1962
84	11	2	2024	1 November, 2024
85	11	2	2025	1 November, 2025
86	11	15	1900	14 November, 1900
87	11	15	1901	14 November, 1901
88	11	15	1962	14 November, 1962
89	11	15	2024	14 November, 2024
90	11	15	2025	14 November, 2025
91	11	30	1900	29 November, 1900
92	11	30	1901	29 November, 1901
93	11	30	1962	29 November, 1962
94	11	30	2024	29 November, 2024
95	11	30	2025	29 November, 2025
96	11	31	1900	Invalid date
97	11	31	1901	Invalid date
98	11	31	1962	Invalid date
99	11	31	2024	Invalid date
100	11	31	2025	Invalid date
101	12	1	1900	30 November, 1900
102	12	1	1901	30 November, 1901
103	12	1	1962	30 November, 1962
104	12	1	2024	30 November, 2024
105	12	1	2025	30 November, 2025
106	12	2	1900	1 December, 1900
107	12	2	1901	1 December, 1901
108	12	2	1962	1 December, 1962
109	12	2	2024	1 December, 2024
110	12	2	2025	1 December, 2025
111	12	15	1900	14 December, 1900
112	12	15	1901	14 December, 1901
113	12	15	1962	14 December, 1962
114	12	15	2024	14 December, 2024

(Contd.)...

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
115	12	15	2025	14 December, 2025
116	12	30	1900	29 December, 1900
117	12	30	1901	29 December, 1901
118	12	30	1962	29 December, 1962
119	12	30	2024	29 December, 2024
120	12	30	2025	29 December, 2025
121	12	31	1900	30 December, 1900
122	12	31	1901	30 December, 1901
123	12	31	1962	30 December, 1962
124	12	31	2024	30 December, 2024
125	12	31	2025	30 December, 2025

Example 8.6

Consider the triangle problem as given in Example 8.3. Generate robust and worst test cases for this problem.

Solution

Robust test cases are:

<i>Test case</i>	<i>x</i>	<i>y</i>	<i>z</i>	<i>Expected output</i>
1	50	50	0	Invalid input
2	50	50	1	Isosceles
3	50	50	2	Isosceles
4	50	50	50	Equilateral
5	50	50	99	Isosceles
6	50	50	100	Not a triangle
7	50	50	101	Invalid input
8	50	0	50	Invalid input
9	50	1	50	Isosceles
10	50	2	50	Isosceles
11	50	99	50	Isosceles
12	50	100	50	Not a triangle
13	50	101	50	Invalid input
14	0	50	50	Invalid input
15	1	50	50	Isosceles
16	2	50	50	Isosceles
17	99	50	50	Isosceles
18	100	50	50	Not a triangle
19	101	50	50	Invalid input

Worst test cases are 125 and are given below:

Test case	x	y	z	Expected output
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	50	Not a triangle
4	1	1	99	Not a triangle
5	1	1	100	Not a triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	50	Not a triangle
9	1	2	99	Not a triangle
10	1	2	100	Not a triangle
11	1	50	1	Not a triangle
12	1	50	2	Not a triangle
13	1	50	50	Isosceles
14	1	50	99	Not a triangle
15	1	50	100	Not a triangle
16	1	99	1	Not a triangle
17	1	99	2	Not a triangle
18	1	99	50	Not a triangle
19	1	99	99	Isosceles
20	1	99	100	Not a triangle
21	1	100	1	Not a triangle
22	1	100	2	Not a triangle
23	1	100	50	Not a triangle
24	1	100	99	Not a triangle
25	1	100	100	Isosceles
26	2	1	1	Not a triangle
27	2	1	2	Isosceles
28	2	1	50	Not a triangle
29	2	1	99	Not a triangle
30	2	1	100	Not a triangle
31	2	2	1	Isosceles
32	2	2	2	Equilateral
33	2	2	50	Not a triangle
34	2	2	99	Not a triangle

(Contd.)...

<i>Test case</i>	<i>x</i>	<i>y</i>	<i>z</i>	<i>Expected output</i>
35	2	2	100	Not a triangle
36	2	50	1	Not a triangle
37	2	50	2	Not a triangle
38	2	50	50	Isosceles
39	2	50	99	Not a triangle
40	2	50	100	Not a triangle
41	2	99	1	Not a triangle
42	2	99	2	Not a triangle
43	2	99	50	Not a triangle
44	2	99	99	Isosceles
45	2	99	100	Scalene
46	2	100	1	Not a triangle
47	2	100	2	Not a triangle
48	2	100	50	Not a triangle
49	2	100	99	Scalene
50	2	100	100	Isosceles
51	50	1	1	Not a triangle
52	50	1	2	Not a triangle
53	50	1	50	Isosceles
54	50	1	99	Not a triangle
55	50	1	100	Not a triangle
56	50	2	1	Not a triangle
57	50	2	2	Not a triangle
58	50	2	50	Isosceles
59	50	2	99	Not a triangle
60	50	2	100	Not a triangle
61	50	50	1	Isosceles
62	50	50	2	Isosceles
63	50	50	50	Equilateral
64	50	50	99	Isosceles
65	50	50	100	Not a triangle
66	50	99	1	Not a triangle
67	50	99	2	Not a triangle
68	50	99	50	Isosceles
69	50	99	99	Isosceles

(Contd.)...

Test case	x	y	z	Expected output
70	50	99	100	Scalene
71	50	100	1	Not a triangle
72	50	100	2	Not a triangle
73	50	100	50	Not a triangle
74	50	100	99	Scalene
75	50	100	100	Isosceles
76	99	1	1	Not a triangle
77	99	1	2	Not a triangle
78	99	1	50	Not a triangle
79	99	1	99	Isosceles
80	99	1	100	Not a triangle
81	99	2	1	Not a triangle
82	99	2	2	Not a triangle
83	99	2	50	Not a triangle
84	99	2	99	Isosceles
85	99	2	100	Scalene
86	99	50	1	Not a triangle
87	99	50	2	Not a triangle
88	99	50	50	Isosceles
89	99	50	99	Isosceles
90	99	50	100	Scalene
91	99	99	1	Isosceles
92	99	99	2	Isosceles
93	99	99	50	Isosceles
94	99	99	99	Equilateral
95	99	99	100	Isosceles
96	99	100	1	Not a triangle
97	99	100	2	Scalene
98	99	100	50	Scalene
99	99	100	99	Isosceles
100	99	100	100	Isosceles
101	100	1	1	Not a triangle
102	100	1	2	Not a triangle
103	100	1	50	Not a triangle
104	100	1	99	Not a triangle

(Contd.)...

Test case	x	y	z	Expected output
105	100	1	100	Isosceles
106	100	2	1	Not a triangle
107	100	2	2	Not a triangle
108	100	2	50	Not a triangle
109	100	2	99	Scalene
110	100	2	100	Isosceles
111	100	50	1	Not a triangle
112	100	50	2	Not a triangle
113	100	50	50	Not a triangle
114	100	50	99	Scalene
115	100	50	100	Isosceles
116	100	99	1	Not a triangle
117	100	99	2	Scalene
118	100	99	50	Scalene
119	100	99	99	Isosceles
120	100	99	100	Isosceles
121	100	100	1	Isosceles
122	100	100	2	Isosceles
123	100	100	50	Isosceles
124	100	100	99	Isosceles
125	100	100	100	Equilateral

8.3.2 Equivalence Class Testing

In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value. That is, if one test case in a class detects an error, all other test cases in the class would be expected to find same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the class would find an error. Two steps are required in implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class $[1 < \text{item} < 999]$; and two invalid equivalence classes $[\text{item} < 1]$ and $[\text{item} > 999]$.
2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.

In Fig. 8.7, both valid and invalid input domains are shown.

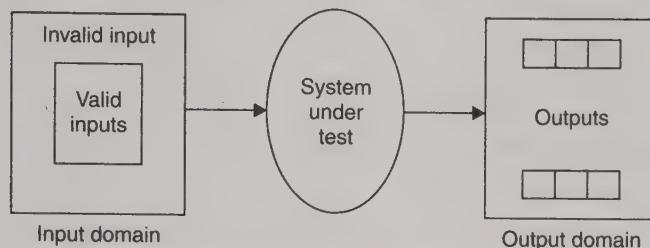


Fig. 8.7: Equivalence partitioning.

The idea is to choose at least one element from each equivalence class. In the triangle problem we would certainly have a test case for equivalent triangle, and we may take (50, 50, 50) as inputs for a test case. If we do this, we would not expect to learn much from test cases such as (40, 40, 40) and (100, 100, 100). These test cases may be treated as redundant test cases.

We should not forget to have equivalence classes for invalid inputs. This is often best source of bugs. We should test different types of invalid inputs in order to get more errors. As an example, for a program that is supposed to accept any number between 1 and 99, there are at least four equivalence classes from input side. The classes are:

- (i) Any number between 1 and 99 is valid input.
- (ii) Any number less than 1. This include 0 and all negative numbers.
- (iii) Any number greater than 99.
- (iv) If it is not a number, it should not be accepted.

Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domains.

Example 8.7

Consider the program for the determination of nature of roots of a quadratic equation as explained in Example 8.1. Identify the equivalence class test cases for output and input domains.

Solution

Output domain equivalence class test cases can be identified as follows:

$$O_1 = \{<a, b, c>: \text{Not a quadratic equation if } a = 0\}$$

$$O_2 = \{<a, b, c>: \text{Real roots if } (b^2 - 4ac) > 0\}$$

$$O_3 = \{<a, b, c>: \text{Imaginary roots if } (b^2 - 4ac) < 0\}$$

$$O_4 = \{<a, b, c>: \text{Equal roots if } (b^2 - 4ac) = 0\}$$

The number of test cases can be derived from above relations and shown below:

Test case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots
3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

We may have another set of test cases based on input domain.

$$\begin{aligned}I_1 &= \{a: a = 0\} \\I_2 &= \{a: a < 0\} \\I_3 &= \{a: 1 \leq a \leq 100\} \\I_4 &= \{a: a > 100\} \\I_5 &= \{b: 0 \leq b \leq 100\} \\I_6 &= \{b: b < 0\} \\I_7 &= \{b: b > 100\} \\I_8 &= \{c: 0 \leq c \leq 100\} \\I_9 &= \{c: c < 0\} \\I_{10} &= \{c: c > 100\}\end{aligned}$$

In these classes, our basic assumption is single fault theory. Hence one value is at an extreme and other values are nominal values. Input domain test cases are:

Test case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary roots
4	101	50	50	Invalid input
5	50	50	50	Imaginary roots
6	50	-1	50	Invalid input
7	50	101	50	Invalid input
8	50	50	50	Imaginary roots
9	50	50	-1	Invalid input
10	50	50	101	Invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are $10 + 4 = 14$ for this problem.

Example 8.8

Consider the program for determining the Previous date in a calendar as explained in Example 8.3. Identify the equivalence class test cases for output & input domains.

Solution

Output domain equivalence classes are:

$$O_1 = \{< D, M, Y >: \text{Previous date if all are valid inputs}\}$$

$$O_2 = \{< D, M, Y >: \text{Invalid date if any input makes the date invalid}\}$$

Test case	M	D	Y	Expected output
1	6	15	1962	14 June, 1962
2	6	31	1962	Invalid date

We may have another set of test cases which are based on input domain.

$$I_1 = \{\text{month: } 1 \leq m \leq 12\}$$

$$I_2 = \{\text{month: } m < 1\}$$

$$I_3 = \{\text{month: } m > 12\}$$

$$I_4 = \{\text{day: } 1 \leq D \leq 31\}$$

$$I_5 = \{\text{day: } D < 1\}$$

$$I_6 = \{\text{day: } D > 31\}$$

$$I_7 = \{\text{year: } 1900 \leq Y \leq 2025\}$$

$$I_8 = \{\text{year: } Y < 1900\}$$

$$I_9 = \{\text{year: } Y > 2025\}$$

Input domain test cases are :

Test case	M	D	Y	Expected output
1	6	15	1962	14 June, 1962
2	-1	15	1962	Invalid input
3	13	15	1962	Invalid input
4	6	15	1962	14 June, 1962
5	6	-1	1962	Invalid input
6	6	32	1962	Invalid input
7	6	15	1962	14 June, 1962
8	6	15	1899	Invalid input (value out of range)
9	6	15	2026	Invalid input (value out of range)

Example 8.9

Consider the triangle problem specified in example 8.3. Identify the equivalence class test cases for output and input domain.

Solution

Output domain equivalence classes are:

$$O_1 = \{<x, y, z>: \text{Equilateral triangle with sides } x, y, z\}$$

$$O_2 = \{<x, y, z>: \text{Isosceles triangle with sides } x, y, z\}$$

$$O_3 = \{<x, y, z>: \text{Scalene triangle with sides } x, y, z\}$$

$$O_4 = \{<x, y, z>: \text{Not a triangle with sides } x, y, z\}$$

The test cases are:

Test case	x	y	z	Expected output
1	50	50	50	Equilateral
2	50	50	99	Isosceles
3	100	99	50	Scalene
4	50	100	50	Not a triangle

Input domain based classes are:

$$I_1 = \{x: x < 1\}$$

$$I_2 = \{x: x > 100\}$$

$$I_3 = \{x: 1 \leq x \leq 100\}$$

$$I_4 = \{y: y < 1\}$$

$$I_5 = \{y: y > 100\}$$

$$I_6 = \{y: 1 \leq y \leq 100\}$$

$$I_7 = \{z: z < 1\}$$

$$I_8 = \{z: z > 100\}$$

$$I_9 = \{z: 1 \leq z \leq 100\}$$

Some input domain test cases can be obtained using the relationship amongst x, y and z.

$$I_{10} = \{<x, y, z>; x = y = z\}$$

$$I_{11} = \{<x, y, z>; x = y, x \neq z\}$$

$$I_{12} = \{<x, y, z>; x = z, x \neq y\}$$

$$I_{13} = \{<x, y, z>; y = z, x \neq y\}$$

$$I_{14} = \{<x, y, z>; x \neq y, x \neq z, y \neq z\}$$

$$I_{15} = \{<x, y, z>; x = y + z\}$$

$$I_{16} = \{<x, y, z>; x > y + z\}$$

$$I_{17} = \{<x, y, z>; y = x + z\}$$

$$I_{18} = \{<x, y, z>; y > x + z\}$$

$$I_{19} = \{<x, y, z>; z = x + y\}$$

$$I_{20} = \{<x, y, z>; z > x + y\}$$

Test cases derived from input domain are:

Test case	x	y	z	Expected output
1	0	50	50	Invalid input
2	101	50	50	Invalid input
3	50	50	50	Equilateral
4	50	0	50	Invalid input
5	50	101	50	Invalid input
6	50	50	50	Equilateral
7	50	50	0	Invalid input
8	50	50	101	Invalid input
9	50	50	50	Equilateral
10	60	60	60	Equilateral
11	50	50	60	Isosceles
12	50	60	50	Isosceles
13	60	50	50	Isosceles
14	100	99	50	Scalene
15	100	50	50	Not a triangle
16	100	50	25	Not a triangle
17	50	100	50	Not a triangle
18	50	100	25	Not a triangle
19	50	50	100	Not a triangle
20	25	50	100	Not a triangle

8.3.3 Decision Table Based Testing

Decisions tables are useful for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Decision tables have been used to represent and analyse complex logical relationships since early 1960s. We would like to show that how these may be applied to the testing and how tester may adopt the principles to his/her own situation. Some of the basic terms are shown in table 8.2.

Table 8.2: Decision table terminology

Condition		Entry							
		True				False			
Stub	c_1	True		False		True		False	
	c_2	True	False	True	False	True	False	—	—
Action Stub	a_1	X	X			X			
	a_2	X		X			X		
	a_3		X			X			
	a_4				X		X		X

There are four portions of a decision table namely, Conditions stub, Action Stub, Condition entries & Action entries. When conditions c_1, c_2 and c_3 are all true, actions a_1 and a_2 occur. When conditions c_1 & c_2 are true and c_3 is false, actions a_1 and a_3 occur. The decision tables in which all entries are binary are called limited entry decision tables. If conditions are allowed to have several values, the resulting tables are called Extended Entry Decision tables.

Test case design

To identify test cases with decision tables, we interpret conditions as inputs, and actions as outputs. Sometimes, conditions end up referring to equivalence classes of inputs, and actions refers to major functional processing portions of the item being tested. The rules are then interpreted as test cases. Because the decision table can mechanically be forced to be completed, we know, we have a comprehensive set of test cases. There are several techniques that produce decision tables that are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible [JORG95].

Consider the decision table shown in table 8.3, we see examples of don't care entries and impossible rule usage.

Table 8.3: Decision table for triangle problem

$c_1 : x, y, z$ are sides of a triangle?	N	Y							
	—	Y				N			
	—	Y		N		Y		N	
	—	Y	N	Y	N	Y	N	Y	N
$a_1 : \text{Not a triangle}$	X								
									X
					X		X	X	
		X							
			X	X		X			
$a_2 : \text{Scalene}$									
$a_3 : \text{Isosceles}$									
$a_4 : \text{Equilateral}$									
$a_5 : \text{Impossible}$									

If the integers x, y and z do not constitute a triangle, we do not even care about possible equalities, we may also choose conditions, but this will increase the size of the decision table as shown in table 8.4. Here, old condition ($c_1 : x, y, z$ are sides of a triangle?) has been expanded to get a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle.

Table 8.4: Modified decision table

Conditions	F	T	T	T	T	T	T	T	T	T	T	T
$c_1 : x < y + z ?$	—	F	T	T	T	T	T	T	T	T	T	T
$c_2 : y < x + z ?$	—	—	F	T	T	T	T	T	T	T	T	T
$c_3 : z < x + y ?$	—	—	—	T	T	T	T	T	T	T	T	T
$c_4 : x = y ?$	—	—	—	T	T	T	T	F	F	F	F	F
$c_5 : x = z ?$	—	—	—	T	T	F	F	T	T	F	F	F
$c_6 : y = z ?$	—	—	—	T	F	T	F	T	F	T	F	F

(Contd.)...

a₁ : Not a triangle	X	X	X									
a₂ : Scalene												X
a₃ : Isosceles							X		X	X	X	
a₄ : Equilateral				X								
a₅ : Impossible					X	X		X				

This is another way of representing the same thing. Table 8.3 seems to be more readable.

Example 8.10

Consider the triangle problem specified in example 8.3. Identify the test cases using the decision table of Table 8.4.

Solution

There are eleven functional test cases; three to fail triangle property, three impossible cases; one each to get equilateral, scalene triangle cases, and three to get on isosceles triangle. The test cases are given in Table 8.5.

Table 8.5: Test cases of triangle problem using decision table

<i>Test case</i>	<i>x</i>	<i>y</i>	<i>z</i>	<i>Expected Output</i>
1	4	1	2	Not a triangle
2	1	4	2	Not a triangle
3	1	2	4	Not a triangle
4	5	5	5	Equilateral
5	?	?	?	Impossible
6	?	?	?	Impossible
7	2	2	3	Isosceles
8	?	?	?	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

Example 8.11

Consider a program for the determination of Previous date. Its input is a triple of day, month and year with the values in the range

$$1 \leq \text{month} \leq 12$$

$$1 \leq \text{day} \leq 31$$

$$1900 \leq \text{year} \leq 2025$$

The possible outputs are “Previous date” and “Invalid date”. Design the test cases using decision table based testing.

Solution

The input domain can be divided into following classes:

- $I_1 = \{M_1: \text{month has 30 days}\}$
 - $I_2 = \{M_2: \text{month has 31 days except March, August and January}\}$
 - $I_3 = \{M_3: \text{month is March}\}$
 - $I_4 = \{M_4: \text{month is August}\}$
 - $I_5 = \{M_5: \text{month is January}\}$
 - $I_6 = \{M_6: \text{month is February}\}$
 - $I_7 = \{D_1: \text{day} = 1\}$
 - $I_8 = \{D_2: 2 \leq \text{day} \leq 28\}$
 - $I_9 = \{D_3: \text{day} = 29\}$
 - $I_{10} = \{D_4: \text{day} = 30\}$
 - $I_{11} = \{D_5: \text{day} = 31\}$
 - $I_{12} = \{Y_1: \text{year is a leap year}\}$
 - $I_{13} = \{Y_2: \text{year is a common year}\}$

The decision table is given below:

The number of test cases are equal to number of columns of the decision table. Hence 60 test cases can be generated. Here input domain is partitioned into thirteen classes. Class identification is somewhat tricky and depends on understanding of the problem. We have separate classes for March, August, January and February. Justification is given below:

- (i) **March:** We have to find previous date. If present date is first March, the previous date may be 28 or 29 depending upon the present year. Hence separate treatment is required for March.
- (ii) **August:** August is a month of 31 days and its previous month July is also of 31 days. This is a typical situation. Normally, 31 days months have previous month of 30 days. In order to accommodate this, August has a separate status.
- (iii) **January:** First January Present date will have 31st December as Previous date. It is similar to August month situation except decrementing the year.
- (iv) **February:** Due to less number of days & leap year situation, February has special features and is required as a separate class.

Similary, separate classes to handle 29, 30 and 31 days have been created. First day of every month has a separate class in order to get previous month's last day. The 60 test cases are given below:

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
1	June	1	1964	31 May, 1964
2	June	1	1962	31 May, 1962
3	June	15	1964	14 June, 1964
4	June	15	1962	14 June, 1962
5	June	29	1964	28 June, 1964
6	June	29	1962	28 June, 1962
7	June	30	1964	29 June, 1964
8	June	30	1962	29 June, 1962
9	June	31	1964	Impossible
10	June	31	1962	Impossible
11	May	1	1964	30 April, 1964
12	May	1	1962	30 April, 1962
13	May	15	1964	14 May, 1964
14	May	15	1962	14 May, 1962
15	May	29	1964	28 May, 1964
16	May	29	1962	28 May, 1962
17	May	30	1964	29 May, 1964
18	May	30	1962	29 May, 1962
19	May	31	1964	30 May, 1964
20	May	31	1962	30 May, 1962

(Contd.)...

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
21	March	1	1964	29 February, 1964
22	March	1	1962	28 February, 1962
23	March	15	1964	14 March, 1964
24	March	15	1962	14 March, 1962
25	March	29	1964	28 March, 1964
26	March	29	1962	28 March, 1962
27	March	30	1964	29 March, 1964
28	March	30	1962	29 March, 1962
29	March	31	1964	30 March, 1964
30	March	31	1962	30 March, 1962
31	August	1	1964	31 July, 1964
32	August	1	1962	31 July, 1962
33	August	15	1964	14 August, 1964
34	August	15	1962	14 August, 1962
35	August	29	1964	28 August, 1964
36	August	29	1962	28 August, 1962
37	August	30	1964	29 August, 1964
38	August	30	1962	29 August, 1962
39	August	31	1964	30 August, 1964
40	August	31	1962	30 August, 1962
41	January	1	1964	31 December, 1963
42	January	1	1962	31 December, 1961
43	January	15	1964	14 January, 1964
44	January	15	1962	14 January, 1962
45	January	29	1964	28 January, 1964
46	January	29	1962	28 January, 1962
47	January	30	1964	29 January, 1964
48	January	30	1962	29 January, 1962
49	January	31	1964	30 January, 1964
50	January	31	1962	30 January, 1962
51	February	1	1964	31 January, 1964
52	February	1	1962	31 January, 1962
53	February	15	1964	14 February, 1964
54	February	15	1962	14 February, 1962
55	February	29	1964	28 February, 1964

(Contd.)...

<i>Test case</i>	<i>Month</i>	<i>Day</i>	<i>Year</i>	<i>Expected output</i>
56	February	29	1962	Impossible
57	February	30	1964	Impossible
58	February	30	1962	Impossible
59	February	31	1964	Impossible
60	February	31	1962	Impossible

8.3.4 Cause Effect Graphing Technique

One weakness of boundary value analysis and equivalence partitioning is that these do not explore combinations of input circumstances. These consider only single input conditions. However, combinations of inputs may result in interesting situations. These situations should be tested. If we consider all valid combinations of equivalence classes, then we will have large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are " n " different input conditions, such that any combination of the input conditions is valid, we will have 2^n test cases [JALO96].

Cause effect graphing [ELME73] is a technique that aids in selecting, in a systematic way, a high-yield set of test cases. It has a beneficial effect in pointing out incompleteness and ambiguities in the specifications. The following process is used to derive test cases [MYER79].

1. The causes & effects in the specifications are identified. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation (a lingering effect that an input has on the state of the program or system). For instance, if a transaction to a program causes a master file to be updated, the alteration to the master file is a system transformation; a confirmation message would be an output condition. Causes and effects are identified by reading the specification word by word and underlining words or phrases that describe causes & effects. Each cause & effect is assigned a unique number.
2. The semantic content of the specification is analysed and transformed into a Boolean graph linking the causes and effects. This is the cause effect graph.
3. The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.
4. By methodically tracing state conditions in the graph, the graph is converted into a limited entry decision table. Each column in the table represents a test case.
5. The columns in the decision table are converted into test cases.

The basic notation for the graph is shown in Fig. 8.8.

Think of each node as having the value 0 or 1; 0 represents the 'absent state' and 1 represents the present state. The identity function states that if c_1 is 1, e_1 is 1; else e_1 is 0. The NOT function states that if c_1 is 1, e_1 is 0 else, e_1 is 1. The OR function states that if c_1 or c_2 or c_3 is 1, e_1 is 1; else e_1 is 0. The AND function states that if both c_1 and c_2 are 1, e_1 is 1; else e_1 is 0. The AND and OR functions are allowed to have any number of inputs.

Myers [MYER79] explained this effectively with the following example. "The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the

file update is made. If the character in column 1 is incorrect, message x is issued. If the character in column 2 is not a digit, message y is issued".

The causes are

- c_1 : character in column1 is A
- c_2 : character in column 1 is B
- c_3 : character in column 2 is a digit

and the effects are

- e_1 : update made
- e_2 : message x is issued
- e_3 : message y is issued

The cause-effect graph is shown in Fig. 8.9.

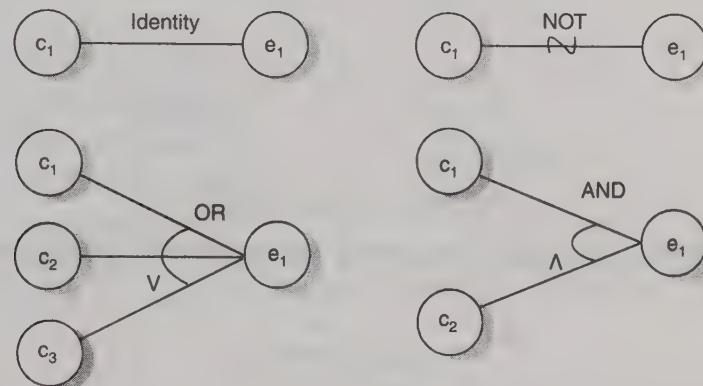


Fig. 8.8: Basic cause effect graph symbols

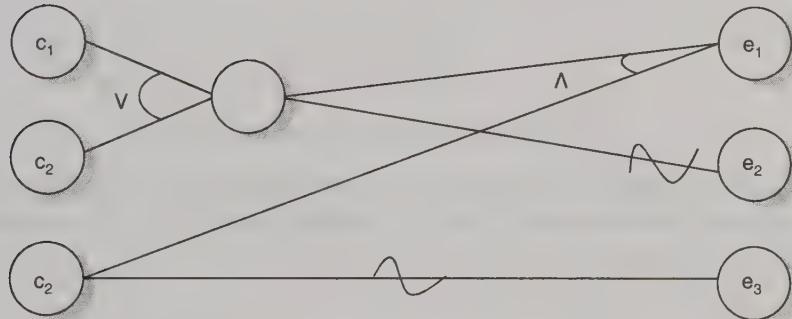


Fig. 8.9: Sample cause effect graph.

Although the graph in Fig. 8.9 represents the specification, it does not contain an impossible combination of causes—it is impossible for both causes c_1 or c_2 to be set to 1 simultaneously. In most programs, certain combinations of causes are impossible because of syntactic or environmental considerations. To account for these, the notations in Fig. 8.10 is used. The **E** constraint states that it must always be true that at most one of c_1 or c_2 can be 1 (c_1 or c_2 cannot be 1 simultaneously). The **I** constraint states that at least one of c_1 , c_2 and c_3 must always be 1.

c_1, c_2 and c_3 cannot be 0 simultaneously). The O constraint states that one, and only one, of c_1 and c_2 must be 1. The R constraint states that, for c_1 to be 1, c_2 must be 1 (*i.e.*, it is impossible for c_1 to be 1 and c_2 to be 0).

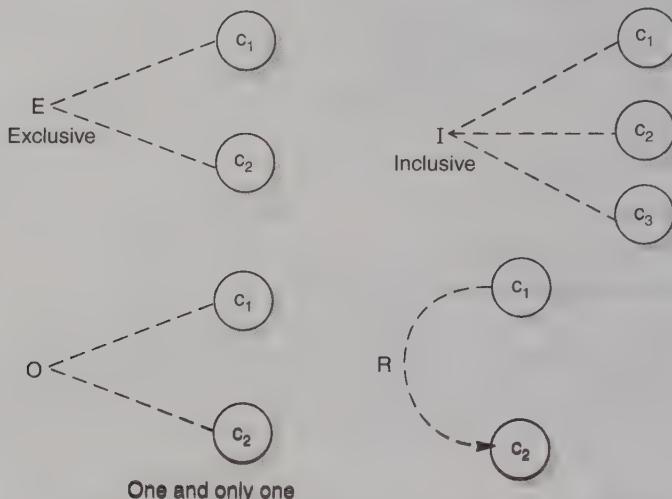


Fig. 8.10: Constraint symbols

There is frequently a need for a constraint among effects. The M constraint in Fig. 8.11 states that if effect e_1 is 1, effect e_2 is forced to be 0.

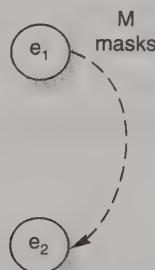


Fig. 8.11: Symbol for masks constraint

Returning to the simple example above, we see that it is physically impossible for causes c_1 and c_2 to be present simultaneously, but it is possible for neither to be present. Hence they are linked with E constraint as shown in Fig. 8.12.

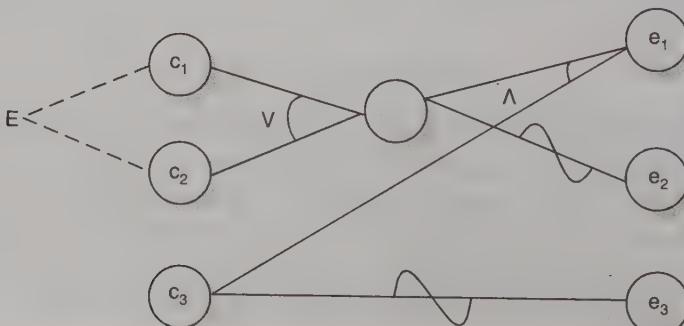


Fig. 8.12: Sample cause effect graph with exclusive constraint

Example 8.12

Consider the triangle problem specified in the example 8.3. Draw the Cause-effect graph and identify the test cases.

Solution

The causes are

- c_1 : side x is less than sum of sides y and z
- c_2 : side y is less than sum of sides x and z
- c_3 : side z is less than sum of sides x and y
- c_4 : side x is equal to side y
- c_5 : side x is equal to side z
- c_6 : side y is equal to side z

and effects are

- e_1 : Not a triangle
- e_2 : Scalene triangle
- e_3 : Isosceles triangle
- e_4 : Equilateral triangle
- e_5 : Impossible stage

The cause effect graph is shown in Fig. 8.13 and decision table is shown in table 8.6. The test cases for this problem are available in table 8.5.

Table 8.6: Decision table

Conditions	0	1	1	1	1	1	1	1	1	1	1
$c_1 : x < y + z ?$	0	1	1	1	1	1	1	1	1	1	1
$c_2 : y < x + z ?$	X	0	1	1	1	1	1	1	1	1	1
$c_3 : z < x + y ?$	X	X	0	1	1	1	1	1	1	1	1
$c_4 : x = y ?$	X	X	X	1	1	1	1	0	0	0	0
$c_5 : x = z ?$	X	X	X	1	1	0	0	1	1	0	0
$c_6 : y = z ?$	X	X	X	1	0	1	0	1	0	1	0
$e_1 : \text{Not a triangle}$	1	1	1								
$e_2 : \text{Scalene}$											1
$e_3 : \text{Isosceles}$							1		1	1	
$e_4 : \text{Equilateral}$				1							
$e_5 : \text{Impossible}$					1	1		1			

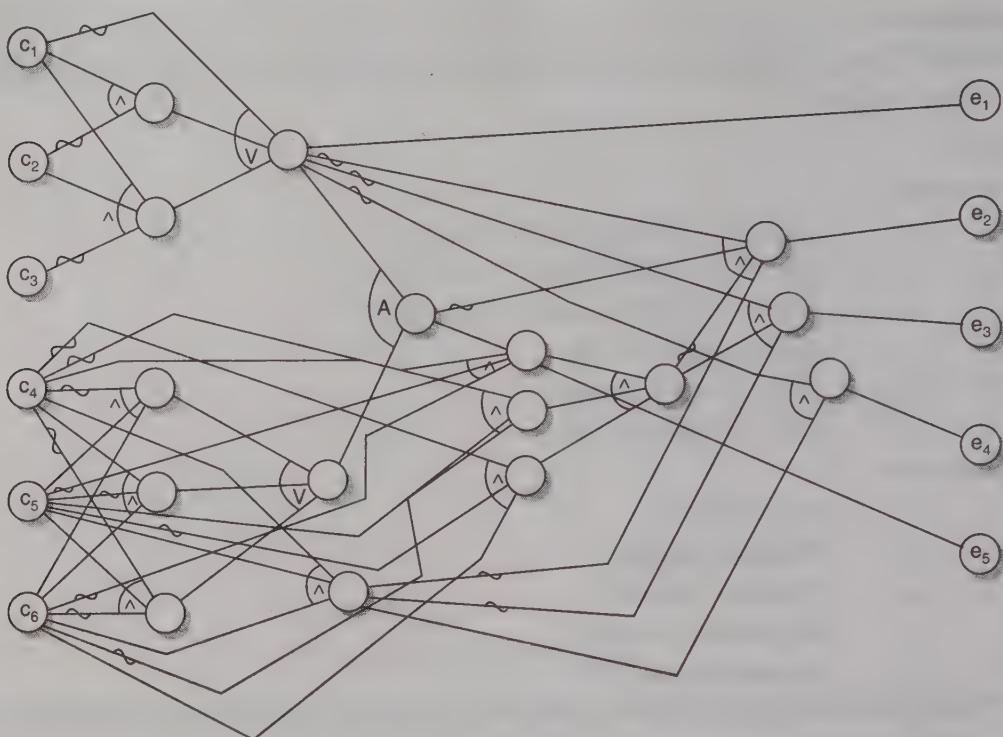


Fig. 8.13: Cause effect graph of triangle problem

Cause-effect graphing is a systematic method of generating test cases representing combinations of conditions. The alternative would be an adhoc selection of combinations, but in doing so it is likely that one would overlook many of the interesting test cases identified by the cause effect graph. Although, it does produce a set of useful test cases, it normally does not produce all of the useful test cases that might be identified.

8.3.5 Special Value Testing

It is probably the most widely practiced form of functional testing. It is the most intuitive and the least uniform type of testing. Special value testing occurs when a tester uses his or her domain knowledge, experience with similar programs and information about “Soft spots” to devise test cases. We may call this as adhoc testing.

Now guidelines are used other than to use “best engineering judgement”. As a result, special value testing is heavily dependent on the abilities of the testing persons.

8.4 STRUCTURAL TESTING

A complementary approach to functional testing is called structural/white box testing. It permits us to examine the internal structure of the program. In using this strategy, we derive test cases from an examination of the program’s logic. We do not pay any attention to specifications. For instance, if the first statement of the code is “if ($x \leq 100$)”, then we may try testing the program with a test case of 100.

Therefore, the knowledge to the internal structure of the code can be used to find the number of test cases required to guarantee a given level of test coverage. It would never be advisable to release a software which contained untested statements and the consequences of which might be disastrous. This goal seems to be easy, but simple objectives of structural testing are harder to achieve than may appear at first glance.

In functional testing, all specifications are being checked against the implementation, so why do we really require structural testing? It seems to be necessary because there might be parts of the code, which are not fully exercised by the functional tests. There may also be sections of the code, which are surplus to requirements. That is to say, we have checked the program against the functional tests and no errors are revealed, then further inspection by structural tests reveals a piece of code that is not even needed by the specifications and hence not examined by functional testing. This can be regarded as an error since it is a deviation from requirements. It may find those errors, which have been missed by functional testing [NORM89].

We want to look in to the program, examine the code and watch it as it runs. This activity is dynamic and is about testing a running program; therefore it is called dynamic white box testing. If we want to test the program without running it; meaning thereby examining and reviewing it; then it is called static white box testing.

Hence, static white box testing is the process of carefully and methodically reviewing the software design, architecture, or code for bugs without executing it. It is sometimes referred to as structural analysis [PATT01].

8.4.1 Path Testing

Path testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement is executed at least once. It is most applicable to new software for module testing or unit testing. It requires complete knowledge of the program's structure and is used by developers to unit test their own code. The effectiveness of path testing rapidly deteriorates as the size of the software under test increases. It is rarely, if ever, used for system testing. For the developer, it is the basic test technique [BEIZ90].

This type of testing involves:

1. generating a set of paths that will cover every branch in the program.
2. finding a set of test cases that will execute every path in this set of program paths.

The two steps are not necessarily executed in sequence. Path generation (*i.e.*, step 1) can be performed through the static analysis of the program control flow and can be automated.

Flow graph

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represent flow of control. If i & j are nodes in the program graph, there is an edge from node i to node j if the statement (fragment) corresponding to node j can be executed immediately after the statement (fragment) corresponding to node i .

A flow graph can easily be generated from the code of any problem. The basic constructs of flow graph are given in Fig. 8.14.

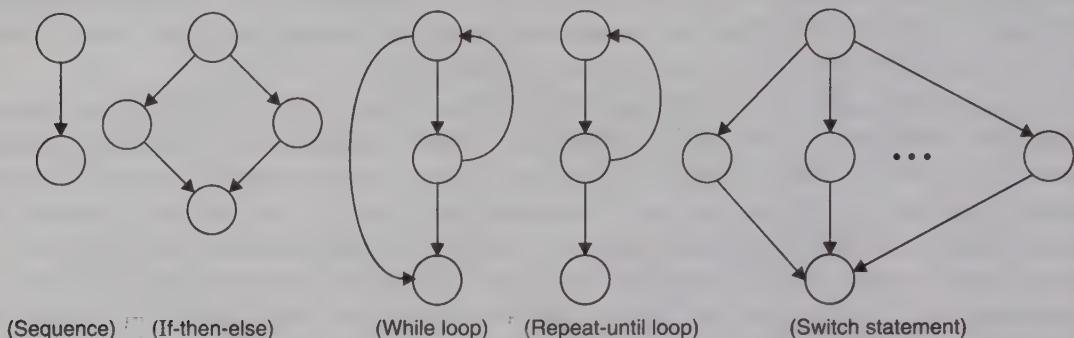


Fig. 8.14: The basic construct of the flow graph

We consider a program that generates the previous date, if a date is given as an input (for details refer Example 8.2) which is given in Fig. 8.15. Here line numbers are statements or fragments of statement. It is a matter of choice to make fragment as a separate node or to include fragments in other portion of a statement.

Our first step is to prepare a flow graph from the code. The flow graph of previous date program (given in Fig. 8.15) is generated and is given in Fig. 8.16. Such a flow graph helps us to understand the flow of control from source to destination. We want to find paths from this control flow and may like to execute every path during testing.

```
/* Program to generate the previous date given a date, assumes data
given as dd mm yyyy separated by space and performs error checks on the
validity of the current date entered. */
```

```
#include <stdio.h>
#include <conio.h>

1 int main()
2 {
3     int day, month, year, validDate = 0;
4     /*Date Entry*/
5     printf("Enter the day value: ");
6     scanf("%d", &day);
7     printf("Enter the month value: ");
8     scanf("%d", &month);
9     printf("Enter the year value: ");
10    scanf("%d", &year);
11    /*Check Date Validity */
12    if (year >= 1900 && year <= 2025) {
13        if (month == 1 || month == 3 || month == 5 || month == 7 ||
14            month == 8 || month == 10 || month == 12) {
```

(Contd.)...

```
12         if (day >= 1 && day <= 31) {
13             validDate = 1;
14         }
15         else {
16             validDate = 0;
17         }
18     }
19     else if (month == 2) {
20         int rVal=0;
21         if (year%4 == 0) {
22             rVal=1;
23             if ((year%100)==0 && (year % 400) !=0) {
24                 rVal=0;
25             }
26         }
27         if (rVal ==1 && (day >=1 && day <=29) ) {
28             validDate = 1;
29         }
30         else if (day >=1 && day <= 28 ) {
31             validDate = 1;
32         }
33         else {
34             validDate = 0;
35         }
36     }
37     else if ((month >= 1 && month <= 12) && (day >= 1 && day <= 30)) {
38         validDate = 1;
39     }
40     else {
41         validDate = 0;
42     }
43 }
44 /*Prev Date Calculation*/
45 if (validDate) {
46     if (day == 1) {
47         if (month == 1) {
48             year--;
49             day=31;
50             month=12;
51         }
52         else if (month == 3) {
53             int rVal=0;
```

(Contd.)...

```
53         if (year%4 == 0) {
54             rVal=1;
55             if ((year%100)==0 && (year % 400) !=0) {
56                 rVal=0;
57             }
58             }
59             if (rVal ==1) {
60                 day=29;
61                 month--;
62             }
63             else {
64                 day=28;
65                 month--;
66             }
67         }
68         else if (month == 2 || month == 4 || month == 6 || month == 9 || month == 11) {
69             day = 31;
70             month--;
71         }
72         else {
73             day=30;
74             month--;
75         }
76     }
77     else {
78         day--;
79     }
80     printf("The next date is: %d-%d-%d",day,month,year);
81 }
82 else {
83     printf("The entered date ( %d-%d-%d ) is invalid",day,month, year);
84 }
85 getch();
86 return 1;
87 }
```

Fig. 8.15: Program for previous date problem

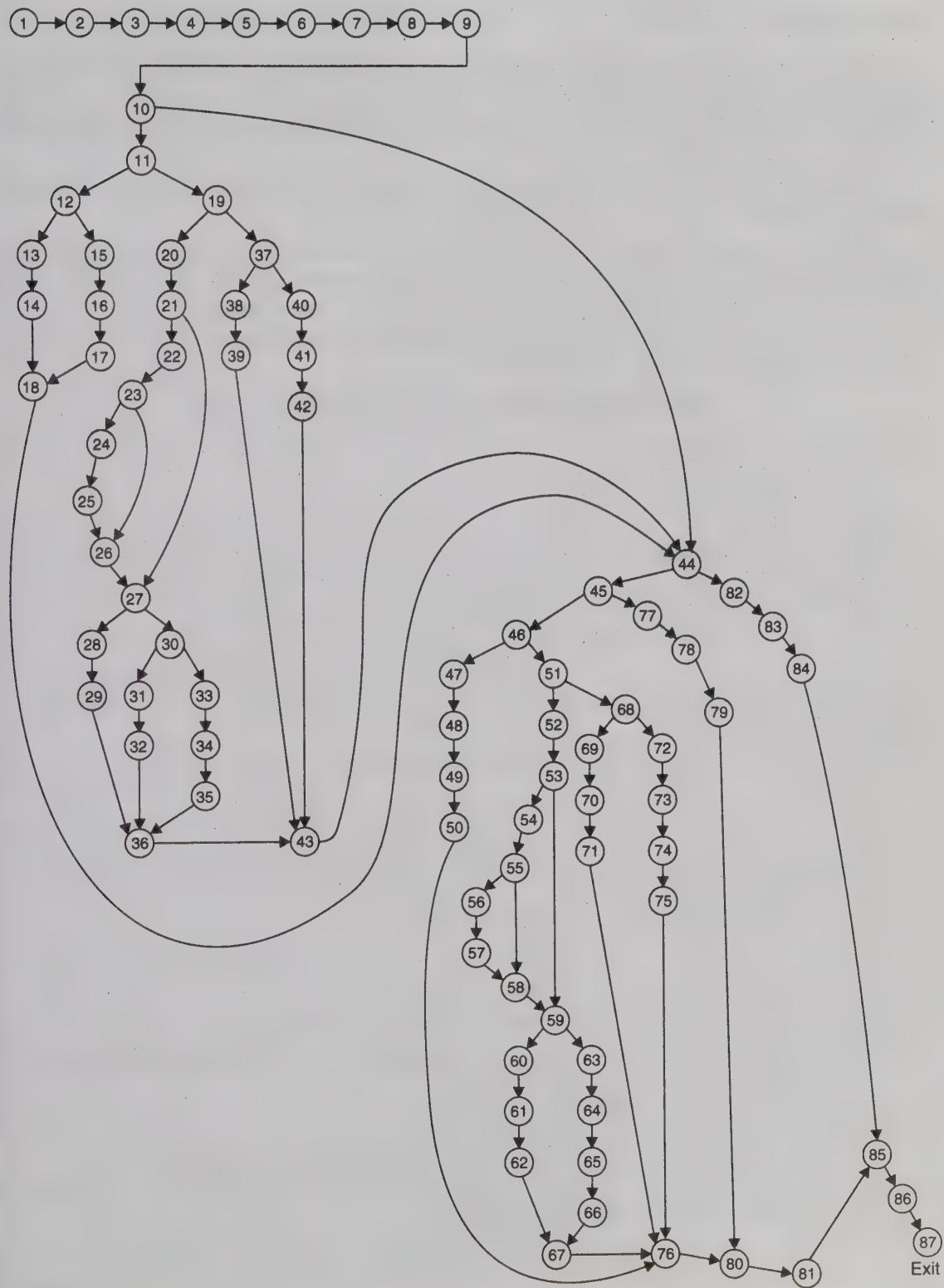


Fig. 8.16: Flow graph of previous date problem

DD path graph

As we know, flow graph generation is the first step of path testing. The second step is to draw a DD path graph from the flow graph. The DD Path graph is known as decision to decision path graph. Here, we concentrate only on decision nodes. The nodes of flow graph, which are in a sequence are combined into a single node.

Hence, DD Path graph is a directed graph in which nodes are sequences of statements and edges represent control flow between nodes.

In order to understand the concept correctly, we consider the previous date generation program, as given in Fig. 8.15 with its flow graph as given in Fig. 8.16. The nodes numbered from 1 to 9 are combined together to form a new node, say n_1 due to sequential flow. If we enter into node 1, we will only be allowed to come out of node 9. Hence, flow graph of Fig. 8.16 can be converted into DD Path graph using Table 8.7.

Table 8.7: Mapping of flow graph nodes and DD path graph nodes

<i>Flow graph nodes</i>	<i>DD Path graph corresponding node</i>	<i>Remarks</i>
1 to 9	n_1	There is a sequential flow from node 1 to 9.
10	n_2	Decision node, if true goto 13 else goto 44.
11	n_3	Decision node, if true goto 12 else goto 19.
12	n_4	Decision node, if true goto 13 else goto 15.
13, 14	n_5	Sequential nodes and are combined to form new node n_5 .
15, 16, 17	n_6	Sequential nodes.
18	n_7	Edges from node 14 and 17 are terminated here.
19	n_8	Decision node, if true goto 20 else goto 37.
20	n_9	Intermediate node with one input edge and one output edge.
21	n_{10}	Decision node, if true goto node 22 else, goto node 27
22	n_{11}	Intermediate node
23	n_{12}	Decision node, if true goto node 24, else node 26.
24, 25	n_{13}	Sequential nodes
26	n_{14}	Two edges from node 25 & 23 are terminated here.
27	n_{15}	Two edges from node 26 & 21 are terminated here. Also a decision node.
28, 29	n_{16}	Sequential nodes.
30	n_{17}	Decision node, if true goto 31, else goto 33.
31, 32	n_{18}	Sequential nodes
33, 34, 35	n_{19}	Sequential nodes

(Contd.)...

<i>Flow graph nodes</i>	<i>DD path graph corresponding node</i>	<i>Remarks</i>
36	n_{20}	Three edges from nodes, 29, 32 and 35 are terminated here.
37	n_{21}	Decision node, if true goto 38 else goto 40.
38, 39	n_{22}	Sequential nodes
40, 41, 42	n_{23}	Sequential nodes
43	n_{24}	Three edges from nodes 36, 39, and 42 are terminated here.
44	n_{25}	Decision node if true goto 45 else 82. Three edges from 18, 43 & 10 are also terminated here.
45	n_{26}	Decision node, if true goto 46 else goto 77.
46	n_{27}	Decision node, if true goto 47 else goto 51.
47, 48, 49, 50	n_{28}	Sequential nodes
51	n_{29}	Decision node, if true goto 52 else goto 68.
52	n_{30}	Intermediate node with one input edge & one output edge.
53	n_{31}	Decision node, if true goto 54 else goto 59.
54	n_{32}	Intermediate node
55	n_{33}	Decision node if true goto 56 else goto 58.
56, 57	n_{34}	Sequential nodes
58	n_{35}	Two edges from nodes 57 and 55 are terminated here.
59	n_{36}	Decision node, if true goto 60 else goto 63. Two edges from nodes 58 and 53 are terminated.
60, 61, 62	n_{37}	Sequential nodes
63, 64, 65, 66	n_{38}	Sequential nodes
67	n_{39}	Two edges from node 62 and 66 are terminated here.
68	n_{40}	Decision node, if true goto 69 else goto 72.
69, 70, 71	n_{41}	Sequential nodes
72, 73, 74, 75	n_{42}	Sequential nodes
76	n_{43}	Four edges from nodes 50, 67, 71 and 75 are terminated here.
77, 78, 79	n_{44}	Sequential nodes
80	n_{45}	Two edges from nodes 76 and 79 are terminated.
81	n_{46}	Intermediate node
82, 83, 84	n_{47}	Sequential nodes
85	n_{48}	Two edges from nodes 81 and 84 are terminated here.
86, 87	n_{49}	Sequential nodes with exit node.

The DD Path graph is given in Fig. 8.17.

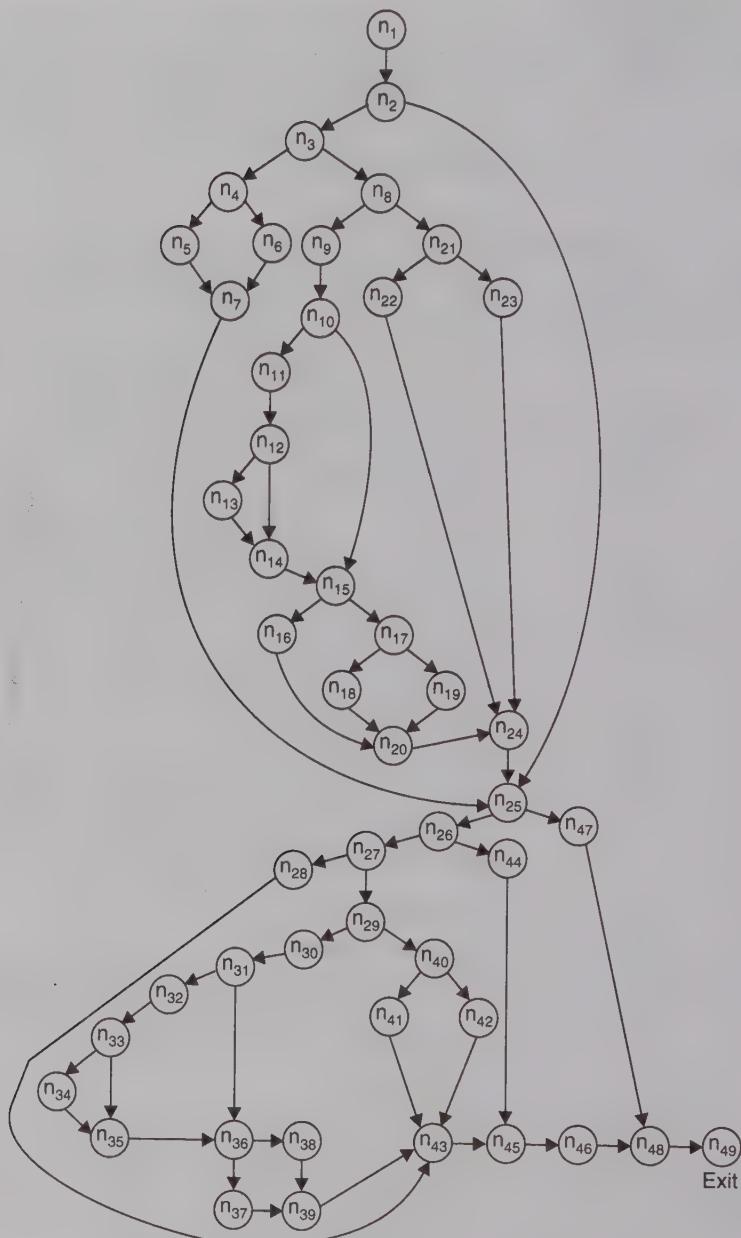


Fig. 8.17: DD path graph of previous date problem

Independent paths

The DD Path graph is used to find independent paths. We are interested to execute all independent paths at least once during path testing.

An independent path is any path through the DD Path graph that introduces at least one new set of processing statements or new conditions. Therefore, an independent path must move along at least one edge that has not been traversed before the path is defined.

We consider the previous date problem and its DD Path graph which is given in Fig. 8.17. The independent paths are found and are given in Fig. 8.18. There are 18 independent paths.

<i>Independent paths of previous date problem</i>	
1	$n_1, n_2, n_{25}, n_{47}, n_{48}, n_{49}$
2	$n_1, n_2, n_3, n_4, n_5, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
3	$n_1, n_2, n_3, n_4, n_6, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
4	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
5	$n_1, n_2, n_3, n_8, n_{21}, n_{23}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
6	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{19}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
7	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
8	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
9	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
10	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
11	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{44}, n_{45}, n_{46}, n_{48}, n_{49}$
12	$n_1, n_2, n_3, n_8, n_9, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{28}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
13	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{41}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
14	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{42}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
15	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{38}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
16	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
17	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{34}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
18	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$

Fig. 8.18: Independent paths of previous date problem.

It is quite interesting to use independent paths in order to ensure that

- (i) Every statement in the program has been executed at least once.
- (ii) Every branch has been exercised for true and false conditions.

There are high quality commercial tools that generate the DD Path graph of a given program. The vendors make sure that the products work for wide variety of programming languages. In practice, it is reasonable to make DD Path graphs for programs upto about 100 source lines. Beyond that, we should go for a standard tool.

Example 8.13

Consider the program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a, b, c) and values may be from interval [0, 100].

The program is given in Fig. 8.19. The output may have one of the following words:

[Not a quadratic equation; real roots; Imaginary roots; Equal roots]

Draw the flow graph and DD Path graph. Also find independent paths from the DD Path graph.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
1     int main()
2     {
3         int a,b,c,validInput=0,d;
4         double D;
5         printf("Enter the 'a' value: ");
6         scanf("%d",&a);
7         printf("Enter the 'b' value: ");
8         scanf("%d",&b);
9         printf("Enter the 'c' value: ");
10        scanf("%d",&c);
11        if ((a >= 0) && (a <= 100) && (b >= 0) && (b <= 100) && (c >= 0)
12            && (c <= 100)) {
13            validInput = 1;
14            if (a == 0) {
15                validInput = -1;
16            }
17            if (validInput==1) {
18                d = b*b - 4*a*c;
19                if (d == 0) {
20                    printf("The roots are equal and are r1 = r2 = %f\n",
21                        -b/(2*(float) a));
22                }
23                else if ( d > 0 ) {
24                    D=sqrt(d);
25                    printf("The roots are real and are r1 = %f and r2 = %f\n",
26                        (-b-D)/(2* a), (-b+D)/(2* a));
27                }
28                else {
29                    D=sqrt(-d)/(2*a);
30                    printf("The roots are imaginary and are r1 = (%f,%f) and
31                        r2 = (%f,%f)\n", -b/(2.0*a),D,-b/(2.0*a),-D);
32                }
33            else if (validInput == -1) {
```

(Contd.)...

```
32     printf("The values do not constitute a Quadratic equation.");
33 }
34 else {
35     printf("The inputs belong to invalid range.");
36 }
37 getch();
38 return 1;
39 }
```

Fig. 8.19: Code of quadratic equation problem

Solution

The flow graph is given below:

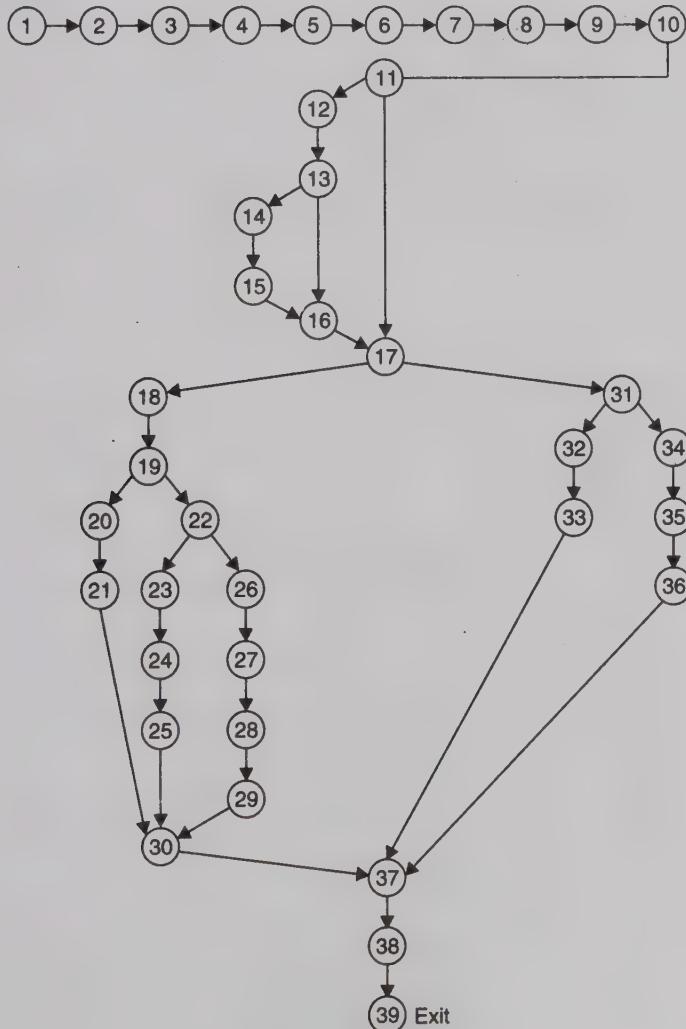


Fig. 8.19(a): Program flow graph.

DD Path graph is given below :

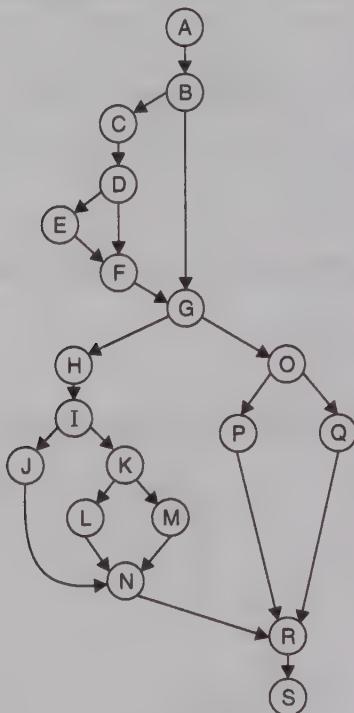


Fig. 8.19(b): DD Path graph

The mapping table for DD path graph is:

<i>Flow graph nodes</i>	<i>DD Path graph corresponding nodes</i>	<i>Remarks</i>
1 to 10	A	Sequential nodes
11	B	Decision node
12	C	Intermediate node
13	D	Decision node
14, 15	E	Sequential nodes
16	F	Two edges are combined here
17	G	Two edges are combined and decision node
18	H	Intermediate node
19	I	Decision node
20, 21	J	Sequential node
22	K	Decision node
23, 24, 25	L	Sequential nodes
26, 27, 28, 29	M	Sequential nodes

(Contd.)...

30	N	Three edges are combined
31	O	Decision node
32, 33	P	Sequential nodes
34, 35, 36	Q	Sequential nodes
37	R	Three edges are combined
38, 39	S	Sequential nodes with exit node.

Independent paths are:

- (i) ABGOQRS
- (ii) ABGOPRS
- (iii) ABCDFGQQRS
- (iv) ABCDEFGOPRS
- (v) ABGHIJNRS
- (vi) ABGHIKLNRS
- (vii) ABGHIKMNRS

Example 8.14

Consider a program given in Fig. 8.20 for the classification of a triangle. Its input is a triple of positive integers (say, a, b, c) from the interval [1, 100]. The output may be [Scalene, Isosceles, Equilateral, Not a triangle].

Draw the flow graph & DD Path graph. Also find the independent paths from the DD Path graph.

```

1 #include <stdio.h>
2 #include <conio.h>
3 int main()
4 {
5     int a,b,c,validInput=0;
6     printf("Enter the side 'a' value: ");
7     scanf("%d",&a);
8     printf("Enter the side 'b' value: ");
9     scanf("%d",&b);
10    printf("Enter the side 'c' value:");
11    scanf("%d",&c);
12    if ((a > 0) && (a <= 100) && (b > 0) && (b <= 100) && (c > 0)
13        && (c <= 100)) {
14        if ((a + b) > c) && ((c + a) > b) && ((b + c) > a)) {
15            validInput = 1;
16        }
17    }
18    else {
19        validInput = -1;
20    }
21    If (validInput==1) {
22        If ((a==b) && (b==c)) {
23            printf("The triangle is equilateral");
24        }
25        else if ((a == b) || (b == c) || (c == a) ) {

```

(Contd.)...

```

23         printf("The triangle is isosceles");
24     }
25     else {
26         printf("The triangle is scalene");
27     }
28 }
29 else if (validInput == 0) {
30     printf("The values do not constitute a Triangle");
31 }
32 else {
33     printf("The inputs belong to invalid range");
34 }
35 getch();
36 return 1;
37 }

```

Fig. 8.20: Code of triangle classification problem.

Flow graph of triangle problem is:

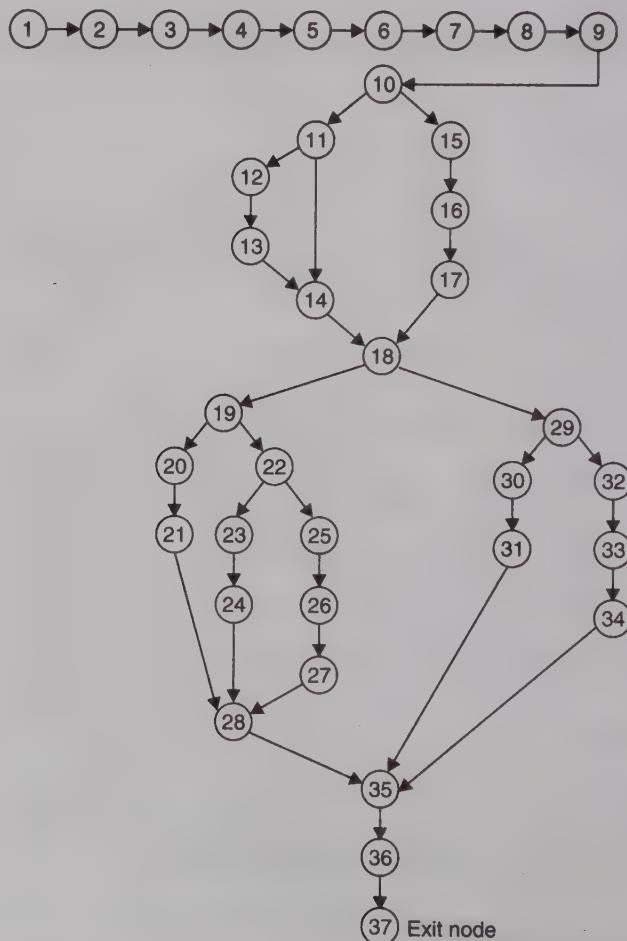


Fig. 8.20(a): Program flow graph

The mapping table for DD path graph is:

Flow graph nodes	DD Path graph corresponding nodes	Remarks
1 to 9	A	Sequential nodes
10	B	Decision node
11	C	Decision node
12, 13	D	Sequential nodes
14	E	Two edges are joined here
15, 16, 17	F	Sequential nodes
18	G	Decision nodes plus joining of two edges
19	H	Decision node
20, 21	I	Sequential nodes
22	J	Decision node
23, 24	K	Sequential nodes
25, 26, 27	L	Sequential nodes
28	M	Three edges are combined here
29	N	Decision node
30, 31	O	Sequential nodes
32, 33, 34	P	Sequential nodes
35	Q	Three edges are combined here
36, 37	R	Sequential nodes with exit node

DD Path graph is given in Fig. 8.20 (b).

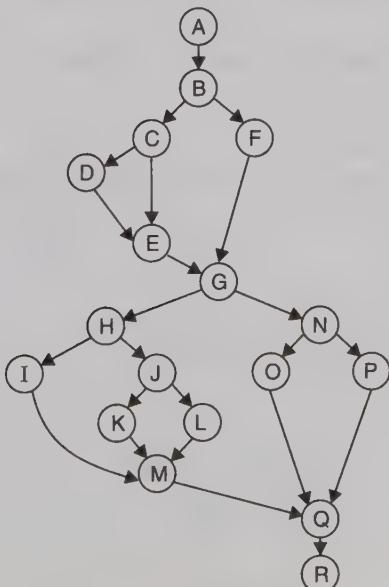


Fig. 8.20(b): DD path graph

Independent paths are:

- | | |
|------------------|-----------------|
| (i) ABFGNPQR | (ii) ABFGNOQR |
| (iii) ABCEGNPQR | (iv) ABCDEGNOQR |
| (v) ABFGHIMQR | (vi) ABFGHJKMQR |
| (vii) ABFGHJLMQR | |

8.4.2 Cyclomatic Complexity

The cyclomatic complexity is also known as structural complexity because it gives internal view of the code. This approach is used to find the number of independent paths through a program. This provides us the upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once and every condition has been executed on its true and false side. If a program has backward branch then it may have infinite number of paths. Although it is possible to define a set of algebraic expressions that gives the total number of possible paths through a program, however, using total number of paths has been found to be impractical. Because of this, the complexity measure is defined in terms of independent paths—that when taken in combination will generate every possible path [MCCA76]. An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

McCabe's cyclomatic metric [MCCA76] $V(G)$ of a graph G with n vertices, e edges, and P connected components is $V(G) = e - n + 2P$.

Given a program we will associate with it a directed graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is classically known as flow graph and it is assumed that each node can be reached by the entry node and each node can reach the exit node. For example, a flow graph shown in Fig. 8.21 with entry node 'a' and exit node 'f'.

The value of cyclomatic complexity can be calculated as

$$V(G) = 9 - 6 + 2 = 5$$

Here $e = 9$, $n = 6$ and $P = 1$

There will be five independent paths for the flow graph illustrated in Fig. 8.21.

- path 1 : $a c f$
- path 2 : $a b e f$
- path 3 : $a d c f$
- path 4 : $a b e a c f$ or $a b e a b e f$
- path 5 : $a b e b e f$

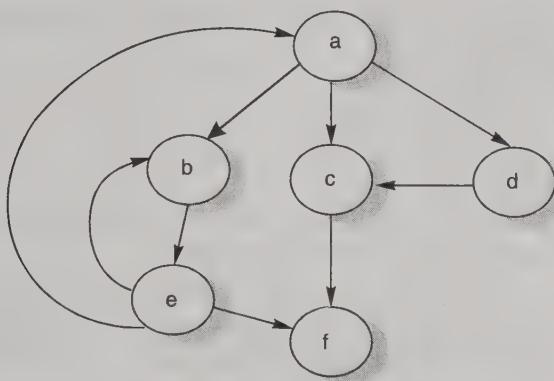


Fig. 8.21

Notice that the sequence of an arbitrary number of nodes always has unit complexity and that cyclomatic complexity conforms to our intuitive notion of minimum number of paths. Several properties of cyclomatic complexity are stated below:

1. $V(G) \geq 1$
2. $V(G)$ is the maximum number of independent paths in graph G .
3. Inserting & deleting functional statements to G does not affect $V(G)$.
4. G has only one path if and only if $V(G) = 1$.
5. Inserting a new row in G increases $V(G)$ by unity.
6. $V(G)$ depends only on the decision structure of G .

The role of P in the complexity calculation $V(G) = e - n + 2P$ is required to be understood correctly. We define a flow graph with unique entry and exit nodes, all nodes reachable from the entry, and exit reachable from all nodes. This definition would result in all flow graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having a flow graph shown in Fig. 8.22.

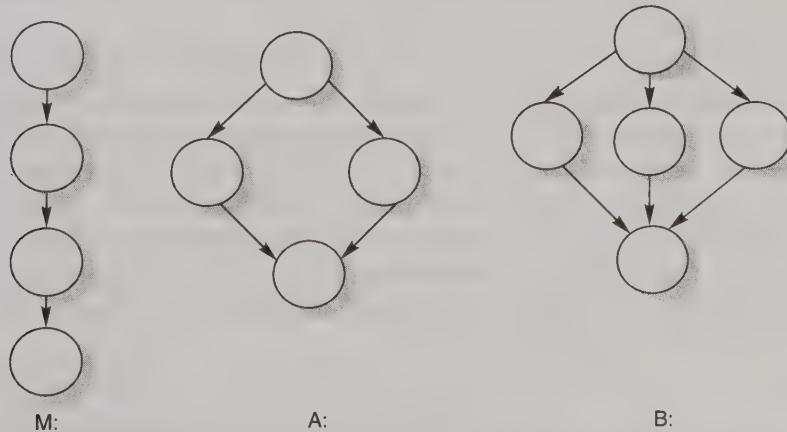


Fig. 8.22

Let us denote the total graph above with 3 connected components as $M \cup A \cup B$. Since $P = 3$, we calculate complexity as

$$\begin{aligned} V(M \cup A \cup B) &= e - n + 2P \\ &= 13 - 13 + 2 * 3 \\ &= 6 \end{aligned}$$

This method with $P \neq 1$ can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines.

Notice that $V(M \cup A \cup B) = V(M) + V(A) + V(B) = 6$. In general, the complexity of a collection C of flow graphs with K connected components is equal to the summation of their complexities. To see this let C_i , $1 \leq i \leq K$ denote the k distinct connected component, and let e_i and n_i be the number of edges and nodes in the i th-connected component. Then

$$\begin{aligned} V(C) &= e - n + 2p = \sum_{i=1}^k e_i - \sum_{i=1}^k n_i + 2K \\ &= \sum_{i=1}^k (e_i - n_i + 2) = \sum_{i=1}^k V(C_i) \end{aligned}$$

Since the calculation $V = e - n + 2P$ can be quite tedious for a developer, an effort has been made to simplify the complexity calculations. Two alternate methods are available for the complexity calculations.

1. Cyclomatic complexity $V(G)$ of a flow graph G is equal to the number of predicate (decision) nodes plus one [MILL72].

$$V(G) = \Pi + 1$$

Where Π is the number of predicate nodes contained in the flow graph G .

The only restriction is that every predicate node should have two outgoing edges *i.e.*, one for “true” condition & another for “false” condition. If there are more than two outgoing edges, the structure is required to be changed in order to have only two outgoing edges. If it is not possible, then this formula ($\Pi + 1$) is not applicable.

2. Cyclomatic complexity is equal to the number of regions of the flow graph.

These results have been used in an operational environment by advising developers to limit their software modules by cyclomatic complexity instead of physical size. The particular upper bound that has been used for cyclomatic complexity is 10, which seems like reasonable, but not magical. Upper limit when it exceeds 10, developers have to either recognise and modularise sub-functions or redo the software. The intentions are to keep size of modules manageable and allow for testing all independent paths. There are situations in which this limit seems unreasonable: *e.g.*, when a large number of independent cases follow a selection function like switch or case statement.

Example 8.15

Consider a flow graph given in the Fig. 8.23 and calculate the cyclomatic complexity by all three methods.

Solution

Cyclomatic complexity can be calculated by any of the three methods.

$$\begin{aligned}1. \quad V(G) &= e - n + 2P \\&= 13 - 10 + 2 = 5\end{aligned}$$

$$\begin{aligned}2. \quad V(G) &= \Pi + 1 \\&= 4 + 1 = 5\end{aligned}$$

$$\begin{aligned}3. \quad V(G) &= \text{number of regions} \\&= 5\end{aligned}$$

Therefore, complexity value of a flow graph in Fig. 8.23 is 5.

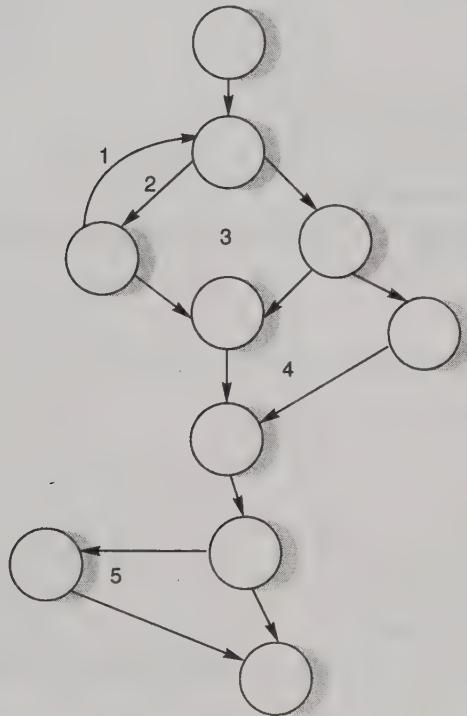


Fig. 8.23: [MCCA76]

Example 8.16

Consider the previous date program with DD path graph given in Fig. 8.17. Find cyclomatic complexity.

Solution

Number of edges (e) = 65

Number of nodes (n) = 49

$$(i) \quad V(G) = e - n + 2P = 65 - 49 + 2 = 18$$

$$(ii) \quad V(G) = \Pi + 1 = 17 + 1 = 18$$

$$(iii) \quad V(G) = \text{Number of regions} = 18.$$

Therefore cyclomatic complexity is 18. This value is quite high and indicates that there is a need to redesign the program. If we review the code, it is clear that we may have two separate modules, one for “checking the validity of date” and another for “previous date calculation”. Actually, these two functions are independent and should be placed in different modules. If we do so, cyclomatic complexity will be reduced to 10 and 8 respectively (split occurs at n_{25}). Hence, this method gives us some idea about the structure of the code and modularity of the program.

Example 8.17

Consider the quadratic equation problem given in Example 8.13 with its DD Path graph. Find the cyclomatic complexity:

Solution

Number of nodes = 19

Number of edges = 24

$$(i) V(G) = e - n + 2P = 24 - 19 + 2 = 7$$

$$(ii) V(G) = \Pi + 1 = 6 + 1 = 7$$

$$(iii) V(G) = \text{Number of regions} = 7$$

Hence cyclomatic complexity is 7 meaning thereby, seven independent paths in the DD Path graph.

Example 8.18

Consider the classification of triangle problem given in Example 8.14. Find the cyclomatic complexity.

Solution

Number of edges = 23

Number of nodes = 18

$$(i) V(G) = e - n + 2P = 23 - 18 + 2 = 7$$

$$(ii) V(G) = \Pi + 1 = 6 + 1 = 7$$

$$(iii) V(G) = \text{Number of regions} = 7$$

The cyclomatic complexity is 7. Hence, there are seven independent paths as given in Example 8.14.

8.4.3 Graph Matrices

Whenever graphs are used for testing, we are interested to find independent paths. The objective is to trace all links of the graph at least once. Path tracing is not an easy task and is subject to errors. If the size of graph increases, it becomes difficult to do path tracing manually. In practice, it is always advisable to go for testing tool. To develop such a tool, a data structure, called graph matrix can be quite helpful.

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (*i.e.*, the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices [BEIZ90] are shown in Fig. 8.24.

In the graph matrix, there is a place to put every possible direct connection between any node and any other node. A connection from node i to node j does not imply a connection from

node j to node i . In Fig. 8.24 (c) the $(5, 6)$ entry is m but the $(6, 5)$ entry is c . If there are several links between two nodes, then the entry is a sum ; the '+' sign denotes parallel links.

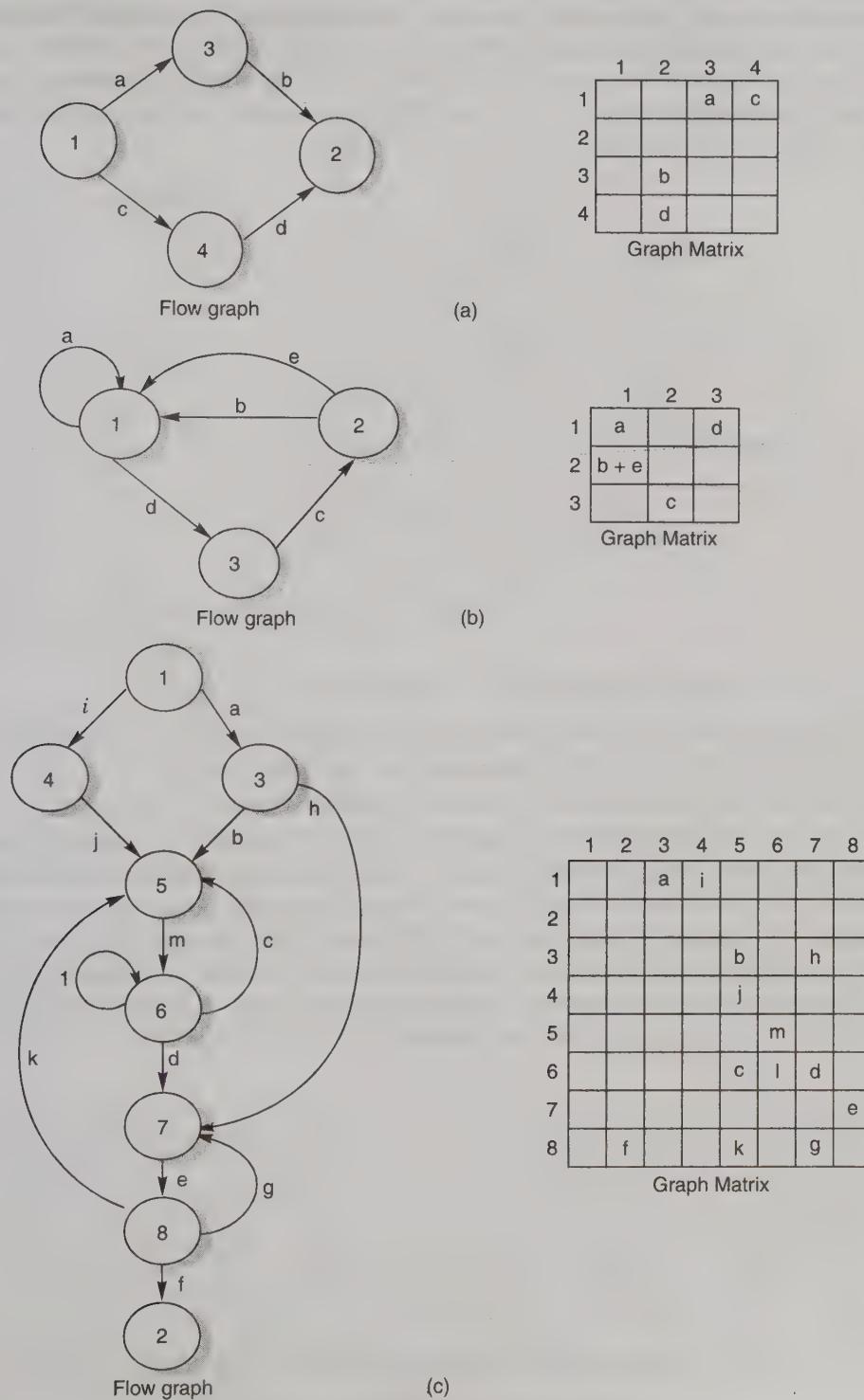


Fig. 8.24: Flow graphs and graphs matrices

Graph matrix is nothing but the tabular representation of a flow graph. It does not seem to be useful in the present form. If we assign weight to each entry, the graph matrix can be used for evaluating useful information required during testing. The simplest weight is 1, if there is a connection and 0 if there is no connection. A matrix with such weights is called a connection matrix. A connection matrix for Fig. 8.24 (c) is obtained by replacing each entry with 1, if there is a link and 0 if there is no link [BEIZ90]. As usual, to reduce clutter we do not write down 0 entries and this matrix is shown in Fig. 8.25.

		Connections							
		1	2	3	4	5	6	7	8
1			1	1					
2									
3					1		1		
4					1				
5						1			
6					1	1	1		
7								1	
8	1			1		1			

$2 - 1 = 1$
 $1 - 1 = 0$
 $1 - 1 = 0$
 $3 - 1 = 2$
 $1 - 1 = 0$
 $3 - 1 = 2$
 $6 + 1 = 7$

Fig. 8.25: Connection matrix of flow graph shown in Fig. 8.24 (c)

The connection matrix can also be used to find cyclomatic complexity as shown in Fig. 8.25. Each row having more than one entry represents the predicate node.

Each entry in the graph matrix expresses a relation between the pair. It is a direct relation, but we are usually interested in indirect relations that exist by virtue of intervening nodes between the two nodes of interest. Squaring a matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that node is transitive. A relation \mathbf{R} is transitive if $a \mathbf{R} b$ and $b \mathbf{R} c$ implies $a \mathbf{R} c$. Most relations used in testing are transitive. Therefore, the square matrix represents, all paths of two links long. The K^{th} power of matrix represents all paths of K links long. Consider the graph matrix shown in Fig. 8.24 (a) and apply this to that matrix.

		1	2	3	4
1			a	c	
2					
3	b				
4	d				

$[A]$

		1	2	3	4
1		ab + cd			
2					
3					
4					

$[A]^2$

The square matrix represent that there are two path ab and cd from node 1 to node 2.

Example 8.19

Consider the flow graph shown in the Fig. 8.26 and draw the graph & connection matrices. Find out cyclomatic complexity and two/three link paths from a node to any other node.

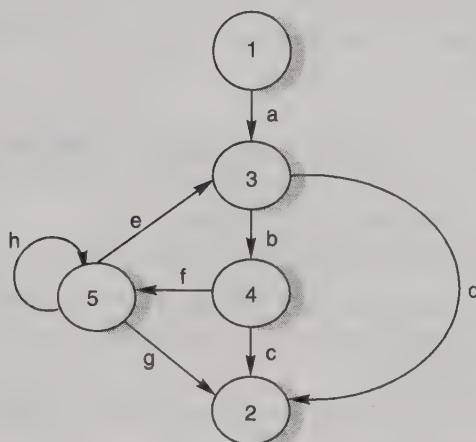


Fig. 8.26: Flow graph [BEI90].

Solution

The graph & connection matrices are given below:

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

Graph Matrix (A)

	1	2	3	4	5	Connections
1			1			1 - 1 = 0
2						2 - 1 = 1
3		1		1		2 - 1 = 1
4		1			1	3 - 1 = 2
5	1	1	1		1	4 + 1 = 5

Connection Matrix

	1	2	3	4	5
1		ad		ab	
2					
3		bc			bf
4		fg	fe		fh
5		ed + hg	he	eb	h^2

$[A^2]$

	1	2	3	4	5
1		abc			afb
2					
3		bfg		bfe	
4		fed + fhg		fhe	feb
5		ebc + hed + h^2g	h^2e	heb	$ebf + h^3$

$[A^3]$

This indicates that there is a two links path “ad” and three-link path “abc” available from node 1 to node 2.

Our main objective is to use matrix operations to obtain the set of all paths between all nodes. This can be obtained by summing A , A^2 , A^3 , ... A^{n-1} .

These operations are easy to programme and can be used for designing testing tools.

8.4.4 Data Flow Testing

Data flow testing is another form of structural testing. It has nothing to do with data flow diagrams. Here, we concentrate on the usage of variables and the focus points are:

- (i) Statements where variables receive values.
- (ii) Statement where these values are used or referenced.

Flow graphs are also used as a basis for the data flow testing as in the case of path testing. Some times, we feel that it may serve as a check on path testing and is treated as another form of path testing [JORG95].

As we know, variables are defined and referenced throughout the program. We may have few define/reference anomalies:

- (i) A variable is defined but not used/referenced.
- (ii) A variable is used but never defined.
- (iii) A variable is defined twice before it is used.

These anomalies can be identified by static analysis of code *i.e.*, analysing code without executing it. In order to formalise the approach of data flow testing, some definitions are required, which are discussed below [JORG95].

Definitions

The definitions refer to a program P that has a program graph $G(P)$ and a set of program variables V . The $G(P)$ has a single entry node and a single exit node. The set of all paths in P is $\text{PATHS}(P)$.

(i) **Defining node:** Node $n \in G(P)$ is a defining node of the variable $v \in V$ written as $\text{DEF}(v, n)$, iff the value of the variable v is defined at the statement fragment corresponding to node n .

(ii) **Usage Node:** Node $n \in G(P)$ is a usage node of the variable $v \in V$, written as $\text{USE}(v, n)$, iff the value of the variable v is used at the statement fragment corresponding to node n . A usage node $\text{USE}(v, n)$ is a predicate use (denoted as p) iff statement n is a predicate statement otherwise $\text{USE}(v, n)$ is a computation use (denoted as c).

(iii) **Definition use:** A definition use path with respect to a variable v (denoted du-path) is a path in $\text{PATHS}(P)$ such that, for some $v \in V$, there are define and usage nodes $\text{DEF}(v, m)$ and $\text{USE}(v, n)$ such that m and n are the initial and final nodes of the path.

(iv) **Definition clear:** A definition clear path with respect to a variable v (denoted dc-path) is a definition use path in $\text{PATHS}(P)$ with initial and final nodes $\text{DEF}(v, m)$ and $\text{USE}(v, n)$, such that no other node in the path is a defining node of v .

The du-paths and dc paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. The du-paths that are not definition clear are potential trouble spots.

Hence, our objective is to find all du-paths and then identify those du-paths which are not dc-paths. The steps are given in Fig. 8.27. We may like to generate specific test cases for du-paths that are not dc-paths.

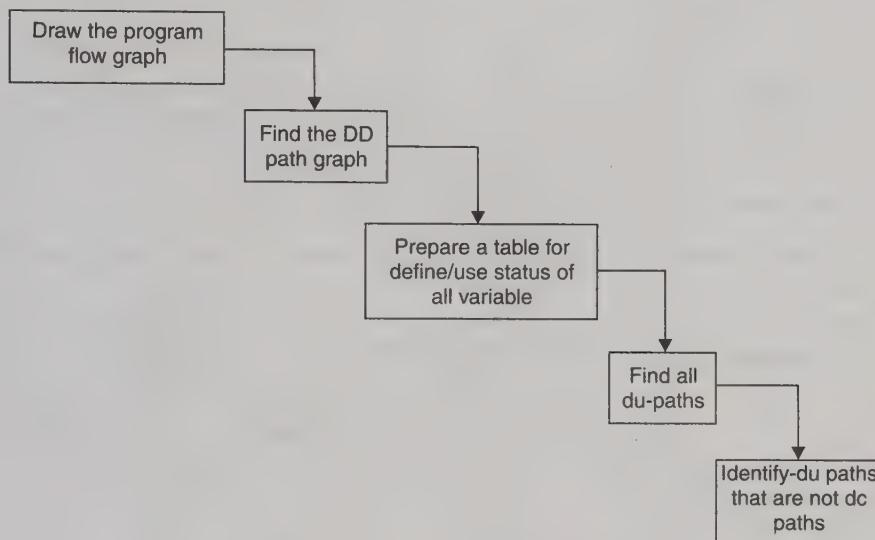


Fig. 8.27: Steps for data flow testing.

One simple approach is to test every du-path at least once. This is known as du testing strategy. In this strategy, chance to cover all edges of the flow graph is very high; although it does not guarantee 100% coverage. This is also effective for error detection specifically for du-paths that are not definition clear paths.

Example 8.20

Consider the program of the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a, b, c) and values for each of these may be from interval $[0, 100]$. The program is given in Fig. 8.19. The output may have one of the options given below:

- (i) Not a quadratic equation
- (ii) real roots
- (iii) imaginary roots
- (iv) equal roots
- (v) invalid inputs.

Find all du-paths and identify those du-paths that are definition clear.

Solution

Step I: The program flow graph is given in Fig. 8.19 (a). The variables used in the program are a, b, c, d , validinput, D.

Step II: DD Path graph is given in Fig. 8.19 (b). The cyclomatic complexity of this graph is 7 indicating there are seven independent paths.

Step III: Define/Use nodes for all variables are given below:

Variable	Defined at node	Used at node
a	6	11, 13, 18, 20, 24, 27, 28
b	8	11, 18, 20, 24, 28
c	10	11, 18
d	18	19, 22, 23, 27
D	23, 27	24, 28
Valid input	3, 12, 14	17, 31

Step IV: The du-paths are identified and are named by their beginning and ending nodes using Fig. 8.19 (a).

Variable	Path (beginning, end) nodes	Definition clear ?
a	6, 11 6, 13 6, 18 6, 20 6, 24 6, 27 6, 28	Yes Yes Yes Yes Yes Yes Yes
b	8, 11 8, 18 8, 20 8, 24 8, 28	Yes Yes Yes Yes Yes
c	10, 11 10, 18	Yes Yes
d	18, 19 18, 22 18, 23 18, 27	Yes Yes Yes Yes
D	23, 24 23, 28 27, 24 27, 28	Yes Path not possible Path not possible Yes
Valid input	3, 17 3, 31 12, 17 12, 31 14, 17 14, 31	no no no no Yes Yes

Total du-paths are 26 out of which 4 paths are not definition clear paths. Two path $< 27, 24 >$, $< 23, 28 >$ are impossible paths. Our emphasis should be to generate test cases for all 26 du-paths, and if not possible, then, at least for 4 du-paths that are not definition clear paths.

Example 8.21

Consider the program given in Fig. 8.20 for the classification of a triangle. Its input is a triple of positive integers (say a, b, c) from the interval [1, 100]. The output may be:

[Scalene, Isosceles, Equilateral, Not a triangle, Invalid inputs].

Find all du-paths & identify those du-paths that are definition clear.

Solution

Step I: The program flow graph is given in Fig. 8.20 (a). The variables used in the program are a, b, c , validinput.

Step II: DD path graph is given in Fig. 8.20 (b). The cyclomatic complexity of the graph is 7 and, thus, there are 7 independent paths.

Step III: Define/use nodes for all variables are given below:

Variable	Defined at node	Used at node
a	6	10, 11, 19, 22
b	7	10, 11, 19, 22
c	9	10, 11, 19, 22
Validinput	3, 12, 16	18, 29

Step IV: The du-paths are identified and are named by their beginning and ending nodes using Fig. 8.20 (a).

Variable	Path (beginning, end) nodes	Definition clear ?
a	5, 10	Yes
	5, 11	Yes
	5, 19	Yes
	5, 22	Yes
b	7, 10	Yes
	7, 11	Yes
	7, 19	Yes
	7, 22	Yes
c	9, 10	Yes
	9, 11	Yes
	9, 19	Yes
	9, 22	Yes

Variable	Path (beginning, end) nodes	Definition clear ?
Validinput	3, 18	no
	3, 29	no
	12, 18	no
	12, 29	no
	16, 18	Yes
	16, 29	Yes

Hence total du-paths are 18 out of which four paths are not definition clear.

8.4.5 Mutation Testing

Mutation testing is a fault based technique that is similar to fault seeding, except that mutations to program statements are made in order to determine properties about test cases. It is basically a fault simulation technique. In this technique, multiple copies of a program are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program. A mutant that is detected by a test case is termed "killed" and the goal of mutation procedure is to find a set of test cases that are able to kill groups of mutant programs [FRIE95].

Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. The new expression should be syntactically legal according to the language. If one or more mutant operators are applied to all expressions in a program, the result is a large set of mutants, all of which must be killed by the test cases or shown to be equivalent to the original expression.

When we mutate code there needs to be a way of measuring the degree to which the code has been modified. For example, if the original expression is $x + 1$ and the mutant for that expression is $x + 2$, that is a lesser change to the original code than a mutant such as $(c*22)$, where both the operand and the operator are changed. We may have a ranking scheme, where a first order mutant is a single change to an expression, a second order mutant is a mutation to a first order mutant, and so on. High order mutants becomes intractable and thus in practice only low order mutants are used.

One difficulty associated with whether mutants will be killed is the problem of reaching the location; if a mutant is not executed, it cannot be killed. Special test cases are to be designed to reach a mutant. For example, suppose, we have the code

```

Read (a, b, c);
If (a > b) and (b = c) then
  x: = a*b*c; {make mutants; m1, m2, m3.....}

```

To execute this, input domain must contain a value such that a is greater than b and b equals c . If input domain does not contain such a value, then all mutants made at this location should be considered equivalent to the original program, because the statement $x: = a*b*c$ is dead code (code that cannot be reached during execution). If we make the mutant $x + y$ for

$x + 1$, then we should take care about the value of y which should not be equal to 1 for designing a test case.

The manner by which a test suite is evaluated (scored) via mutation testing is as follows: for a specific test suite and a specific set of mutants, there will be three types of mutants in the code *i.e.*, killed or dead, live, equivalent. The sum of the number of live, killed, and equivalent mutants will be the total number of mutants created. The score associated with a test suite T and mutants M is simply

$$\frac{\# \text{ killed}}{\# \text{ total} - \# \text{ equivalent}} \times 100\%$$

This equation allows us to determine the likelihood that for a particular mutation adequate test suite will catch real faults based on how well the suite-killed mutants. Note that this scheme does not penalize a test suite if it is of a large size than a different test suite. For instance, suppose that test suite A with 100 test cases had a score of 50%, and test suite B with 25 test cases had a score of 49%. Although A has a better mutation score, it is also four times as large as B, and for the small increase in mutation score there is a large increase in testing costs.

8.5 LEVELS OF TESTING

Our emphasis during testing is to examine and modify the source code. There are three levels of testing *i.e.*, individual module to the entire software system. At one end, we attempt to test modules in all possible ways so as to detect any errors. From there, we combine to form aggregates of modules and test their detailed structure and functions. At the end, we may ignore the internal structure of the software and concentrate on how it responds to the typical kind of operations that will be requested by the user. These three levels of testing are usually referred to as unit testing, integration testing, and system testing [JONE90] as shown in Fig. 8.28.

Out of the three traditional levels of testing, unit testing is best understood. The testing methods discussed so far in this chapter are directly applicable to unit testing. System testing is understood better than integration testing, but both need clarification. The bottom up approach sheds some insight: test the individual units, and then integrate these into subsystems until the entire system is tested. System testing is something that the customer understands, and it often borders on customer acceptance testing. Generally, system testing is functional rather than structural. This is mostly due to the absence of a structural basis for system test cases. In the traditional view, integration testing is what's leftover; it is not unit testing, and it is not system testing. Most of the usual discussions on integration testing concentrate on order in which units are integrated: top-down, bottom-up, or the big bang (everything at once) of the three phases, integration is the least well understood [JORG95].

Errors located during testing may fall into several categories [DEUT82]. Those that require immediate attention are usually of a nature that they crash the software under consideration, and testing cannot continue until they are removed. Some errors need to be corrected before testing is complete but can be ignored during the immediate testing schedule. In general, the importance of removing an error is proportional to its severity, the frequency with which it

occurs, and the degree to which the customer is aware of it. There are errors that may be acceptable to the user when compared to the cost of correcting them at the moment; these will be held until some future release date or until the customer becomes concerned enough to request a change.

Finally, there are errors, usually non reproducible, for which insufficient evidence exists to evaluate them. They are flagged for ongoing consideration until they become more tractable. Non-reproducible errors are a frequent problem with concurrent software.

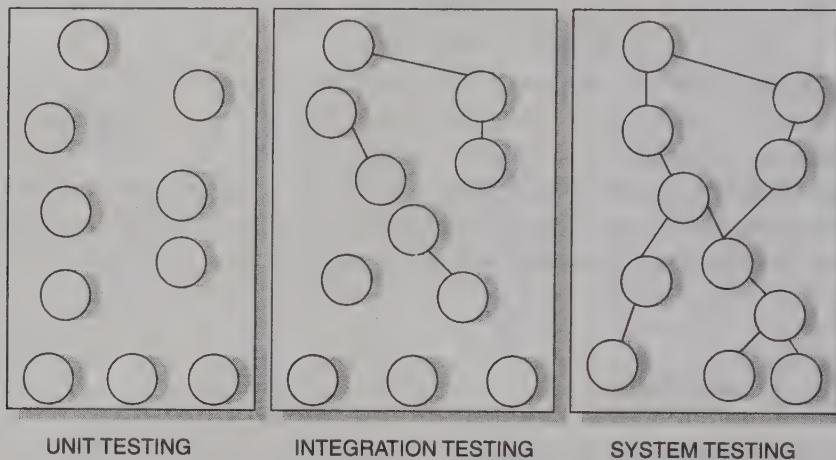


Fig. 8.28: Levels of testing.

Number of other activities are also carried out along with testing. These include test documentation, product modification, and discussions with the customer to schedule installation, preparation of installation & training material etc. When the product has finally been released by the quality assurance group at the end of testing, we want to move immediately into the installation portion of product delivery.

8.5.1 Unit Testing

Unit testing is the process of taking a module and running it in isolation from the rest of the software product by using prepared test cases and comparing the actual results with the results predicted by the specifications and design of the module. One purpose of testing is to find (and remove) as many errors in the software as practical. There are number of reasons in support of unit testing than testing the entire product [JONE90].

1. The size of a single module is small enough that we can locate an error fairly easily.
2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
3. Confusing interactions of multiple errors in widely different parts of the software are eliminated.

There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? One approach is to construct an appropriate driver routine to call it and, simple stubs to be called by it, and to insert output statements in it.

Stubs serve to replace modules that are subordinate to (called by) the module to be tested. A stub or dummy subprogram uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns [PRES97].

This overhead code, called scaffolding represents effort that is important to testing, but does not appear in the delivered product as shown in Fig. 8.29 [JONE90]. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many modules cannot be adequately unit tested with simple overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers and stubs are also used). A second approach is to generate the scaffolding automatically by means of a test harness. A test harness, among other things, allows us to own a single unit in isolation, while simulating the rest of the software system environment by providing appropriate input, output, parameters, and interaction for the unit. A third and rather ineffective technique is to omit unit testing and simply to allow incremental addition of modules to a partially integrated product, hoping that the integration testing will also provide sufficient coverage of the module's structure. This technique is usually inadequate but nevertheless it is often recommended in the literature. The white box testing approaches are normally used for unit testing and the steps can be conducted in parallel for multiple modules.

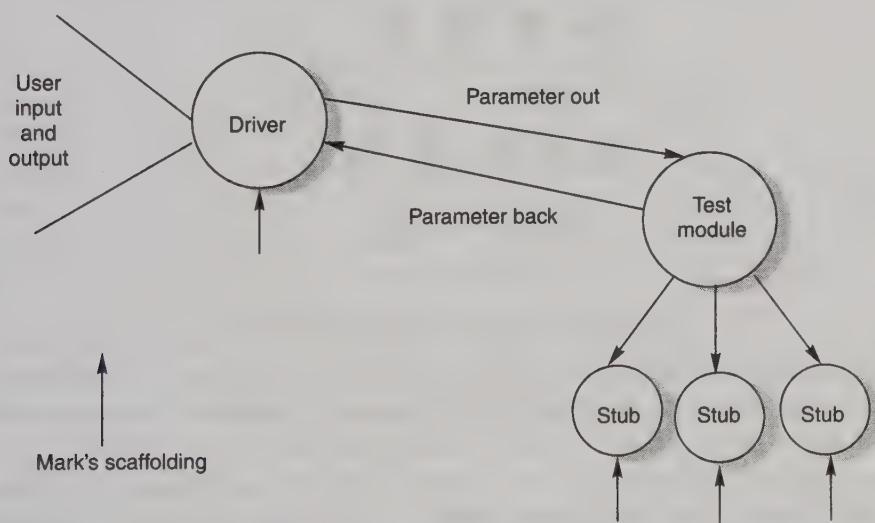


Fig. 8.29: Scaffolding required testing a program unit (module).

8.5.2 Integration Testing

The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed. One specific target of integration testing is the interface: whether parameters match on both sides as to type, permissible ranges, meaning and utilization [COLL88].

There are several classical integration strategies that really have little basis in a rational methodology. Top down-integration proceeds down the invocation hierarchy, adding one module

at a time until an entire tree level is integrated; and thus it eliminates the need for drivers. The bottom-up strategy works similarly from the bottom and has no need of stubs. A sandwich strategy runs from top and bottom concurrently, meeting somewhere in the middle. All the three approaches are shown in Fig. 8.30.

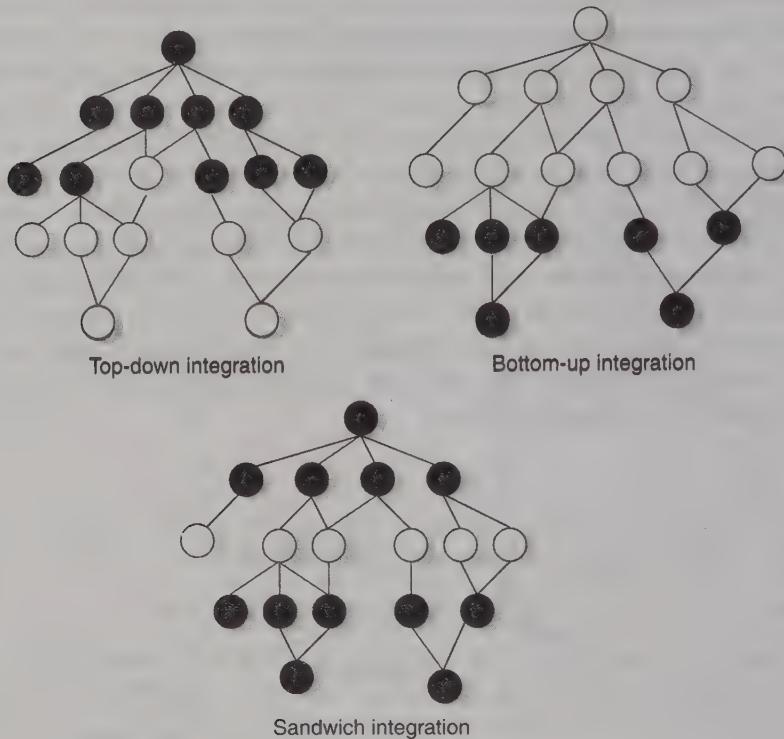


Fig. 8.30: Three different integration approaches.

There has been a lot of discussion about which one of the three; viz top-down, bottom up or sandwich integration is best. Most of this concern is driven by the need to mix module testing into integration, because of a lack of test harnesses.

When the lack disappears, the concern is no longer significant. But there are a few principles that should guide integration. First, if we are going to overlap module testing and integration testing, then we will find that not all modules will be ready for integration at the same time.

Integration should follow the lines of first putting together those subsystems that are of great concern. This prioritisation might dictate top-down integration if control and the user interface were the most worrisome or complex part of software. This can occur, for instance, if the customer is anxious to see a running program early in the testing phase. In another situation, the machine interface and performance might be of special interest, and then bottom-up integration would be dictated. With bottom-up integration we stand better chance of experiencing a high degree of concurrency during our integration. It has also been said that top-down integration is an exercise in faith that everything will work out well, whereas bottom-up integration shows that things really do work.

With integration testing, we move slowly away from structural testing and toward functional testing, which treats a module as an impenetrable mechanism for performing a function. As the aggregated modules become larger and larger, we lose our ability to think about path coverage and domains, and must be satisfied with simply determining that the product seems to do what we intended. That is, we treat the product as a black box and verify that specified inputs produce appropriate outputs, without concern for the internal structure of the software. However, even in functional tests we can and should continue to choose test data that represent important or unusual conditions [JONE90]. In attempting to bridge the gap from small units to a large software system, we may not be able to jump from structural tests to functional tests immediately, while maintaining the principle of adding one unit at a time. One way of testing intermediate levels of the product is to isolate utility, or internal functions defined during the architectural design stage and to use them as the basis for functional testing.

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly.

8.5.3 System Testing

Of the three levels of testing, the system level is closest to everyday experience. We test many things; a used car before we buy it, an on-line cable network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations; not with respect to a specification or a standard. Consequently, goal is not to find faults, but to demonstrate performance. Because of this we tend to approach system testing from a functional standpoint rather than from a structural one. Since it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and is compounded by the reduced testing interval that usually remains before a delivery deadline [JORG95].

As we know, software is one component of a large computer based system. Ultimately, software is incorporated with other system components (e.g., new hardware, information), and thus, a series of special tests are to be conducted. Many times, software products are designed to run on a variety of hardware configurations. The software should actually be tested on many different hardware set-ups, although the full range of memory, processor, operating system, and peripheral possibilities may be too large for complete testing. There are many types of specifications, and we should be aware of those as we perform system testing. For instance, there may be a specified level of performance required of the software. This may involve measurement of response time under various loads and operating conditions. It may also require measurement of main and disk memory usage. Software reliability should also be measured during all other tests of the integrated product. If minimum and average up-time behaviour of the product were specified, then these should be met. The time and effort needed to recover from failures should also be recorded & compared with specifications. These specifications should represent customer's wants & needs, and during system testing, we can try to see if the requirements & specifications really do coincide.

Petschenik gives some guidelines for choosing test cases during system testing [PETS85]. The first is that testing the system's capabilities is more important than testing its components. This implies that failures that are catastrophic should be looked for whereas failures that are merely annoying need not worry us. The idea is that a user can deal with a badly formatted report, but probably cannot deal with unavailability of the report.

Petschenik's second rule is that testing the usual is more important than testing the exotic. This can be accomplished by subjecting the software to the kind of use that is representative of actual use, as described in the operational profile. The user may be able to help with this kind of testing. In fact, software engineers may exhibit blind spots that cause them to notice exotic problems; while they ignore problems that user would spot immediately.

Third, if we are testing after modification of an existing product, we should test old capabilities rather than new ones. The rationale here is that the user is not depending on the new functions of the software, and would not be paralysed if these were not right. But a failure in the old functionality could do just that-paralyse the user's entire operation.

The rationale for Petschenik's testing priorities is that exhaustive testing of functional capabilities is incomplete with the kind of short update cycle imposed on many software products. Just as dependability is more important to the user than the total correctness, the basic and existing functionality also weigh much more heavily than total functionality. Foremost we should avoid disrupting current usage by introducing a new release of the product that would not support current functions.

During system testing, we should evaluate a number of attributes of the software that are vital to the user and are listed in Fig. 8.31. These represent the operational correctness of the product and may be part of the software specifications [JONE90].

Usable	Is the product convenient, clear, and predictable?
Secure	Is access to sensitive data restricted to those with authorization?
Compatible	Will the product work correctly in conjunction with existing data, software, and procedures?
Dependable	Do adequate safeguards against failure and methods for recovery exist in the product?
Documented	Are manuals complete, correct, and understandable?

Fig. 8.31: Attributes of software to be tested during system testing.

8.6 DEBUGGING

As discussed earlier, the goal of testing is to identify errors (bugs) in the program. The process of testing generates symptoms, and a program's failure is a clear symptom of the presence of an error. After getting a symptom, we begin to investigate the cause and place of that error. After identification of place, we examine that portion to identify the cause of the problem. This process is called debugging.

Hence, debugging is the activity of locating and correcting errors. It can start once a failure has been detected. Unfortunately, going from the detection of a failure to correcting the error that is responsible, is far from trivial. It is one of the least understood activities in software development and is practiced with the least amount of discipline. It is often approached with much hope and little planning [GHEZ94].

8.6.1 Debugging Techniques

Most developers have learned through experience several techniques for debugging. Generally these are applied in a trial and error manner. Debugging is not an easy process. This is probably due to human psychology rather than software technology. Error removal requires humility to even admit the possibility of errors in the code we have created and requires an open mind that is willing to see what the software actually does rather than it should do. Commenting on human aspect of debugging, Shneiderman [SHNE80], states:

"It is one of the most frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that we have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors, increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately corrected".

However, Pressman [PRES97] explained few characteristics of bugs that provide some clues.

1. "The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located in other part. Highly coupled program structures may complicate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g. round-off inaccuracies).
4. The symptom may be caused by a human error that is not easily traced.
5. The symptom may be a result of timing problems rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g. a real time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware with software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors".

A number of popular techniques are given in table 8.8.

In general, none of these techniques should be used without a thorough prior analysis of symptoms the error resulting in a hypothesis concerning the cause of the errors. For instance, if two modules behave properly when operated separately, and a failure occurs when they are integrated, their interface should be checked for consistency.

8.6.2 Debugging Approaches

In heart of debugging process is not the debugging tools, but the underlying approaches used to deduce the cause of the error.

Table 8.8: A comparison of various debugging techniques

Techniques	Features	Advantages	Disadvantages
1. Core dumps	A printout of all registers and relevant memory locations is obtained and studied. All dumps should be well documented and retained for possible use on subsequent problems.	<ol style="list-style-type: none"> 1. The complete contents of memory at a crucial instant of time are obtained for study. 2. Can be cost-effective if used to explore validity of a well-formulated error hypothesis. 	<ol style="list-style-type: none"> 1. Require some CPU time, significant I/O time, and much analysis time. 2. Wasteful if used indiscriminately (<i>i.e.</i>, at noncrucial instant or without an error theory or debugging plan.) 3. Hexadecimal numbers are cumbersome to interpret and it is difficult to determine the address of source-language variables.
2. Traces	Essentially similar to core dumps, except the printout contains only certain memory and register contents and printing is conditional on some event occurring. Typical conditioning events are entry, exit, or use of (1) a particular subroutine, statement, macro, or database; (2) communication with a terminal, printer, disk, or other peripheral; (3) the value of a variable or expression; and (4) timed actuations (periodic or random) in certain real-time systems. A special problem with trace programs is that the conditions are entered in the source language and any changes require a recompilation.		
3. Print statements	The standard print statement in the language being used is sprinkled throughout the program to output values of key variables.	<ol style="list-style-type: none"> 1. This is a simple way to test whether a particular variable changes, as it should after a particular event. 2. A sequence of print statements portrays the dynamics of variable changes. 	<ol style="list-style-type: none"> 1. They are cumbersome to use on large programs. 2. If used indiscriminately, they can produce copious data to be analysed, much of which are superfluous.
4. Debugging Programs	A program which runs concurrently with the program under test and provides commands to (1) examine memory and registers; (2) stop execution of the program at a particular point; (3) search for references to particular constants, variables, registers.	<ol style="list-style-type: none"> 1. Terminal-oriented real-time program. 2. Considerable flexibility to examine dynamics of operation. 	<ol style="list-style-type: none"> 1. Generally works on a machine language program. 2. Higher-level-language versions must work with interpreters. 3. More commonly used on microcomputers than large computers.

One obvious techniques is that of trial & error. The debugger looks at the error symptoms, reaches a snap judgement as to where in the code the underlying error might be, and jumps into and roam around in the program with one or more debugging techniques from the standard pit of tools. Obviously, this is slow and wasteful approach.

The second approach, called backtracking, is to examine the error symptoms to see where they are first noticed. One then backtracks in the program flow of control to a point where the symptoms have disappeared. Generally, this process brackets the location of the error in the program. Subsequent careful study of the bounded segment of the code generally reveals the cause. Another obvious variation of backtracking is forward tracking, where we use, print statements or other means to examine a succession of intermediate results to determine at what point the result first become wrong. If we assume that we know the correct values of the variables at several key points within a program, we can adopt a binary search type of strategy. A set of inputs are injected near the middle of the program and the output is examined. If the output is correct the error is in the first half of the program; and if output is wrong, the error is in the second half of the program. This process is repeated as many times as it is feasible to bracket the erroneous portion of the code for final analysis [SHOO87].

The third approach could be to insert watch points (output statements) at the appropriate place in the program. We can use a software to insert watch points in a program without modifying the program manually. This eliminates many practical problems involved with adding adhoc-debugging statements to the program manually. For example, in the manual modes, we have to be sure that after finding & fixing the error, we remove any debugging statements we have inserted.

The fourth approach is more general and called induction & deduction [MYER79]. The inductive approach comes from the formulation of a single working hypothesis based on the data, on the analysis of existing data, and on especially collected data to prove or disprove the working hypothesis. A description of the steps follows; a flowchart of their application sequence is shown in Fig. 8.32.

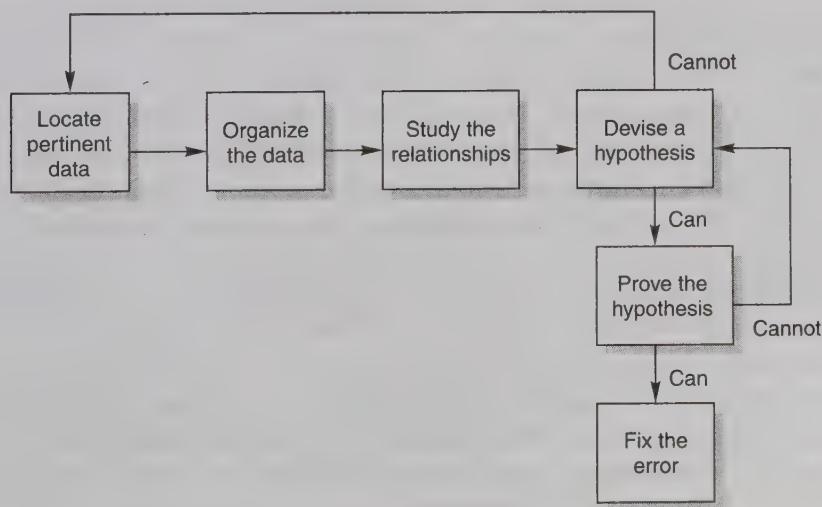


Fig. 8.32: The inductive debugging process [MYER79].

Induction approach

1. **Locate the pertinent data:** A major mistake made when debugging a program is failing to take account of all available data or symptoms about the problem. The first step is the enumeration of all that is known about what the program did correctly, and what it did incorrectly (i.e., the symptoms that led one to believe that an error exists). Additional valuable clues are provided by similar, but different, test cases that do not cause the symptoms to appear.
2. **Organize the data:** Remembering that induction implies that one is progressing from the specific to the general, the second step is the structuring of the pertinent data to allow one to observe patterns. Of particular importance is the search for contradictions (i.e., "the error occurs only when the customer has no outstanding balance in his margin account").
3. **Devise a hypothesis:** The next steps are to study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. If one cannot devise a theory more data is necessary, possibly obtained by devising and executing additional test cases. If multiple theories seem possible the most probable one is selected first.
4. **Prove the hypothesis:** A major mistake at this point, given the pressures under which debugging is usually performed, is skipping this step by jumping to conclusions and attempting to fix the problem. However, it is vital to prove the reasonableness of the hypothesis before proceeding. A failure to do this often results in the fixing of only a symptom of the problem, or only a portion of the problem. The hypothesis is proved by comparing it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues. If it does not either the hypothesis is invalid, or the hypothesis is incomplete, or multiple errors are present.

Deduction approach

The process of deduction begins by enumerating all causes or hypotheses, which seem possible. Then, one by one, particular causes are ruled out until a single one remains for validation. A description of the steps follows; a flowchart of their sequence appears in Fig. 8.33 [MYER79].

1. **Enumerate the possible causes or hypotheses:** The first step is to develop a list of all conceivable causes of the error. They need not be complete explanations; they are merely theories through which one can structure and analyse the available data.
2. **Use the data to eliminate possible causes:** By a careful analysis of the data, particularly by looking for contradictions, one attempts to eliminate all but one of the possible causes. If all are eliminated, additional data are needed (e.g. by devising additional test cases) to devise new theories. If more than one possible cause remains, the most probable cause, the prime hypothesis, is selected first.
3. **Refine the remaining hypothesis:** The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory (e.g. "error in handling the last transaction in the file") to something more specific (e.g., "the last transaction in the buffer is overlaid with the end-of-file indicator").

4. Prove the remaining hypothesis: This vital step is identical to step 4 in the induction method.

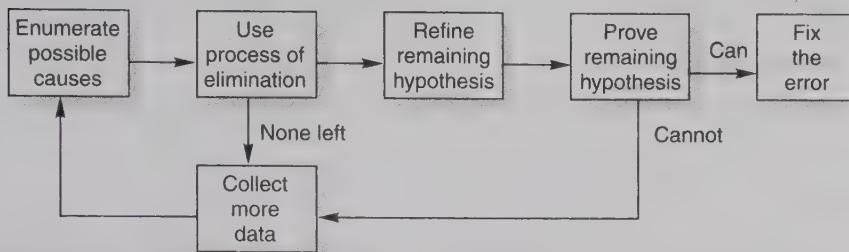


Fig. 8.33: The deductive debugging process [MYER79].

8.6.3 Debugging Tools

Each of the debugging approaches can be supplemented with debugging tools. We can apply wide variety of debugging compilers, dynamic debugging aids, automatic test case generators, memory dumps and cross reference maps. However, tools are not substitute for careful evaluation based on a complete software design document and clear source code.

Compiler is an effective tool for checking & diagnostics. Of course, it checks only syntax errors and particular kind of runtime errors. Compiler should give proper & detailed messages of errors that will be of great help to the debugging process. Compiler can give all such informations in the attribute table, which is printed along with the listing. The attributes table contains various level of warnings which have been picked up by the compiler scan and which are noted. Therefore, compilers are coming with error-detection features and there is no excuse for compilers without meaningful error messages.

8.7 TESTING TOOLS

One way to improve the quality & quantity of testing is to make the process as pleasant as possible for the tester. This means that tools should be as concise, powerful & natural as possible.

The two broad categories of software testing tools are: static and dynamic. Most tool functions fall cleanly into one category or the other, but there are some exceptions like symbolic evaluation systems and mutation analysis systems (which actually run interpretively). There are different types of tools available and some are listed below [VICK84].

- (i) Static analysers, which examine programs systematically and automatically.
- (ii) Code inspectors, who inspect programs automatically to make sure they adhere to minimum quality standards.
- (iii) Standards enforcers, which impose simple rules on the developer.
- (iv) Coverage analysers, which measure the extent of coverage.
- (v) Output comparators, used to determine whether the output in a program is appropriate or not.

- (vi) Test file/data generators, used to set up test inputs.
- (vii) Test harnesses, used to simplify test operations.
- (viii) Test archiving systems, used to provide documentation about programs.

8.7.1 Static Testing Tools

Static testing tools are those that perform analysis of the programs without executing them at all.

Static analysers

A static analyser operates from a precomputed database of descriptive information derived from the source text of the program. The idea of a static analyser is to prove allegations, which are claims about the analysed programs that can be demonstrated by systematic examination of all the cases.

There is a close relation to code inspectors, but static analysers are stronger. Typical cases include FACES, DAVE, RXVP, PL/I, checkout compiler, LINT on PWB/Unix and so on [MILL79]. All of these systems are language dependent in the sense that they apply to a particular language, and also often to a particular system. Many applications using static analysers find 0.1 to 0.2% NCSS (Non-Comment source statements) deficiency reports. Some of these are real, and others are spurious in the sense that these are false warnings that are later ignored after interpretation. These are language dependent and require high initial tool investment costs.

Code inspectors

A code inspector does a simple job of enforcing standards in a uniform way for many programs. These can be single statement or multiple statement rules. It is also possible to build code inspector assistance programs that force the inspector to do a good job by linking him to the process through an interactive environment. The AUDIT system is available which imposes some minimum conditions on the program. Code inspection activity is found in some COBOL tools (like AORIS librarian system) and in some parts of tools like RXVP.

Standard enforcers

This tool is like a code inspector, except that the rules are generally simpler. The main distinction is that a full-blown static analyser looks at whole programs, whereas a standard enforcer looks at only single statements.

Since only single statements are treated, the standards enforced tend to be cosmetic ones; even so, they are valuable because they enhance the readability of the programs. It seems well established that the readability of a program is an indirect indicator of its quality.

Other tools

Related tools are used to catch bugs indirectly through listings of the program that highlight the mistakes.

One example is a program generator that is used (mostly in COBOL environments, but possibly in others as well) to produce the proforma parts of each source module. Use of such a

method ensures that all programs look alike, which in itself enhances the readability of the programs.

Another example is using structured programming preprocessors that produce attractive print output. Such augmented program listings typically have automatic indentation, indexing features, and in some cases much more.

8.7.2 Dynamic Testing Tools

Dynamic testing tools seek to support the dynamic testing process. Besides individual tools that accomplish these functions, a few integrated systems group the functions under a single implementation.

A test consists of a single invocation of the test object and all of the execution that ensues until the test object returns control to the point where the invocation was made. Subsidiary modules called by the test object can be real or they can be simulated by testing stubs.

A series of tests is normally required to test one module or to test a set of modules. Test support tools must perform these functions:

- (i) Input setting: selecting of the test data that the test object reads when called
- (ii) Stub processing: handling outputs and selecting inputs when a stub is called
- (iii) Results display: providing the tester with the values that the test object produces so that they can be validated
- (iv) Test coverage measurement: determining the test effectiveness in terms of the structure of the program
- (v) Test planning: helping the tester to plan tests so they are both efficient and also effective at forcing discovery of defects

Coverage analyzers (execution verifiers)

A coverage analyser or execution verifier (or automated testing analyser, or automated verification system, etc.) is the most common and important tool for testing. It is often relatively simple. One of the common strategies for testing involves declaring a minimum level of coverage, ordinarily expressed as a percentage of the elemental segments that are exercised in aggregate during the testing process. This is called CI coverage, where CI denotes the minimum level of testing coverage so that every logically separate part of the program has been exercised at least once during the set of tests. Unit testing usually requires at least 85–90 of CI coverage.

Most often, CI is measured by planting subroutine calls-called software probes-along each segment of the program. The test object is then executed and some kind of run-time system is used to collect the data, which are then reported to the user in fixed-format reports. Use of coverage analysis can be incorporated into most quality assurance situations, although it is more difficult when there is too little space or when non real-time operation is inequivalent to real-time operation (an artifact of the instrumentation process).

Output comparators

Output comparators are used in dynamic testing-both single-module and multiple-module (system level) varieties to check that predicted and actual outputs are equivalent. This is also done during regression testing. The typical output comparator system objective is to identify differ-

ences between two files; the old and the new output from a program. Typical operating systems for the better minicomputers often have an output comparator, sometimes called a file comparator, built-in.

Test file generators

A test file generator creates a file of information that is used as the program and does so based on commands given by the user and/or from data descriptions (in a COBOL) program's data definition section, for example). Mostly, this is a COBOL-oriented idea in which the file of test data is intended to simulate transaction inputs in a data base management situation. This idea can be adapted to other environments.

Test data generators

The test data generation problem is a difficult one, and at least for the present is one for which no general solution exists. On the other hand, there is a practical need for methods to generate test data that meet a particular objective, normally to execute a previously unexercised segment in the program.

One of the practical difficulties with test data generation is that it requires generation of sets of inequalities that represent the conditions along a chosen path, and the reality is that:

- (i) Paths are too long and produce very complex formulas.
- (ii) Formula sets are non-linear.
- (iii) Many paths are illegal (not logically possible).

Practical approaches to automatic test data generation run into very difficult technical limits.

In practice, the techniques of variational test data generation are often quite effective. The test data are derived (rather than created) from an existing path that comes near the intended segment for which test data are to be found. This is often very easy to do, apparently because programs' structures tend to assist in the process. Automatically generating test data is effectively impossible, but good R & D work is now being done on the problem.

Test harness systems

A test harness system is one that is bound (*i.e.*, link-edited and relocated) around the test object and that (1) permits easy modification and control of test inputs and outputs and (2) provides for online measurement of CI coverage values. Some test harnesses are batch oriented, but the high degree of interaction available in a full-interactive system makes it seem very attractive in practical use. Modern thinking favours interactive test harness systems, which tend to be the focal point for installing many other kinds of analysis support.

Test-archiving systems

The goal of a test-archiving system is to keep track of series of tests and to act as the basis for documenting that the tests have been done and that no defects were found during the process.

A typical design involves establishing both procedures for handling files of test information and procedures for documenting which tests were run when and with what effect. Test archive systems are mainly developed on a system-specific/application-specific basis.

8.7.3 Characteristics of Modern Tools

The characteristics of modern software testing tools differ somewhat from previous systems. Modern tools are modular and highly adaptable to different environments. They also tend to make use of the facilities provided by most operating systems rather than provide those capabilities internally.

Case tools that support DFD, ERD, and structure chart diagramming, offer minimal data dictionary support, are available in the market. Project & Instaplan Management tools like MS Project, a configuration management tool like PVCS, and testing tool like Visual Text & flow graph generator, commercially software design tools like object modelling technique (OMT) and CTT development environment are also available in the market.

REFERENCES

- [BEIZZ90] Beizer B., "Software Testing Techniques", Van Nostrand Reinhold, New York, 1990.
- [BERT94] Bertolino A. & Marre M., "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs", IEEE Trans on Software Engineering, Vol. 20, No. 2, Dec. 1994.
- [BIEM94] Bieman J.M. & Off L.M., "Measuring Functional Cohesion", IEEE Trans. On Software Engineering, Vol. SE-20, No. 8, pp. 644–657, Aug. 1994.
- [COLL88] Collofello J.S., "Introduction to Software Verification and Validation", SEI-CM-13-1.1, Software Engineering Institute, Pittsburgh, P.A., USA.
- [DAHL72] Dahl O.J., Dijkstra E. W. & Hoare C.A.R., "Structure Programming", Academic Press, New York, 1972.
- [DEUT82] Deutsch M.S., "Software Verification and Validation", Prentice-Hall, Englewood Cliffs, NJ.
- [ELME73] Elmendorf W.R., "Cause-Effect Graphs in Functional Testing", Poughkeepsie, NY; IBM System Development Division TR-00. 2487, 1973.
- [FRIE95] Friedman M.A. & M. Voas Jeffrey, "Software Assessment", John Wiley & Sons, 1995.
- [GHEZ94] Ghezzi C. et al., "Fundamentals of Software Engineering", Prentice-Hall, 1994.
- [JALO96] Jalote P., "An Integrated Approach to Software Engineering", Narosa, Delhi, 1996.
- [JONE90] Jones G.W., "Software Engineering", John Wiley & Sons, 1990.
- [JORG95] Jorgensen P.C., "Software Testing: A Craftsman's Approach", CRC Press, USA, 1995.
- [MCCA76] McCabe T.J., "A Complexity Metric", IEEE Transactions on Software Engineering, SE-2,4, 308–320, December, 1976.
- [MILL72] Mills H.D., "Mathematical Foundations for Structured Programming", Federal System Division, IBM Corp, Gaithersburg, MD, FSC 72-6012, 1972.
- [MILL77] Miller E. F., "Tutorial: Program Testing Techniques", COMPSAC' 77, IEEE Computer Society, 1977.
- [MILL79] Miller E.F. & Howden W.E., "Software Testing & Validation Techniques", IEEE Computer Society, 1980.
- [MYER79] Myers G.J., "The Art of Software Testing", New York, Wiley, Interscience, 1979.
- [NORM89] Norman Parrington & Marc Roper, "Understanding Software Testing", John Wiley & Sons, 1989.
- [PATT01] Patton R., "Software Testing", Techmedia, 2001.
- [PETS85] Petschenik N.H., "Practical Properties in Software Testing", IEEE Software 2 (5): 18-23.

- [PRES97] Pressman R.S., "Software Engineering: A Practitioner's Approach", McGraw Hill, New York, 1997.
- [RAJA04] Rajani R. & Oak P., "Software Testing", Tata McGraw Hill, 2004.
- [SHNE80] Shneiderman B., "Software Psychology", Winthrop Publishers, 1980.
- [SHOO87] Shooman M.L., "Software Engineering", McGraw Hill, NY, 1987.
- [SOMM96] Sommerville I., "Software Engineering", Addison-Wesley, 1996.
- [TAMR03] Tamres L., "Introducing Software Testing", Pearson Education, 2003.
- [VICK84] Vick C.R. & Ramamoorthy C.V., "Handbook of Software Engineering", Van Nostrand Reinhold Company, New York.
- [WEIS82] Weiser M., "Program Slicing, Proc. of 5th Int. Conf. On Software Engineering, March 1981, pp. 439-449.

MULTIPLE CHOICE QUESTIONS

Note: Select most appropriate answer of the following questions.

- 8.1.** Software testing is:
- (a) the process of demonstrating that errors are not present
 - (b) the process of establishing confidence that a program does what it is supposed to do.
 - (c) the process of executing a program to show that it is working as per specifications
 - (d) the process of executing a program with the intent of finding errors.
- 8.2.** Software mistakes during coding are known as:
- | | |
|--------------|-------------|
| (a) failures | (b) defects |
| (c) bugs | (d) errors. |
- 8.3.** Functional testing is known as:
- | | |
|------------------------|------------------------|
| (a) Structural testing | (b) Behaviour testing |
| (c) Regression testing | (d) None of the above. |
- 8.4.** For a function of n variables, boundary value analysis yields:
- | | |
|-------------------------|-------------------------|
| (a) $4n + 3$ test cases | (b) $4n + 1$ test cases |
| (c) $n + 4$ test cases | (d) None of the above. |
- 8.5.** For a function of two variables, how many test cases will be generated by robustness testing ?
- | | |
|--------|---------|
| (a) 9 | (b) 13 |
| (c) 25 | (d) 42. |
- 8.6.** For a function of n variables robustness testing of boundary value analysis yields:
- | | |
|--------------|------------------------|
| (a) $4n + 1$ | (b) $4n + 3$ |
| (c) $6n + 1$ | (d) None of the above. |
- 8.7.** Regression testing is primarily related to:
- | | |
|-------------------------|--------------------------|
| (a) Functional testing | (b) Data flow testing |
| (c) Development testing | (d) Maintenance testing. |
- 8.8.** A node with indegree = 0 and outdegree $\neq 0$ is called
- | | |
|-------------------|------------------------|
| (a) Source node | (b) Destination node |
| (c) Transfer node | (d) None of the above. |
- 8.9.** A node with indegree $\neq 0$ and outdegree = 0 is called:
- | | |
|----------------------|------------------------|
| (a) Source node | (b) Predicate node |
| (c) Destination node | (d) None of the above. |

- 8.10.** A decision table has
(a) Four portions
(c) Five portions

8.11. Beta testing is carried out by
(a) Users
(c) Testers

8.12. Equivalence class partitioning is related to
(a) Structural testing
(c) Mutation testing

8.13. Cause-effect graphing technique is one form of
(a) Maintenance testing
(c) Function testing

8.14. During validation:
(a) Process is checked
(c) Developer's performance is evaluated

8.15. Verification is
(a) Checking the product with respect to customer's expectations
(b) Checking the product with respect to specifications
(c) Checking the product with respect to the constraints of the project.
(d) All of the above.

8.16. Validation is
(a) Checking the product with respect to customer's expectations
(b) Checking the product with respect to specification
(c) Checking the product with respect to constraints of the project
(d) All of the above.

8.17. Alpha testing is done by
(a) Customer
(c) Developer

8.18. Site for Alpha testing is
(a) Software company
(c) Any where

8.19. Site of Beta testing is
(a) Software company
(c) Any where

8.20. Acceptance testing is done by
(a) Developers
(c) Testers

8.21. One fault may lead to
(a) One failure
(c) Many failures

8.22. Test suite is
(a) Set of test cases
(c) Set of outputs

- 8.23.** Behavioural specifications are required for:
- (a) Modelling
 - (b) Verification
 - (c) Validation
 - (d) None of the above.
- 8.24.** During the development phase, the following testing approach is not adopted
- (a) Unit testing
 - (b) Bottom up testing
 - (c) Integration testing
 - (d) Acceptance testing.
- 8.25.** Which is not a functional testing technique?
- (a) Boundary value analysis
 - (b) Decision table
 - (c) Regression testing
 - (d) None of the above.
- 8.26.** Decision tables are useful for describing situations in which:
- (a) An action is taken under varying sets of conditions
 - (b) Number of combinations of actions are taken under varying sets of conditions
 - (c) No action is taken under varying sets of conditions
 - (d) None of the above.
- 8.27.** One weakness of boundary value analysis and equivalence partitioning is
- (a) They are not effective
 - (b) They do not explore combinations of input circumstances
 - (c) They explore combinations of input circumstances
 - (d) None of the above.
- 8.28.** In cause effect graphing technique, cause & effect are related to
- (a) Input and output
 - (b) Output and input
 - (c) Destination and source
 - (d) None of the above.
- 8.29.** DD Path graph is called as
- (a) Design to Design Path graph
 - (b) Defect to Defect Path graph
 - (c) Destination to Destination Path graph
 - (d) Decision to decision Path graph.
- 8.30.** An independent path is
- (a) Any path through the DD path graph that introduces at least one new set of processing statements or new conditions
 - (b) Any path through the DD Path graph that introduces at most one new set of processing statements or new conditions
 - (c) Any path through the DD Path graph that introduces one and only one new set of processing statements or new conditions.
 - (d) None of the above.
- 8.31.** Cyclomatic complexity is developed by
- (a) B.W. Boehm
 - (b) T.J. McCabe
 - (c) B.W. Littlewood
 - (d) Victor Basili.
- 8.32.** Cyclomatic complexity is denoted by
- (a) $V(G) = e - n + 2P$
 - (b) $V(G) = \Pi + 1$
 - (c) $V(G) = \text{Number of regions of the graph}$
 - (d) All of the above.
- 8.33.** The equation $V(G) = \Pi + 1$ of cyclomatic complexity is applicable only if every predicate node has
- (a) two outgoing edges
 - (b) three or more outgoing edges.
 - (c) No outgoing edges
 - (d) None of the above.

- 8.34.** The size of the graph matrix is
(a) Number of edges in the flow graph
(b) Number of nodes in the flow graph
(c) Number of paths in the flow graph
(d) Number of independent paths in the flow graph.
- 8.35.** Every node is represented by
(a) One row and one column in graph matrix
(b) two rows and two columns in graph matrix
(c) one row and two columns in graph matrix
(d) None of the above.
- 8.36.** Cyclomatic complexity is equal to
(a) Number of independent paths
(b) Number of paths
(c) Number of edges
(d) None of the above.
- 8.37.** Data flow testing is related to
(a) Data flow diagrams
(b) E-R diagrams
(c) Data dictionaries
(d) None of the above.
- 8.38.** In data flow testing, objective is to find
(a) All dc-paths that are not du-paths
(b) All du-paths
(c) All du-paths that are not dc-paths
(d) All dc-paths.
- 8.39.** Mutation testing is related to
(a) Fault seeding
(b) Functional testing
(c) Fault checking
(d) None of the above.
- 8.40.** The overhead code required to be written for unit testing is called
(a) Drivers
(b) Stubs
(c) Scaffolding
(d) None of the above.
- 8.41.** Which is not a debugging technique ?
(a) Core dumps
(b) Traces
(c) Print statements
(d) Regression testing.
- 8.42.** A break in the working of a system is called
(a) Defect
(b) Failure
(c) Fault
(d) Error.
- 8.43.** Alpha and Beta testing techniques are related to
(a) System testing
(b) Unit testing
(c) acceptance testing
(d) Integration testing.
- 8.44.** Which one is not the verification activity ?
(a) Reviews
(b) Path testing
(c) Walkthrough
(d) Acceptance testing.
- 8.45.** Testing the software is basically
(a) Verification
(b) Validation
(c) Verification and validation
(d) None of the above.
- 8.46.** Integration testing techniques are
(a) Topdown
(b) Bottom up
(c) Sandwich
(d) All of the above.

EXERCISES

- 8.1. What is software testing? Discuss the role of software testing during software life cycle and why is it so difficult?
 - 8.2. Why should we test? Who should do the testing?
 - 8.3. What should we test? Comment on this statement. Illustrate the importance of testing.
 - 8.4. Define the following terms:

(i) fault	(ii) failure
(iii) bug	(iv) mistake
 - 8.5. What is the difference between
 - (i) Alpha testing & beta testing
 - (ii) Development & regression testing
 - (iii) Functional & structural testing.
 - 8.6. Discuss the limitations of testing. Why do we say that complete testing is impossible?
 - 8.7. Briefly discuss the following
 - (i) Test case design, Test & Test suite
 - (ii) Verification & Validation
 - (iii) Alpha, beta & acceptance testing.
 - 8.8. Will exhaustive testing (even if possible for very small programs) guarantee that the program is 100% correct?
 - 8.9. Why does software fail after it has passed from acceptance testing? Explain.
 - 8.10. What are various kinds of functional testing? Describe any one in detail.
 - 8.11. What is a software failure? Explain necessary and sufficient conditions for software failure. Mere presence of faults means software failure. Is it true? If not, explain through an example, a situation in which a failure will definitely occur.
 - 8.12. Explain the boundary value analysis testing technique with the help of an example.
 - 8.13. Consider the program for the determination of next date in a calendar. Its input is a triple of day, month and year with the following range

$$1 \leq \text{month} \leq 12$$

$1 \leq \text{day} \leq 31$

$1900 \leq \text{year} \leq 2025$

The possible outputs would be Next date or invalid input date. Design boundary value, robust and worst test cases for this programs.

- 8.14.** Discuss the difference between worst test case and adhoc test case performance evaluation by means of testing. How can we be sure that the real worst case has actually been observed?
- 8.15.** Describe the equivalence class testing method. Compare this with boundary value analysis technique.
- 8.16.** Consider a program given below for the selection of the largest of numbers.

```
main()
{
    float A,B,C;
    printf("Enter three values\n");
    scanf("%f%f%f", &A,&B,&c);
    printf("\n Largest value is");
    if (A>B)
    {
        if(A>C)
            printf("%f\n",A);
        else
            printf("%f\n",c);
    }
    else
    {
        if(C>B)
            printf("%f\n",C);
        else
            printf("%f\n",B);
    }
}
```

(i) Design the set of test cases using boundary value analysis technique and equivalence class testing technique.

(ii) Select a set of test cases that will provide 100% statement coverage.

(iii) Develop a decision table for this program.

- 8.17.** Consider a small program and show, why is it practically impossible to do exhaustive testing?
- 8.18.** Explain the usefulness of decision table during testing. Is it really effective? Justify your answer.
- 8.19.** Draw the cause effect graph of the program given in exercise 8.16.
- 8.20.** Discuss cause effect graphing technique with an example
- 8.21.** Determine the boundary value test cases for the extended triangle problem that also considers right angle triangles.
- 8.22.** Why does software testing need extensive planning? Explain.
- 8.23.** What is meant by test case design? Discuss its objectives and indicate the steps involved in test case design.
- 8.24.** Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

<i>Marks obtained</i>	<i>Grade</i>
80–100	Distinction
60–79	First division
50–59	Second division
40–49	Third division
0–39	Fail

Generate test cases using equivalence class testing technique.

- 8.25. Consider a program to determine whether a number is ‘odd’ or ‘even’ and print the message
NUMBER IS EVEN

Or

NUMBER IS ODD

The number may be any valid integer.

Design boundary value & equivalence class test cases.

- 8.26. Admission to a professional course is subject to the following conditions:

(a) Marks in Mathematics >=	60
(b) Marks in Physics >=	50
(c) Marks in Chemistry >=	40
(d) Total in all three subjects >=	200

Or

Total in Mathematics and Physics >= 150

If aggregate marks of an eligible candidate are more than 225, he/she will be eligible for honours course, otherwise he/she will be eligible for pass course. The program reads the marks in the three subjects and generates the following outputs:

- (a) Not Eligible
- (b) Eligible to Pass Course
- (c) Eligible to Honours Course

Design test cases using decision table testing technique.

- 8.27. Draw the flow graph for program of largest of three numbers as shown in exercise 8.16. Find out all independent paths that will guarantee that all statements in the program have been tested.
- 8.28. Explain the significance of independent paths. Is it necessary to look for a tool for flow graph generation, if program size increases beyond 100 source lines?
- 8.29. Discuss the structural testing. How is it different from functional testing?
- 8.30. What do you understand by structural testing? Illustrate important structural testing techniques.
- 8.31. Discuss the importance of path testing during structural testing.
- 8.32. What is cyclomatic complexity? Explain with the help of an example.
- 8.33. Is it reasonable to define “thresholds” for software modules? For example, is a module acceptable if its $V(G) \leq 10$? Justify your answer.
- 8.34. Explain data flow testing. Consider an example and show all “du” paths. Also identify those “du” paths that are not “dc” paths.
- 8.35. Discuss the various steps of data flow testing.
- 8.36. If we perturb a value, changing the current value of 100 by 1000, what is the effect of this change? What precautions are required while designing the test cases?

- 8.37. What is the difference between white and black box testing? Is determining test cases easier in black or white box testing? Is it correct to claim that if white box testing is done properly, it will achieve close to 100% path coverage?
- 8.38. What are the objectives of testing? Why is the psychology of a testing person important?
- 8.39. Why does software fail after it has passed all testing phases? Remember, software, unlike hardware does not wear out with time.
- 8.40. What is the purpose of integration testing? How is it done?
- 8.41. Differentiate between integration testing & system testing.
- 8.42. Is unit testing possible or even desirable in all circumstances? Provide examples to Justify your answer?
- 8.43. Petschenik suggested that a different team than the one that does integration testing should carry out system testing. What are some good reasons for this?
- 8.44. Test a program of your choice, and uncover several program errors. Localise the main route of these errors, and explain how you found the courses. Did you use the techniques of Table 8.8? Explain why or why not.
- 8.45. How can design attributes facilitate debugging?
- 8.46. List some of the problems that could result from adding debugging statements to code. Discuss possible solutions to these problems.
- 8.47. What are various debugging approaches? Discuss them with the help of examples.
- 8.48. Researchers & practitioners have proposed several mixed testing strategies intended to combine advantages of the various techniques discussed in this chapter. Propose your own combination, perhaps also using some kind of random testing at selected points [GHEE94].
- 8.49. Design a test set for a spell checker. Then run it on a word processor having a spell checker, and report on possible inadequacies with respect to your requirements.
- 8.50. 4 GLs represent a major step forward in the development of automatic program generators. Explain the major advantages & disadvantages in the use of 4 GLs. What are the cost impacts of applications of testing and how do you justify expenditures for these activities.

Contents

- 9.1 What is Software Maintenance ?**
 - 9.1.1 Categories of Maintenance
 - 9.1.2 Problems during Maintenance
 - 9.1.3 Maintenance is Manageable
 - 9.1.4 Potential Solutions to Maintenance Problems
- 9.2 The Maintenance Process**
 - 9.2.1 Program Understanding
 - 9.2.2 Generating Particular Maintenance Proposal
 - 9.2.3 Ripple Effect
 - 9.2.4 Modified Program Testing
 - 9.2.5 Maintainability
- 9.3 Maintenance Models**
 - 9.3.1 Quick-fix Model
 - 9.3.2 Iterative Enhancement Model
 - 9.3.3 Reuse Oriented Model
 - 9.3.4 Boehm's Model
 - 9.3.5 Taute Maintenance Model
- 9.4 Estimation of Maintenance Costs**
 - 9.4.1 Belady and Lehman Model
 - 9.4.2 Boehm Model
- 9.5 Regression Testing**
 - 9.5.1 Development Testing Versus Regression Testing
 - 9.5.2 Regression Test Selection
 - 9.5.3 Selective Retest Techniques
- 9.6 Reverse Engineering**
 - 9.6.1 Scope and Tasks
 - 9.6.2 Levels of Reverse Engineering
 - 9.6.3 Reverse Engineering Tools
- 9.7 Software Re-engineering**
 - 9.7.1 Source Code Translation
 - 9.7.2 Program Restructuring
- 9.8 Configuration Management**
 - 9.8.1 Configuration Management Activities
 - 9.8.2 Software Versions
 - 9.8.3 Change Control Process
- 9.9 Documentation**
 - 9.9.1 User Documentation
 - 9.9.2 System Documentation
 - 9.9.3 Other Classification Schemes

Software maintenance is a task that every development group has to face when the software is delivered to the customer's site, installed and is operational. Therefore, delivery or release of software inaugurates the maintenance phase of the life cycle. The time spent and effort required keeping software operational after release is very significant and consumes about 40-70% of the cost of the entire life cycle. Despite the fact that it is very important and challenging task; it is routinely the poorly managed headache that nobody wants to do.

9.1 WHAT IS SOFTWARE MAINTENANCE

The term maintenance is a little strange when applied to software. In common speech, it means fixing things that break or wear out. In software, nothing wears out; it is either wrong from the beginning, or we decide later that we want to do something different. However, the term is so common that we must live with it [LAMB88].

Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization [STEP78]. Because change is inevitable, mechanisms must be developed for evaluating, controlling and making modifications. So any work done to change the software after it is in operation is considered to be maintenance work. The purpose is to preserve the value of software over time. The value can be enhanced by expanding the customer base, meeting additional requirements, becoming easier to use, more efficient and employing newer technology. Maintenance may span for 20 years, whereas development may be 1-2 years.

9.1.1 Categories of Maintenance

The only thing that remains constant in life is "CHANGE". As the specification of the computer systems change, reflecting changes in the external world, so must the systems themselves. More than two-fifths of maintenance activities are extensions and modifications requested by the users. There are four major categories of software maintenance, which are discussed below:

Corrective maintenance

This refers to modifications initiated by defects in the software. A defect can result from design errors, logic errors and coding errors. Design errors occur when changes made to the software are incorrect, incomplete, wrongly communicated or the change request is misunderstood. Logic errors result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow or incomplete test data. Coding errors are caused by incorrect

implementation of detailed logic design and incorrect use of the source code logic. Defects are also caused by data processing errors and system performance errors [LIEN80].

In the event of system failure due to an error, actions are taken to restore operation of the software system. Due to pressure from management, maintenance personnel sometimes resort to emergency fixes known as "patching" [BENN91]. The adhoc nature of this approach often gives rise to a range of problems that include increased program complexity and unforeseen ripple effects [TAKA96]. Unforeseen ripple effects imply that a change to one part of a program may affect other sections in an unpredictable manner, thereby leading to distortion in the logic of the system. This is often due to lack of time to carry out a through "impact analysis" before effecting the change.

Adaptive maintenance

It includes modifying the software to match changes in the ever-changing environment. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the software, for example, business rules, government policies, work patterns, software and hardware operating platforms. A change to the whole or part of this environment will require a corresponding modification of the software [BROO87].

Thus, this type of maintenance includes any work initiated as a consequence of moving the software to a different hardware or software platform-compiler, operating system or new processor. Any change in the government policy can have far-reaching ramifications on the software. When European countries had decided to go for "single European currency", this change affected all banking system software and was modified accordingly.

Perfective maintenance

It means improving processing efficiency or performance, or restructuring the software to improve changeability. When the software becomes useful, the user tends to experiment with new cases beyond the scope for which it was initially developed. Expansion in requirements can take the form of enhancement of existing system functionality or improvement in computational efficiency; for example, providing a Management Information System with a data entry Module or a new message handling facility [STRI82].

Hence, perfective maintenance refers to enhancements: making the product better, faster, smaller, better documented, cleaner structured, with more functions or reports.

Other types of maintenance

There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflects deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible. This work may be named as preventive maintenance. This term is often used with hardware systems and implies such things as lubrication of parts before need occurs, or automatic replacement of banks of light bulbs before they start to individually burn out. Since software does not degrade in the same way as hardware and does not need maintenance to retain the presently established level of functionality. Some authors do not use this term. That is why we have included this type of activity under "Other Types of Maintenance" category.

This activity is usually initiated from within the maintenance organization with the intention of making program easier to understand and hence facilitating future maintenance work. This includes code restructuring, code optimization and documentation updating. After a series of quick fixes to software, the complexity of its source code can increase to an unmanageable level, thus justifying complete restructuring of the code. Code optimization can be performed to enable the programs to run faster or to make more efficient use of storage. Updating user and system documentation, though frequently ignored, is often necessary when any part of software is changed. The documents affected by the change should be modified to reflect the current state of the system [TAKA96].

The requests come regularly to carry out maintenance activities. The request may be for corrective, adaptive or perfective maintenance. However most of the requests are for perfective maintenance. The distribution is shown in Fig. 9.1 [LIEN80].

“Other types of maintenance” is required, as mentioned earlier, to reduce the complexity of the code. This is not very high as shown in Fig. 9.1, because, we may not get any external requests for this activity and this is confined to maintenance organisation only.

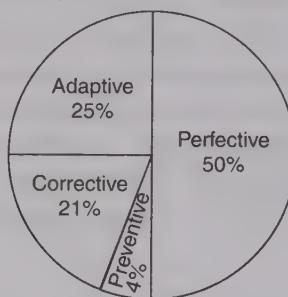


Fig. 9.1: Distribution of maintenance effort.

9.1.2 Problems During Maintenance

The most important problem during maintenance is that before correcting or modifying a program, the programmer must first understand it. Then, the programmer must understand the impact of the intended change. Few problems are discussed below [LEHM85, ARNO82]:

- Often the program is written by another person or group of persons working over the years in isolation from each other.
- Often the program is changed by person who did not understand it clearly, resulting in a deterioration of the program's original organization.
- Program listings, even those that are well organized, are not structured to support reading for comprehension. We normally read an article or book straight through, but with listing, programmer rummages back and forth.
- There is a high staff turnover within Information Technology industry. Due to this many systems are maintained by persons who are not the original authors. These persons may not have adequate knowledge about the system. This may mean that these persons may introduce changes to programs without being aware of their effects on other parts of the system—the ripple effect. This problem may be worsened by the absence of documentation. Even where it exists, it may be out of date or inadequate.

- Some problems only become clearer when a system is in use. Many users know what they want but lack the ability to express it in a form understandable to programmers/analysts. This is primarily due to information gap.
- Systems are not designed for change. If there is hardly any scope for change, maintenance will be very difficult. Therefore approach of development should be the production of maintainable software.

All these problems translate to a huge maintenance expenditure, which has been estimated ranging from 40% to 70% of the total cost during the life cycle of large scale software systems [STEP80].

9.1.3 Maintenance is Manageable

A common misconception about maintenance is that it is not manageable. We can not control it. Every fix is different, and it is like hysterical phone calls at 2.00 in the morning.

The data obtained by Lientz and Swanson [LIEN80] suggested other side of maintenance work. In their survey, they investigated the distribution of effort in software maintenance. The results are given in the Table 9.1.

Table 9.1: Distribution of maintenance effort

1	Emergency debugging	12.4%
2	Routine debugging	9.3%
3	Data environment adaptation	17.3%
4	Changes in hardware and OS	6.2%
5	Enhancements for users	41.8%
6	Documentation Improvement	5.5%
7	Code efficiency improvement	4.0%
8	Others	3.5%

It is clear from the Table 9.1, only 12.4% of effort is devoted to emergency debugging. Of course, getting a call at 2.00 in morning is more memorable than getting a request for an innocuous modification in a report format.

Another item in the survey of Lientz and Swanson asked about the kinds of maintenance requests that are made in the companies. These results are given in table 9.2.

Table 9.2: Kinds of maintenance requests

1	New reports	40.8%
2	Add data in existing reports	27.1%
3	Reformed reports	10%
4	Condense reports	5.6%
5	Consolidate reports	6.4%
6	Others	10.1%

The key lesson to learn from these studies is that most maintenance can be managed. Certainly the 79% in the adaptive, perfective and preventive categories can be anticipated, scheduled, monitored, estimated and managed. From the Table 9.1, we see that at least some of the corrective maintenance activities can also be managed. We may say, only about 10% of maintenance requests really require special and immediate handling. The other 90% requests can be handled in an organised and routine managerial system.

If we do proper and timely preventive maintenance, this 10% emergency requests may also be minimised. The preventive maintenance would involve assigning some time of maintenance persons to re-examine a product that has been modified several times to evaluate whether its structure can be reconstituted, cleaned up, or enhanced in anticipation of further maintenance. Preventive maintenance may cost, but it produces a real benefit to reduce the maintenance activities.

9.1.4 Potential Solutions to Maintenance Problems

A number of possible solutions to maintenance problems have been suggested. They include: budget and effort reallocation; complete replacement of existing systems; and enhancement of existing systems.

Budget and effort reallocation

It is now-a-days suggested that more time and resources should be invested in the development-specification and design of more maintainable systems rather than allocating resources to develop unmaintainable or difficult to maintain systems. The use of more advanced requirement specification approaches [BOTT94], design techniques and tools [ZEVG95], quality assurance procedures and standards such as ISOP 9000, CMM and maintenance standards are aimed at addressing this issue.

Complete replacement of the system

If maintaining an existing system costs as much as developing a new one, why not develop a new system from scratch. This point of view is understandable, but in practice it is not simple. The risk and costs associated with complete system replacement are very high. Corrective and preventive maintenance take place periodically at relatively small but incremental costs. Some organizations can afford to pay for these comparatively small maintenance charges while at the same time supporting more ambitious and financially demanding projects may not be possible.

The creation of another system is no guarantee that it will function better than the old one. On installation, errors may be found in functions which the old system performed correctly and such errors will need to be fixed as they are discovered.

Maintenance of existing system

Complete replacement of the system is not usually a viable option. An operational system in itself can be an asset to an organization in terms of the investment in technical knowledge and the working culture engendered. The current system may need to have the potential to evolve to a higher state, providing more sophisticated user-driven functionality, the capability of developing cutting edge technology, and of allowing the integration of other systems in a cost effective manner.

9.2 THE MAINTENANCE PROCESS

Once particular maintenance objective is established, the maintenance personnel must first understand what they are to modify. They must then modify the program to satisfy the maintenance objectives. After modification, they must ensure that the modification does not affect other portions of the program. Finally, they must test the program. These activities can be accomplished in the four phases as shown in Fig. 9.2 [STEP80].

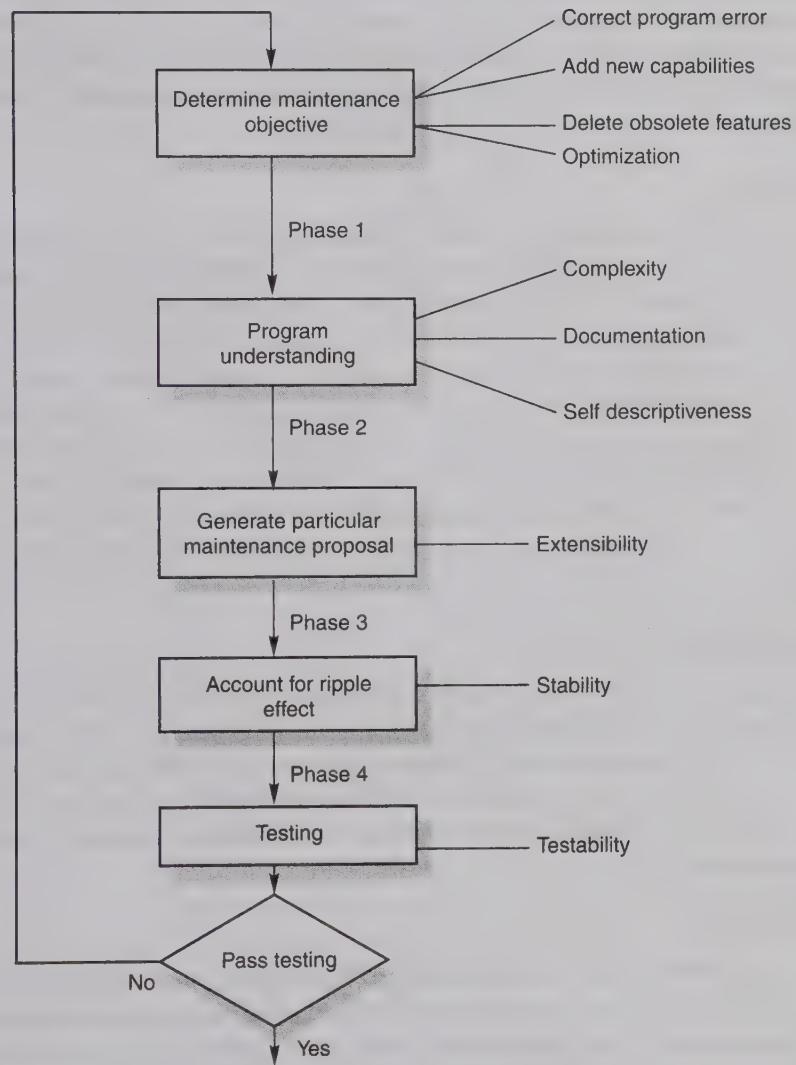


Fig. 9.2: The software maintenance process.

9.2.1 Program Understanding

The first phase consists of analyzing the program in order to understand it. Several attributes such as the complexity of the program, the documentation, and the self-descriptiveness of the program contribute to the ease of understanding the program. The complexity of the program

is a measure of the effort required to understand the program and is usually based on the control or data flow of the program. The self-descriptiveness of the program is a measure of how clear the program is, *i.e.*, how easy it is to read, understand, and use.

9.2.2 Generating Particular Maintenance Proposal

The second phase consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective. This requires a clear understanding of both the maintenance objective and the program to be modified. However, the ease of generating maintenance proposals for a program is primarily affected by the attribute extensibility. The extensibility of the program is a measure of the extent to which the program can support extensions of critical functions.

9.2.3 Ripple Effect

The third phase consists of accounting for all of the ripple effect as a consequence of program modifications. In software, the effect of a modification may not be local to the modification, but may also affect other portions of the program. There is a ripple effect from the location of the modification to the other parts of the programs that are affected by the modification [COLLO78]. One aspect of this ripple effect is logical or functional in nature. Another aspect of this ripple effect concerns the performance of the program. Since a large-scale program usually has both functional and performance requirements, it is necessary to understand the potential effect of a program modification from both a logical and a performance point of view. The primary attribute affecting the ripple effect as a consequence of a program modification is the stability of the program. Program stability is defined as the resistance to the amplification of changes in the program.

9.2.4 Modified Program Testing

The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before. It is important that cost-effective testing techniques be applied during maintenance. The primary factor contributing to the development of these cost-effective techniques is the testability of the program. Program testability is defined as a measure of the effort required to adequately test the program according to some well defined testing criterion.

9.2.5 Maintainability

Each of these four phases and their associated software quality attributes are critical to the maintenance process. All of these factors must be combined to form maintainability. How easy is it to maintain a program? To a large extent, that depends on how difficult the program is to understand. Program maintainability and program understandability are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain. And the more difficult it is to maintain, the higher its maintainability risk. Maintainability may be defined qualitatively as: the ease with which software can be understood, corrected, adapted, and/or enhanced.

9.3 MAINTENANCE MODELS

Maintenance is nothing but development and requires special skills. Many times, maintenance activities are performed without requirements or design documents. Sometimes it is also difficult to understand the old code. Therefore, the need of maintenance models has been recognized, but presently, the models are neither so well developed nor so well understood as models for software development.

9.3.1 Quick-fix Model

This is basically an adhoc approach to maintaining software. It is a fire fighting approach, waiting for the problem to occur and then trying to fix it as quickly as possible. The model is shown in the Fig. 9.3.

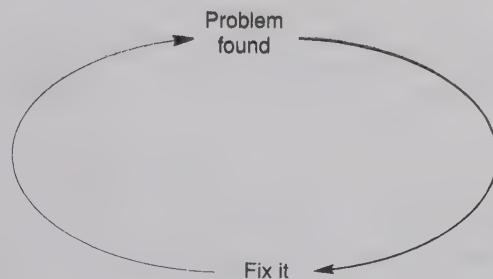


Fig. 9.3: The quick-fix model.

In this model, fixes would be done without detailed analysis of the long-term effects, for example, ripple effects through the software or effects on the code structure. There would be little, if any documentation. Where are the advantages of such a model and why it is still used? In an appropriate environment it can work perfectly well. If, for example, a system is developed and maintained by a single person, he or she can come to learn the system well enough to be able to manage without detailed documentation, to be able to make instinctive judgements about how and how not to implement change. The job gets done quickly and cheaply. This is normally used due to pressure of deadlines and resources.

If customers are demanding the correction of an error they may not be willing to wait for the organization to go through detailed and time-consuming stage of risk analysis. The organization may run a higher risk in keeping its customers waiting than it runs in going for the quickest fix. But what of the long-term problems?

If an organization relies on quick fix alone, it will run into difficult and very expensive problems, thus losing any advantage it gained from using the quick-fix model in the first place.

We should distinguish the short-term and long term upgrades. If a user finds a bug in a commercial word processor, for example, it would be unrealistic to expect a whole new upgrade immediately. Often, a company will release a quick fix as a temporary measure. The real solution will be implemented, along with other corrections and enhancements, as a major upgrade at a later date [TAKA96].

9.3.2 Iterative Enhancement Model

In this model, it has been proposed that the changes in the software system throughout its lifetime are an iterative process. Originally proposed as a development model but well suited

to maintenance, the motivation for this was the environment where requirements were not fully understood and a full system could not be built.

Adapted for maintenance, the model assumes complete documentation as it relies on modification of this as the starting point of each iteration. The model is effectively a three-stage cycle as shown in the Fig. 9.4.

- Analysis
- Characterization of proposed modifications
- Redesign and implementation

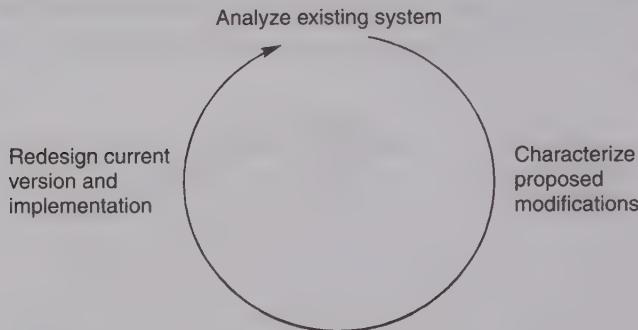


Fig. 9.4: The three-stage cycle of iterative enhancement.

The existing documentation of each stage (requirements, design, coding, testing and analysis) is modified starting with the highest-level document affected by the proposed changes. These modifications are propagated through the set of documents and the system redesigned. The model explicitly supports reuse and also accommodates other models, for example the quick-fix model.

The pressure of the maintenance environment often dictates that a quick solution is found but, as we have seen, the use of the quickest solution can lead to more problems than it solves. In the first iteration, problem areas would be identified and next iteration would specifically address the problem.

The problems with this model are the assumptions made about the existence of full documentation and the ability of the maintenance team to analyze the existing product in full. Wider use of structured maintenance models will lead to a culture where documentation tends to be kept up to date and complete, the current situation is that this is not often the case.

9.3.3 Reuse Oriented Model

This model is based on the principle that maintenance could be viewed as an activity involving the reuse of existing program components. The reuse model [BASI90] has four main steps:

- (i) Identification of the parts of the old system that are candidates for reuse.
- (ii) Understanding these system parts.
- (iii) Modification of the old system parts appropriate to the new requirements.
- (iv) Integration of the modified parts into the new system.

A detailed framework is required for the classification of components and the possible modifications. With the full reuse model the starting point may be any phase of the life cycle —

the requirements, the design, the code or the test data—unlike other models. (The model is shown in Fig. 9.5) For example, in the quick-fix model, the starting point is always the code.

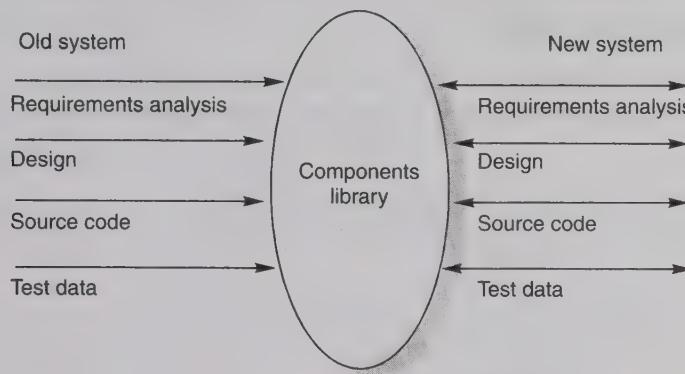


Fig. 9.5: The reuse model.

9.3.4 Boehm's Model

In 1983 Boehm [BOEH83] proposed a model for the maintenance process based upon the economic models and principles. Economic models are nothing new, economic decisions are a major driving force behind many processes and Boehm's thesis was that economic models and principles could not only improve productivity in the maintenance but also help understanding the process.

Boehm represents the maintenance process as a closed loop cycle as shown in Fig. 9.6. He theorizes that it is the stage where management decisions are made that drives the process. In this stage, a set of approved changes is determined by applying particular strategies and cost-benefit evaluations to a set of proposed changes. The approved changes are accompanied by their own budgets, which will largely determine the extent and type of resources expanded.

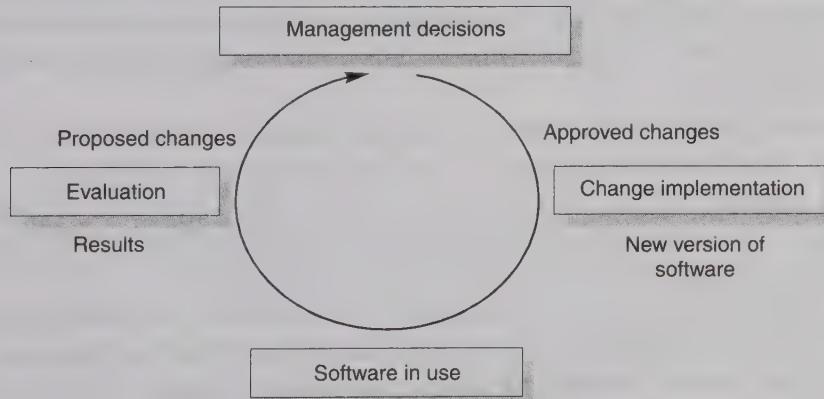


Fig. 9.6: Boehm's model.

Boehm sees the maintenance manager's task as one of balancing the pursuit of the objectives of maintenance against the constraints imposed by the environment in which

maintenance work is carried out. Thus the maintenance process is driven by the maintenance manager's decisions, which are based on the balancing of objectives against the constraints.

9.3.5. Taute Maintenance Model

The model was developed by B.J. Taute in 1983 and is very easy to understand and implement. It is a typical maintenance model and has eight phases in cycle fashion. The phases are shown in Fig. 9.7.

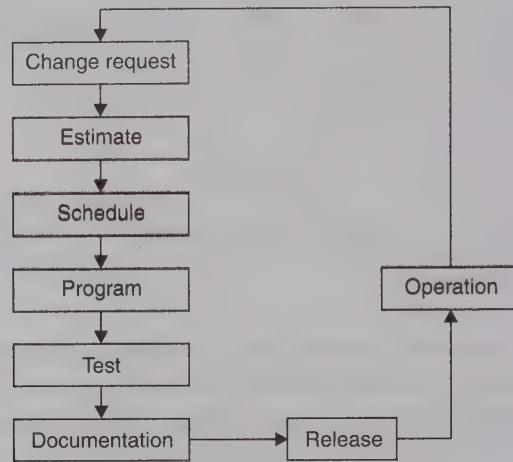


Fig. 9.7: Taute maintenance model.

(i) **Change request phase.** Maintenance team gets a request in a prescribed format from the client to make a change. This change may fall in any category of maintenance activities. We identify the type of request (*i.e.* corrective, adaptive, perfective or preventive) and assign a unique identification number to the request.

(ii) **Estimate Phase.** This phase is devoted to estimate the time and effort required to make the change. It is difficult to make exact estimates. But our objective is to have at least reasonable estimate of time and effort. Impact analysis on existing system is also required to minimise the ripple effect.

(iii) **Schedule phase.** We may like to identify change requests for the next scheduled release and may also prepare the documents that are required for planning.

(iv) **Programming phase.** In this phase, source code is modified to implement the requested change. All relevant documents like design document, manuals etc. are updated accordingly. Final output is the test version of the source code.

(v) **Test phase.** We would like to ensure that modification is correctly implemented. Hence, we test the code. We may use already available test cases and may also design new test cases. The term used for such testing is known as regression testing.

(vi) **Documentation phase.** After regression testing, system and user documents are prepared/ updated before releasing the system. This helps us to maintain co-relation between code and documents.

(vii) **Release phase.** The new software product alongwith updated documents are delivered to the customer. Acceptance testing is carried out by the users of the system.

(viii) **Operation phase.** After acceptance testing, software is placed under normal operation. During usage, when another problem is identified or new functionality requirement is felt or enhancement of existing capability is desired, again a 'Change request' process is initiated. If we do so, we may go back to change request phase and repeat all phases to implement the change.

9.4 ESTIMATION OF MAINTENANCE COSTS

We had earlier discussed that maintenance effort is very significant and consumes about 40-70% of the cost of the entire life cycle. However, this cost may vary widely from one application domain to another.

It is advisable to invest more effort in early phases of software life cycle to reduce the maintenance costs. The defect repair ratio increases heavily from analysis phase to implementation phase and given in the Table 9.3.

Table 9.3: Defect repair ratio

Phase	Ratio
Analysis	1
Design	10
Implementation	100

Therefore more effort during development will certainly reduce the cost of maintenance. Good software engineering techniques such as precise specification, loose coupling and configuration management all reduce maintenance costs.

9.4.1 Belady and Lehman Model

This model indicates that the effort and cost can increase exponentially if poor software development approach is used and the person or group that used the approach is no longer available to perform maintenance. The basic equation [BELA76] is given below:

$$M = P + K e^{(c-d)}$$

where

M : Total effort expended.

P : Productive effort that involves analysis, design, coding, testing and evaluation.

K : An empirically determined constant.

c : Complexity measure due to lack of good design and documentation.

d : Degree to which maintenance team is familiar with the software.

In this relation, the value of 'c' is increased if the software system is developed without use of a software engineering process. Of course, 'c' will be higher for a large software product with a high degree of systematic structure than a small one with the same degree. If the software is maintained without an understanding of the structure, function, and purpose of the software, then the value of 'd' will be low [SAGE90].

Example 9.1

The development effort for a software project is 500 person-months. The empirically determined constant (K) is 0.3. The complexity of the code is quite high and is equal to 8. Calculate the total effort expended (M) if

- (i) maintenance team has good level of understanding of the project ($d = 0.9$)
- (ii) maintenance team has poor understanding of project ($d = 0.1$).

Solution

$$\text{Development effort (P)} = 500 \text{ PM}$$

$$K = 0.3$$

$$C = 8$$

- (i) Maintenance team has good level of understanding ($d = 0.9$)

$$\begin{aligned} M &= P + K e^{(c-d)} \\ &= 500 + 0.3 e^{(8-0.9)} \\ &= 500 + 363.59 = 863.59 \text{ PM} \end{aligned}$$

- (ii) Maintenance team has poor level of understanding ($d = 0.1$)

$$\begin{aligned} M &= P + K e^{(c-d)} \\ &= 500 + 0.3 e^{(8-0.1)} \\ &= 500 + 809.18 = 1309.18 \text{ PM} \end{aligned}$$

Hence, it is clear that effort increases exponentially, if poor software engineering approaches are used and understandability of the project is poor.

9.4.2 Boehm Model

Boehm [BOEH81] proposed a formula for estimating maintenance costs as part of his COCOMO Model. Using data gathered from several projects, this formula was established in terms of effort. Boehm used a quantity called Annual Change Traffic (ACT) which is defined as:

"The fraction of a software product's source instructions which undergo change during a year either through addition, deletion or modification".

The ACT is clearly related to the number of change requests.

$$ACT = \frac{KLOC_{\text{added}} + KLOC_{\text{deleted}}}{KLOC_{\text{total}}}$$

The Annual Maintenance Effort (AME) in person-months can be calculated as:

$$AME = ACT \times SDE$$

where, SDE : Software development effort in person-months.

ACT : Annual change Traffic

Suppose a software project required 400 person-months of development effort and it was estimated that 25% of the code would be modified in a year. The AME will be $0.25 \times 400 = 100$ person_month. Boehm suggested that this rough estimate should be refined by judging the importance of factors affecting the cost and selecting the appropriate cost multipliers. Using these factors, Effort Adjustment Factor (EAF) should be calculated as in the case of COCOMO model. The modified equation is given below:

$$AME = ACT * SDE * EAF$$

Example 9.2

Annual Change Traffic (ACT) for a software system is 15% per year. The development effort is 600 PMs. Compute an estimate for Annual Maintenance Effort (AME). If life time of the project is 10 years, what is the total effort of the project?

Solution

The development effort = 600 PM

Annual Change Traffic (ACT) = 15%

Total duration for which effort is to be calculated = 10 years.

The maintenance effort is a fraction of development effort and is assumed to be constant.

$$\text{AME} = \text{ACT} \times \text{SDE}$$

$$= 0.15 \times 600 = 90 \text{ PM}$$

$$\text{Maintenance effort for 10 years} = 10 \times 90 = 900 \text{ PM.}$$

$$\text{Total effort} = 600 + 900 = 1500 \text{ PM.}$$

Example 9.3

A software project has development effort of 500 PM. It is assumed that 10% code will be modified per year. Some of the cost multipliers are given as:

(i) Required software Reliability (RELY) : high

(ii) Date base size (DATA) : high

(iii) Analyst capability (ACAP) : high

(iv) Application experience (AEXP) : Very high

(v) Programming language experience (LEXP) : high

Other multipliers are nominal. Calculate the Annual Maintenance Effort (AME).

Solution

Annual change traffic (ACT) = 10%

Software development effort (SDE) = 500 PM

Using Table 4.5 of COCOMO model, effort adjustment factor can be calculated given below:

RELY = 1.15

ACAP = 0.86

AEXP = 0.82

LEXP = 0.95

DATA : 1.08

Other values are nominal values. Hence,

$$\text{EAF} = 1.15 \times 0.86 \times 0.82 \times 0.95 \times 1.08 = 0.832$$

$$\text{AME} = \text{ACT} * \text{SDE} * \text{EAF}$$

$$= 0.1 * 500 * 0.832 = 41.6 \text{ PM}$$

$$\text{AME} = 41.6 \text{ PM}$$

9.5 REGRESSION TESTING

Software inevitably changes, how so ever well written and designed it may be initially. This changed software is required to be retested in order to ensure that changes work correctly and these changes have not adversely affected other parts of the software. This is necessary because small changes in one part of a software may have subtle undesired effects in other seemingly unrelated parts of the software.

When we develop software, we use development testing to obtain confidence in the correctness of the software. Development testing involves constructing a test plan that describes how should we test the software, and then, designing and running suite of test cases that satisfy the requirements of the test plan. When we modify software, we typically retest it. This retesting is called regression testing. Hence, “Regression testing is the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested code”.

Therefore, regression testing tests both the modified code and other parts of the program that may be affected by the program change. It serves many purposes such as to:

- increase confidence in the correctness of the modified program
- locate errors in the modified program
- preserve the quality and reliability of software
- ensure the software's continued operation.

9.5.1 Development Testing Versus Regression Testing

We typically think of regression testing as a software maintenance activity; however, we also perform regression testing during the latter stages of software development. This latter stage starts after we have developed test plans and test suites and used them initially to test the software.

During this stage of development, we fine tune the code and correct errors in it, hence our activities resemble maintenance activities. The comparision of both is given in Table 9.4.

Table 9.4: Comparision of development and regression testing techniques

Sr. No.	<i>Development testing</i>	<i>Regression testing</i>
1.	We create test suites and test plans	We can make use of existing test suites and test plans.
2.	We test all software components.	We retest affected components that have been modified by modifications.
3.	Budget gives time for testing.	Budget often does not give time for regression testing.
4.	We perform testing just once on a software product.	We perform regression testing many times over the life of the software product.
5.	Performed under the pressure of release date of the software.	Performed in crisis situations, under greater time constraints.

9.5.2 Regression Test Selection

Regression testing is very expensive activity and consumes significant amount of effort/cost. Many techniques are available to reduce this effort/cost. Simplest is to reuse the test suite that was used to test the original version of the software. Re-running all test cases in the test-suite, however, may still require excessive time.

An improvement is to reuse the existing test suite, but to apply a regression test selection technique to select an appropriate subset of the test suite to be run. If the subset is small enough, significant savings in time are achieved. To date, a number of regression test selection techniques have been developed for use in testing procedural and object oriented languages.

Testing professionals are reluctant, however, to omit from a test suite any test case that might expose a fault in the modified software. Consider, for example, the code fragments and associated test cases in Fig. 9.8 and Fig. 9.9 respectively. In Fig. 9.8, fragment B represents an erroneously modified version of fragment A, in which statement S_5 is modified.

We execute test cases t_3 and t_4 , where both execute statements S_5 and S'_5 . Test case t_3 causes a divide by zero problem in S'_5 , whereas test case t_4 does not.

Fragment A		Fragment B (modified form of A)	
S_1	$y = (x - 1) * (x + 1)$	S'_1	$y = (x - 1) * (x + 1)$
S_2	if ($y = 0$)	S'_2	if ($y = 0$)
S_3	return (error)	S'_3	return (error)
S_4	else	S'_4	else
S_5	return $\left(\frac{1}{y}\right)$	S'_5	return $\left(\frac{1}{y - 3}\right)$

Fig. 9.8: code fragments A and B

Test cases		
Test number	Input	Execution History
t_1	$x = 1$	S_1, S_2, S_3
t_2	$x = -1$	S_1, S_2, S_3
t_3	$x = 2$	S_1, S_2, S_5
t_4	$x = 0$	S_1, S_2, S_5

Fig. 9.9: Test cases for code fragment A of Fig. 9.8

If we execute all test cases, we will detect this divide by zero fault. But we have to minimise the test suite. From the Fig. 9.9, it is clear that test cases t_3 and t_4 have the same execution history i.e. S_1, S_2, S_5 . If few test cases have the same execution history ; minimization methods select only one test case. Others may be selected for coverage elsewhere in the code. Hence, either t_3 or t_4 will be selected. If we select t_4 , we lose the opportunity to expose the fault that t_3 exposes.

Hence minimisation methods can omit some test cases that might expose fault in the modified software and so, they are not safe. We should be careful in the process of minimisation of test cases and always try to use safe regression test selection technique.

A safe regression test selection technique is one that, under certain assumptions, selects every test case from the original test suite that can expose faults in the modified program [ROTH96].

9.5.3 Selective Retest Techniques

Selective retest techniques differ from the “retest-all” technique, which reruns all tests in the existing test suite.

Selective retest technique may be more economical than the “retest-all” technique if the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that the selective retest technique lets us omit.

Selective retest techniques partition an existing test suite into subsets containing reusable, retestable, and obsolete test cases. Reusable test cases exercise only unmodified code and unmodified specifications, and do not need to run, but may be saved for reuse in subsequent testing sessions. Retestable test cases exercise modified code or test modified specification and should be repeated. Obsolete test cases either (*i*) specify incorrect input-output relations due to specification modifications or (*ii*) no longer exercise the program components or specifications they were designed to exercise. The test cases that are obsolete for the first reason are called specification-obsolete test cases, and the test cases that are obsolete for the second reason are called coverage-obsolete test cases. Because it may be difficult to recognize obsolete test cases, we may list some test cases as unclassified. In addition to reclassifying existing tests, selective retest techniques may create, or recommend creation of, new-structural or new-specification test cases, that test the program constructs or specification items which are not covered by existing test cases. Selective retest techniques are broadly classified in three categories :

1. Coverage techniques: They are based on test coverage criteria. They locate coverable program components that have been modified, and select test cases that exercise these components.

2. Minimization techniques: They work like coverage techniques, except that they select minimal sets of test cases.

3. Safe techniques: They do not focus on coverage criteria ; instead they select every test, case that causes a modified program to produce different output than its original version.

Rothermal [ROTH96a] identified categories in which regression test selection techniques can be compared and evaluated. These categories are:

- inclusiveness
- precision
- efficiency
- generality.

Inclusiveness measures the extent to which a technique chooses test cases that will cause the modified program to produce different output than the original program, and thereby expose faults caused by modifications.

Precision measures the ability of a technique to avoid choosing test cases that will not cause the modified program to produce different output than the original program.

Efficiency measures the computational cost, and thus, practicality, of a technique.

Generality measures the ability of a technique to handle realistic and diverse language constructs, arbitrarily complex modifications, and realistic testing applications.

Evaluation and comparision of existing techniques helps us to choose appropriate techniques for particular applications.

For example, if we require very reliable code, we may insist on a safe selective technique regardless of the cost. On the other hand, if we want to reduce the testing time, we may choose minimisation technique, even though in doing so we may fail to select some test cases that expose faults in the modified program.

9.6 REVERSE ENGINEERING

Reverse Engineering is the process followed in order to find difficult, unknown and hidden information about a software system. It is becoming important, since several software products lack proper documentation, and are highly unstructured, or their structure has degraded through a series of maintenance efforts. Maintenance activities cannot be performed without a complete understanding of the software system.

Apparently, understanding software looks a trivial task, but in the absence of any external documentation or clues, the task often becomes almost impossible.

9.6.1 Scope and Tasks

Since the main purpose of reverse engineering is to recover information from the existing code or any other intermediate documents, any activity that requires program understanding at any level may fall within the scope of reverse engineering. What is achieved as a result of any reverse engineering activity varies according to what is going to be done with the extracted information. The areas where reverse engineering is applicable include (but not limited to): (i) program comprehension, (ii) redocumentation and/or document generation (iii) recovery of design approach and design details at any level of abstraction, (iv) identifying reusable components, (v) identifying components that need restructuring, (vi) recovering business rules, and (vii) understanding high-level system description. Processes attempting to re-design, re-structure and enhance functionality of a system are not within the scope of reverse engineering.

Reverse Engineering encompasses a wide array of tasks related to understanding and modifying software systems. This array of tasks can be broken into a number of classes. A few of these classes are briefly discussed below:

- Mapping between application and program domains: Computer programs are representations of problem situations from some application domain. The programs usually do not contain any hint about the problem. The task of the reverse engineer is to reconstruct the mapping from the application domain to the program domain as shown in Fig. 9.10.

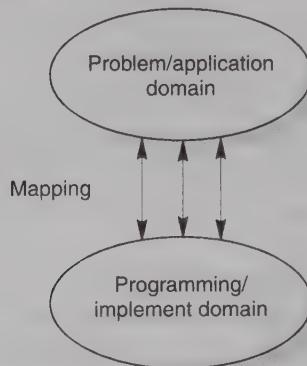


Fig. 9.10: Mapping between application and domains program.

- **Mapping between concrete and abstract levels:** Software development process follows from high-level abstraction to more detailed design and concrete implementation. A reverse engineer has to move backward and create an abstract representation of the implementation from the mass of concrete details.
- **Rediscovering high-level structures:** A program is the embodiment of a well-defined purpose and coherent high-level structure. However, the purpose and structure may be lost in course of time, and through maintenance activities, such as bug fixing, porting, modifying and enhancement. One of the tasks of reverse engineering is to detect the purpose and high-level structure of a program when the original one may have changed and where, in fact, there may be no such specific purpose left in the program.
- **Finding missing links between program syntax and semantics:** Computer programs are formal, in the sense they have well-defined syntax and semantics. In the formal world, the meaning of a syntactically correct program determines the output for a specific input. But systems that require reverse engineering generally would have lost their original semantics. Moreover, certain languages, such as the object-oriented languages, do not have strong formal basis. Reverse engineering process should determine the semantics of a given program from its syntax.
- **To extract reusable component:** Based on the premise that the use of existing program components can lead to an increase in productivity and improvement in product quality [BIGG89], the concept of reuse has increasingly become popular amongst software engineers. Success in reusing components depends in part on their availability. Reverse engineering tools and methods offer the opportunity to access and extract program components.

9.6.2 Levels of Reverse Engineering

Reverse Engineers detect low-level implementation constructs and replace them with their high level counterparts. The process eventually results in an incremental formation of an overall architecture of the program. It should, nonetheless, be noted that the product of a reverse engineering process does not necessarily have to be at a higher level of abstraction. If it is at the same level as the original system, the operation is commonly known as "redocumentation" [CHIK90]. If on the other hand, the resulting product is at a higher level of

abstraction, the operation is known as “design recovery” [BIGG89] or specification recovery as shown in Fig. 9.11 [TAKA96].

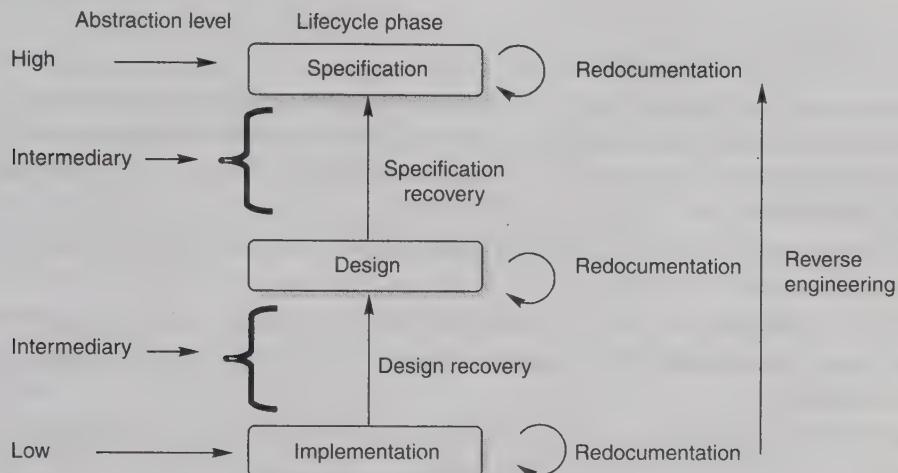


Fig. 9.11: Levels of abstraction.

Redocumentation

Redocumentation is the recreation of a semantically equivalent representation within the same relative abstraction level [CHIK 90]. The goals of this process are threefold. Firstly, to create alternative views of the system so as to enhance understanding, for example the generation of a hierarchical data flows or control flow diagram from source code. Secondly, to improve current documentation. Ideally, such documentation should have been produced during the development of the system and updated as the system changed. This, unfortunately, is not usually the case. Thirdly, to generate documentation for a newly modified program. This is aimed at facilitating future maintenance work on the system; preventive maintenance.

Design recovery

Design recovery entails identifying and extracting meaningful higher-level abstractions beyond those obtained directly from examination of the source code [CHIK90]. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains [BIGG89]. The recovered design—which is not necessarily the original design—can then be used for redeveloping the system. In other words, the resulting design forms a baseline for future system modifications [GILLI90]. The design could also be used to develop similar but non-identical applications. For example, after recovering the design of a spelling check(er) application, it can be used in the design of a spell-checking module in a new word processing package.

Different approaches, which vary in their focus, can be used to recover these designs. Some draw heavily on programming language constructs contained in the program text as seen in the model by Rugaber et al. [RUGA90]. They argue that an important aspect of design recovery is being able to recognize, understand and represent design decisions present in a given source code. Program constructs—which vary between programming languages—enable

recognition of design decisions. Examples of such constructs are control and data structures, variables, procedures and functions, definition and implementation modules, and class hierarchies.

9.6.3 Reverse Engineering Tools

Reverse engineering is performed with the help of certain tools, otherwise doing the job manually would require enormous amount of time and labour, rendering the whole purpose of it not only useless but also expensive. Rugaber [RUGA94] identifies the following four basic components of a reverse engineering tool:

- The Restructurer detects poorly structured code fragments and replaces them by equivalent structured code.
- The Cross Referencer lists the places where each variable is defined and used.
- The Static Analyzer detects anomalous constructs such as uninitialized variables and dead code.
- The Text Editor and other simple tools support browsing and editing of the source code.

In addition to these, a reverse engineering tool may incorporate graphical user-interface and other visualization tools. In the following subsections, we shall look at two reverse engineering tools.

RIGI

The Rigi System [MULL92] is an interactive graph editor that provides a graphical representation of software systems. The Rigi approach to reverse engineering includes the following phases:

- Extracting system components and relationships from source code to create resource-flow graphs. The extraction phase is initially automatic and involves parsing the source code and storing the extracted artifacts (*i.e.*, datatypes, functions, dependencies) in a repository. The rest of the phases in Rigi are semi-automatic in the sense it requires user intervention.
- Creating subsystems from the flow-graphs using graph editor. Subsystem hierarchies are built on top of the initial call graphs by using the subsystem composition methodology, and subsystems are grouped into composite subsystems. These hierarchies can be documented as views, which are snapshots of the various reverse engineering states. Computing interfaces between subsystems by analyzing and propagating the dependencies extracted from the source code.
- Evaluating subsystems for cohesion and coupling and iterating until satisfactory subsystems have been created. Rigi is useful for identifying hot-spots for maintenance as well as candidate modules for reengineering. This information does not provide insight to the tool user on how to restructure the modules. The dynamic aspects of a software system are not modeled in Rigi and therefore it is doubtful how well it would perform for object-oriented languages.

"Refine" language tools

Reasoning Systems, inc. markets the Refine Language Tools, a family of interactive, extensible workbenches for analyzing and reengineering code in programming languages including Ada, C, COBOL, and FORTRAN. One of the most attractive features of these tools is that they are customizable to suit any particular version of the programming language. Each Refine Language Tool comes with a fully documented reengineering API (Application Programming Interface) for building customizations. Refine Language Tools use an intuitive X Window System based graphical interface and provide capabilities including: (i) Interactive source code navigation, (ii) Generation of set/use, structure chart, and identifier definition reports, (iii) Online viewing and Postscript printing of all reports, (iv) Consistent graphical user interface and a standardized report format, and (v) Ability to export design information to forward-engineering CASE tools.

Reverse engineering plays a very significant role in software maintenance. Good tools can make the task of maintaining software less expensive and less troublesome. Experience and studies show that more than two-thirds of the money spent for a software system is chewed up by maintenance activities. In this backdrop, the role and importance of reverse engineering attains a very high standing. If a reverse engineering tool can reduce the maintenance cost by even a marginal amount, say 5%, it would save huge amount of money.

Since reverse engineering is applicable at any phase of software development or any level of abstraction, it can be effectively employed at strategic points during software development process. It would provide important feedback and enable the developers to go back, rectify faults, and design and implement more efficient software. Although this would increase development cost but eventually it would save much more from maintenance expenses.

9.7 SOFTWARE RE-ENGINEERING

Systems that have been in the field for a long time evolve and mutate in ways that were never planned. As enhancements are added, bugs fixed, and piece-meal solutions tacked on, the once elegant system grows into something unmanageable and expensive to maintain.

These old systems that must still be maintained are sometimes called legacy systems. These legacy systems may have been the best that technology had to offer. But, as mission requirements change and better technology becomes available, the users are faced with the difficult problem of reconciling the large investments they have already made in their existing systems against the promise of better designs, lower cost of maintenance, cheaper technology and large improvements in capabilities. Most of the legacy systems may be poorly structured and their documentation may be either out of date or non-existent. The developers of these systems having left the organization; there may be no one in the organization who really understands these systems in detail. These systems were also not designed for change. Another problem is the non-availability of requirements, design and test cases.

Software re-engineering is concerned with taking existing legacy systems and re-implementing them to make them more maintainable. As a part of this re-engineering process, the system may be redocumented or restructured. It may be translated to a more modern programming language, implemented on existing hardware technology. Thus, software re-engineering allows us to translate source code to a new language, restructure our old code,

migrate to a new platform (such as client-server), capture and then graphically display design information, and re-document poorly documented systems.

The critical distinction between re-engineering and new software development is the starting point for the development as shown in Fig. 9.12. Rather than start with a written specification, the old system acts as a specification for the new system.

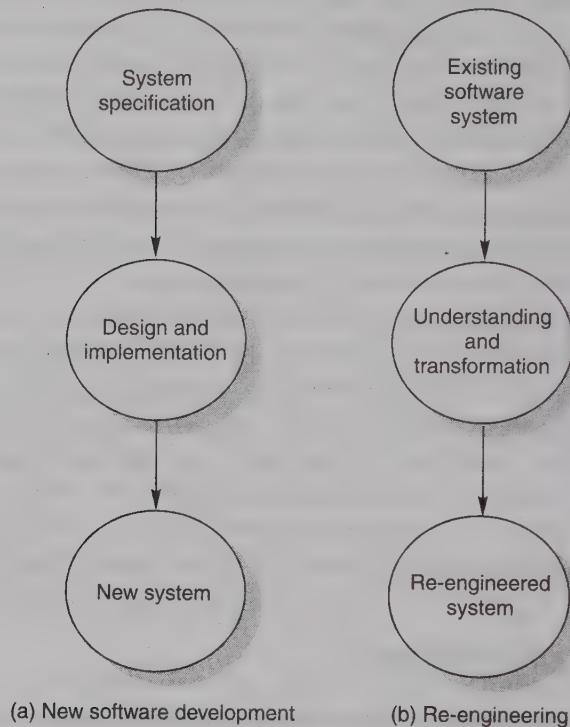


Fig. 9.8: Comparison of new software development with re-engineering.

The costs of re-engineering depend on the extent of the work that is carried out. Other factors affecting costs are; the quality of the software, tool support available, extent of data conversion, availability of expert staff.

The alternative to re-engineering a software system is to redevelop that system using modern software engineering techniques. Where systems are very badly structured this may be the only viable option as the re-engineering costs for these systems are likely to be high.

The following suggestions may be useful for the modification of the legacy code:

- Study code well before attempting changes.
- Concentrate on overall control flow and not coding.
- Heavily comment internal code.
- Create Cross References
- Build Symbol tables
- Use own variables, constants and declarations to localize the effect of change.
- Keep detailed maintenance document.
- Use modern design techniques.

9.7.1 Source Code Translation

Simplest method is to translate source code to another programming language. The target language may be an updated version of the original language (Basic to Visual Basic) or may be a completely different language (FORTRAN to JAVA). Source level translation may be required for the following reasons [SOMM00]:

- (i) **Hardware platform update:** The organization may wish to change its standard hardware platform. Compilers for the original language may not be available on the new platform.
- (ii) **Staff Skill Shortages:** There may be lack of trained maintenance staff for the original language. This is a particular problem where programs were written in some non-standard language that has now gone out of general use.
- (iii) **Organisational policy changes:** An organization may decide to standardize on a particular language to minimize its support software costs. Maintaining many versions of old compilers can be very expensive.

Translation can be done either manually or using an automation tool. One of the most sophisticated tools available to assist this process is the REFINE system [MARK94] that incorporates powerful pattern matching and program transformation capabilities. It is therefore possible to define patterns in the source code, which should be converted. REFINE recognizes these and replaces them with the new constructs. Although, manual intervention is almost always required to tune and improve the general system.

9.7.2 Program Restructuring

This involves transforming a system from one representational form to another without a change in the semantics or functionality. The transformation would usually be to a more desirable format.

Due to maintenance activities, the programs become complex and difficult to understand. To control this increase in complexity, the source code needs to be restructured. There are various types of restructuring techniques and some are discussed below:

- (i) **Control flow driven restructuring:** This involves the imposition of a clear control structure within the source code and can be either inter modular or intra-modular in nature. This makes the program more readable and easier to understand. However, the program may still suffer from a lack of modularity whereby related components of the program may be dispersed through the code.
- (ii) **Efficiency driven restructuring:** This involves restructuring a function or algorithm to make it more efficient. A simple example is the replacement of an IF-THEN-ELSE-IF-ELSE construct with a CASE construct as shown in Fig. 9.13. With the CASE statement, only one boolean variable is evaluated, whereas with the IF-THEN-ELSE-IF-ELSE construct, more than one of the boolean expressions may need to be tested during execution thereby making it less efficient.

<pre> IF Score > = 75 THEN Grade: = 'A' ELSE IF Score > = 60 THEN Grade: = 'B' ELSE IF Score > = 50 THEN Grade: = 'C' ELSE IF Score > = 40 THEN Grade: = 'D' ELSE IF Grade = 'F' END </pre> <p style="text-align: center;">(a)</p>	<pre> CASE Score of 75, 100: Grade: = 'A' 60, 74: Grade: = 'B'; 50, 59: Grade: = 'C'; 40, 49: Grade: = 'D'; ELSE Grade: = 'F' END </pre> <p style="text-align: center;">(b)</p>
--	---

Fig. 9.13: Restructuring a program

(iii) **Adaption-driven restructuring:** This involves changing the coding style in order to adapt the program to a new programming language or new operating environment, for instance changing an imperative program in PASCAL into a functional program in LISP. Another example is the transformation of a program functions in a sequential environment to an equivalent but totally different form of processing in a parallel environment.

9.8 CONFIGURATION MANAGEMENT

The software may be considered as configurations of software components. These software components are released in the form of executable code whereas supplier organization keeps the source code. This source code is the representation of an executable equivalent, but which one? Source Code can be modified without there being any effect upon executable versions in use and, if strict controls are not kept, the source code which is the exact representation of a particular executable version may no longer exist. The means by which the process of software development and maintenance is controlled is called configuration management. The configuration management is different in development and maintenance phases of life cycle due to different environments. Software maintenance is undertaken in the environment of a live system in use by a probably large user base. Potential effects upon a live system are much more immediate than those upon a system still under development. Thus, configuration management is concerned with the development of procedures and standards for cost effective managing and controlling changes in an evolving software system.

9.8.1 Configuration Management Activities

The activities are divided into four broad categories [TAKA96].

1. The identification of the components and changes.
2. The control of the way by which the changes are made.
3. Auditing the changes.
4. Status accounting-recording and documenting all the activities that have taken place.

The following documents are required for these activities:

- Project plan
- Software requirements specification document

- Software design description document
- Source code listing
- Test plans/procedures/test cases
- User manuals

All components of the system's configuration are recorded along with all relationships and dependencies between them. Any change-addition, deletion or modification-must be recorded and its effect upon the rest of the system's components should be checked. After a change has been made, a new configuration is recorded. There is a need to know who is responsible for every procedure and process along the way and it is a management task both to assign these responsibilities and to conduct audits to see that they are carried out.

9.8.2 Software Versions

During software maintenance, there will be at least two versions of the software system; the old version and the new version(s). Since a software system is comprised of software components, there will also be two or more versions of each component that has been changed. Thus, the software maintenance team has to cope with multiple versions of the software. We can distinguish between two types of versions namely revisions (replace) and variations (variety).

Version Control: During the process of software evolution, many objects are produced, for example files, electronic documents, paper documents, source code, executable code and bitmap graphics. A version control tool is the first stage towards being able to manage multiple versions. Once it is in place, a detailed record of every version of the software must be kept. This comprises the

- Name of each source code component, including the variations and revisions,
- The versions of the various compilers and linkers used,
- The name of the software staff who constructed the component.
- The date and the time at which it was constructed.

9.8.3 Change Control Process

It will be appropriate if changes to software can be predicted. Change control process comes into effect when the software and associated documentation are delivered to configuration management change request form (as shown in Fig. 9.14), which should record the recommendations regarding the change. The recommendations may include assessment of the proposed change, the estimated costs and how the change should be implemented. This form is submitted to a Change Control Authority (CCA), which decides whether or not the change is to be accepted. If change is approved by the CCA, it is applied to the software. The revised software is revalidated by the Software Quality Assurance (SQA) team to ensure that the change has not adversely affected other parts of the software. The changed software is handed over to the software configuration team and is incorporated in a new version of the system.

CHANGE REQUEST FORM	
Project ID:	
Change Requester with date:	
Requested change with date:	
Change analyzer:	
Components affected:	
Associated components:	
Estimated change costs:	
Change priority:	
Change assessment:	
Change implementation:	
Date submitted to CCA:	
Date of CCA decision:	
CCA decision:	/
Change implementer:	
Date submitted to QA:	
Date of implementation:	
Date submitted to CM:	
QA decision:	

Fig. 9.14: Change request form

9.9 DOCUMENTATION

Software documentation is the written record of facts about a software system recorded with the intent to convey purpose, content and clarity. The recording process usually begins when the need for the system is conceived and continues until the system is no longer in use.

There are different categories of software documentation like user documentation, system documentation, etc.

9.9.1 User Documentation

It refers to those documents, containing descriptions of the functions of a system without reference to how these functions are implemented. A list of user documentation is given in Table 9.5.

Table 9.5: User documentation

S. No.	Document	Function
1.	System Overview	Provides general description of system's functions
2.	Installation Guide	Describes how to set up the system, customize it to local hardware needs and configure it to particular hardware and other software systems.
3.	Beginner's Guide	Provides simple explanations of how to start using the system.
4.	Reference Guide	Provides in-depth description of each system facility and how it can be used.
5.	Enhancement	Booklet Contains a summary of new features.
6.	Quick reference card	Serves as a factual lookup.
7.	System administration	Provides information on services such as net-working, security and upgrading.

9.9.2 System Documentation

It refers to those documentation containing all facets of system, including analysis, specification, design, implementation, testing, security, error diagnosis and recovery. System documentation details are given in Table 9.6.

9.9.3 Other Classification Schemes

There are other ways in which documentation may be classified. There may be three levels of documentation: user manuals, operator's manuals and maintenance manuals. User manual describes what the system does without necessarily going into the details of how it does it or how to get the system to do it. The operator's manual describes how to use the system as well as giving instructions on how to recover from faults. The maintenance manual contains details of the functional specification, design, code listing, test data and results.

Table 9.6: System documentation

S. No.	Document	Function
1.	System Rationale	Describes the objectives of the entire system.
2.	SRS	Provides information on exact requirements of system as agreed between user and developer
3.	Specification/Design	Provides description of: (i) How system requirements are implemented. (ii) How the system is decomposed into a set of interacting program units. (iii) The function of each program unit.
4.	Implementation	Provides description of: (i) How the detailed system design is expressed in some formal programming language. (ii) Program actions in the form of intra program comments.

(Contd.)...

S. No.	Document	Function
5.	System Test Plan	Provides description of how program units are tested individually and how the whole system is tested after integration
6.	Acceptance Test Plan	Describes the tests that the system must pass before users accept it.
7.	Data Dictionaries	Contains description of all terms that relate to the software system in question.

This classification and user/system documentation schemes are identical in the sense that they both include all the information that is contained in software documents.

REFERENCES

- [ARNO82] Arnold R.S., "The Dimensions of Healthy Maintenance", IEEE Software Engineering, 1982.
- [BASI90] Basili V.R., "Viewing Software Maintenance as Reuse-Oriented Software Development", IEEE Software, 7, January, 19-25, 1990.
- [BEIZ90] Beizer B., "Software Testing Techniques", Van Nostrand Reinhold, New York NY, 1990.
- [BELA76] Belady L. & Lehman W., "A Model of Large Program Development", IBM Systems Journal, Vol. 15, No. 3, 225-252, 1976.
- [BENN91] Bennett K. et al., "Software Maintenance", Butterworth-Heinemann Ltd., Oxford, 1991.
- [BIGG89] Biggerstaff T.J., "Design Recovery for Maintenance and Reuse", Computer, 22(7), July, 36-49, 1989.
- [BOEH81] Boehm B.W., "Software Engineering Economics", Prentice Hall, Englewood Cliffs, New Jersey, 1981.
- [BOEH83] Boehm B.W., "The Economic of Software Maintenance", Proc. Workshop on Software Maintenance", Silver Spring, MD, IEEE Computer Society Press, 9-37, 1983.
- [BOTT94] Bottaci L. & Jones J.G., "Formal Specification on Using Z: A Modeling Approach", Int. Thomson Publishing, London, 1994.
- [BROO87] Brooks R., "No Silver Bullet-Essence and Accidents of Software Engineering", IEEE Computer, 20 (4), 10-20, 1987.
- [CHIK90] Chikofsky E.J. & Cross Jr. J.H., "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, 7, January, 13-17, 1990.
- [COLLO78] Collofello J.S. et al., "Ripple Effect Analysis of Software Maintenance", Proc. COMPSAC 78, 60-65, 1978.
- [GILLI90] Gillis K.D. & Wright D.G., "Improving Software Maintenance using System Level Reverse Engineering", In Proc. IEEE Conference on Software Maintenance", 84-90, LA, IEEE Computer Society Press, 1990.
- [LAMB88] Lamb D.A., "Software Engineering: Planning for Change", Prentice Hall, Englewood Cliffs, NJ, 1988.
- [LEHM85] Lehman M.M., "Program Evolution", Academic Press, London, 1985.
- [LIEN80] Lientz B.P. & Swanson E.B., "Software Maintenance Management", Addison-Wesley Publishing Company, Reading, Massachusetts, 1980.
- [MARK94] Morkosian L. et al. "Using an Enabling Technology to Reengineer Legacy Systems", Communications of the ACM, 34(5), 58-70, May, 1994.
- [MULL92] Muller H.A. et al., "A Reverse Engineering Environment Based on Spital & Visual Software Interconnection Models", Software Engineering Notes, 17(5), Dec, 34, 88-98, 1992.

- [ROTH96] Rothermel G. & Harrold M.J., "Analyzing Regression Test Selection Techniques", IEEE Transactions on Software Engineering 22 (8), 529-551, August 1996.
 - [ROTH96 a] Rothermel R., "Efficient Effective Regression Testing Using Safe Test Selection Techniques", Ph. D Thesis, Clemson University, May, 1996.
 - [RUGA90] Rugaber S. et al., "Recognizing Design Decisions in Programs," IEEE Software, 7(1), January, 46-54, 1990.
 - [RUGA94] Rugaber S., "White Paper on Reverse Engineering", Tech Report, Georgia Institute of Technology, March, 1994.
 - [SAGE90] Sage A. & Palmer J.D., "Software Systems Engineering", John Wiley & Sons, 1990.
 - [SOMM00] Sommerville Ian, "Software Engineering", Addison-Wesley, 2000.
 - [STEP78] Stephen S.Y. et al., "Ripple Effect Analysis of Software Maintenance", in Proc. COMPSAC78, 1978.
 - [STEP80] Stephen S.Y. & Collofello J.S., "Some Stability Measures for Software Maintenance", IEEE Software Engg., 1980.
 - [STEP80] Stephen S.Y. & Collofello J.S., "Some Stability Measures for Software Maintenance", IEEE Trans on Software Engineering, 28-35, 1980.
 - [STRI82] Strickland J.P. et al., "An Evolving System", IBM Systems Journal, 21 (4), 490-513, 1982.
 - [TAKA96] Takang A.A., Grubb P.A., "Software Maintenance", Thomson Computer Press, U.K.
 - [TAUT 83] Taute B.J., 'Quality Assurance and Maintenance Application Systems', Proc. of the National AFIPS computer conference, PP 123-129, 183.
 - [ZVEG95] Zvezintzov N., "Software Management Technology Reference Guide: 1994/95 European Edition", Software Maintenance News, Inc., Los Alamitos, California, 1995.

MULTIPLE CHOICE QUESTIONS

Note: Choose most appropriate answer of the following questions.

- (c) Modification in software due to increase in complexity
 - (d) Modification in software to match changes in the ever-changing environment.

9.7. Perfective maintenance refers to enhancements

 - (a) Making the product better
 - (b) Making the product faster and smaller
 - (c) Making the product with new functionalities
 - (d) All of the above.

9.8. As per distribution of maintenance effort, which type of maintenance has consumed maximum share?

 - (a) Adaptive
 - (b) Corrective
 - (c) Perfective
 - (d) Preventive.

9.9. As per distribution of maintenance effort, which type of maintenance has consumed minimum share?

 - (a) Adaptive
 - (b) Corrective
 - (c) Perfective
 - (d) Preventive.

9.10. Which one is not a maintenance model?

 - (a) CMM
 - (b) Iterative Enhancement model
 - (c) Quick-fix model
 - (d) Reuse-Oriented model.

9.11. In which model, fixes are done without detailed analysis of the long-term effects?

 - (a) Reuse oriented model
 - (b) Quick-fix model
 - (c) Taute maintenance model
 - (d) None of the above.

9.12. Iterative enhancement model is a

 - (a) three stage model
 - (b) two stage model
 - (c) four stage model
 - (d) seven stage model.

9.13. Taute maintenance model has

 - (a) two phases
 - (b) six phases
 - (c) eight phases
 - (d) ten phases.

9.14. In Boehm model, ACT stands for

 - (a) Actual change time
 - (b) Actual change traffic
 - (c) Annual change traffic
 - (d) Annual change time.

9.15. Regression testing is known as

 - (a) the process of retesting the modified parts of the software
 - (b) the process of testing the design documents
 - (c) the process of reviewing the SRS
 - (d) None of the above.

9.16. The purpose of regression testing is to

 - (a) increase confidence in the correctness of the modified program
 - (b) locate errors in the modified program
 - (c) preserve the quality and reliability of software
 - (d) All of the above.

9.17. Regression testing is related to

 - (a) maintenance of software
 - (b) development of software
 - (c) both (a) and (b).
 - (d) none of the above.

EXERCISES

- 9.1. What is software maintenance? Describe various categories of maintenance. Which category consumes maximum effort and why?
 - 9.2. What are the implications of maintenance for a one person software production organisation?
 - 9.3. Some people feel that “maintenance is manageable”. What is your opinion about this issue?
 - 9.4. Discuss various problems during maintenance. Describe some solutions to these problems.
 - 9.5. Why do you think that the mistake is frequently made of considering software maintenance inferior to software development?
 - 9.6. Explain the importance of maintenance. Which category consumes maximum effort and why?
 - 9.7. Explain the steps of software maintenance with help of a diagram.
 - 9.8. What is self descriptiveness of a program? Explain the effect of this parameter on maintenance activities.
 - 9.9. What is ripple effect? Discuss the various aspects of ripple effect and how does it affect the stability of a program?
 - 9.10. What is maintainability? What is its role during maintenance?
 - 9.11. Describe Quick-fix model. What are the advantages and disadvantages of this model?

- 9.12. How iterative enhancement model is helpful during maintenance? Explain the various stages cycles of this model.
- 9.13. Explain the Boehm's maintenance model with the help of a diagram.
- 9.14. State the various steps of reuse oriented model. Is it a recommended model in object oriented design?
- 9.15. Describe the Taute maintenance model. What are various phases of this model?
- 9.16. Write a short note on Belady and Lehman model for the calculation of maintenance effort.
- 9.17. Describe various maintenance cost estimation models.
- 9.18. The development effort for a project is 600 PMs. The empirically determined constant (K) of Belady and Lehman model is 0.5. The complexity of code is quite high and is equal to 7. Calculate the total effort expended (M) if maintenance team has reasonable level of understanding of the project ($d = 0.7$).
- 9.19. Annual change traffic (ACT) in a software system is 25% per year. The initial development cost was Rs. 20 lacs. Total life time for software is 10 years. What is the total cost of the software system?
- 9.20. What is regression testing? Differentiate between regression and development testing.
- 9.21. What is the importance of regression test selection? Discuss with the help of examples.
- 9.22. What are selective retest techniques? How are they different from "retest-all" technique?
- 9.23. Explain the various categories of retest techniques. Which one is not useful and why?
- 9.24. What are the categories to evaluate regression test selection techniques? Why do we use such categorisation?
- 9.25. What is reverse engineering? Discuss levels of reverse engineering.
- 9.26. What are the appropriate reverse engineering tools? Discuss any two tools in detail.
- 9.27. Discuss reverse engineering and re-engineering.
- 9.28. What is re-engineering? Differentiate between re-engineering and new development.
- 9.29. Discuss the suggestions that may be useful for the modification of the legacy code.
- 9.30. Explain various types of restructuring techniques. How does restructuring help in maintaining a program?
- 9.31. Explain why single entry, single exit modules make testing easier during maintenance.
- 9.32. What are configuration management activities? Draw the performance of change request form.
- 9.33. Explain why the success of a system depends heavily on the quality of the documentation generated during system development.
- 9.34. What is an appropriate set of tools and documents required to maintain large software product?
- 9.35. Explain why a high degree of coupling among modules can make maintenance very difficult?
- 9.36. Is it feasible to specify maintainability in the SRS? If yes, how would we specify it?
- 9.37. What tools and techniques are available for software maintenance? Discuss any two of them.
- 9.38. Why is maintenance programming becoming more challenging than new development? What are desirable characteristics of a maintenance programmer?
- 9.39. Why little attention is paid to maintainability during design phase?
- 9.40. List out system documentation and also explain their purpose.

Answers

CHAPTER 1

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 1.1. (a) | 1.2. (c) | 1.3. (d) | 1.4. (a) | 1.5. (b) | 1.6. (c) |
| 1.7. (b) | 1.8. (a) | 1.9. (b) | 1.10. (d) | 1.11. (a) | 1.12. (d) |
| 1.13. (a) | 1.14. (a) | 1.15. (b) | 1.16. (d) | 1.17. (b) | 1.18. (d) |
| 1.19. (d) | 1.20. (a) | | | | |

CHAPTER 2

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 2.1. (b) | 2.2. (c) | 2.3. (d) | 2.4. (c) | 2.5. (b) | 2.6. (c) |
| 2.7. (a) | 2.8. (b) | 2.9. (a) | 2.10. (c) | 2.11. (a) | 2.12. (b) |
| 2.13. (d) | 2.14. (d) | 2.15. (c) | 2.16. (b) | 2.17. (d) | 2.18. (c) |
| 2.19. (d) | 2.20. (b) | | | | |

CHAPTER 3

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 3.1. (c) | 3.2. (d) | 3.3. (a) | 3.4. (a) | 3.5. (b) | 3.6. (d) |
| 3.7. (d) | 3.8. (d) | 3.9. (c) | 3.10. (d) | 3.11. (b) | 3.12. (a) |
| 3.13. (a) | 3.14. (c) | 3.15. (b) | 3.16. (a) | 3.17. (d) | 3.18. (b) |
| 3.19. (d) | 3.20. (a) | | | | |

CHAPTER 4

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 4.1. (d) | 4.2. (c) | 4.3. (d) | 4.4. (b) | 4.5. (c) | 4.6. (a) |
| 4.7. (b) | 4.8. (b) | 4.9. (a) | 4.10. (b) | 4.11. (a) | 4.12. (d) |
| 4.13. (c) | 4.14. (b) | 4.15. (d) | 4.16. (a) | 4.17. (b) | 4.18. (c) |
| 4.19. (d) | 4.20. (b) | 4.21. (d) | 4.22. (c) | 4.23. (a) | 4.24. (b) |
| 4.25. (c) | | | | | |

CHAPTER 5

- | | | | | | |
|-----------------|-----------------|-----------------|------------------|------------------|------------------|
| 5.1. (b) | 5.2. (a) | 5.3. (c) | 5.4. (b) | 5.5. (c) | 5.6. (c) |
| 5.7. (a) | 5.8. (d) | 5.9. (a) | 5.10. (b) | 5.11. (a) | 5.12. (a) |

CHAPTER 6

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 6.1. (d) | 6.2. (a) | 6.3. (a) | 6.4. (b) | 6.5. (c) | 6.6. (d) |
| 6.7. (a) | 6.8. (a) | 6.9. (c) | 6.10. (d) | 6.11. (d) | 6.12. (a) |
| 6.13. (b) | 6.14. (a) | 6.15. (d) | | | |

CHAPTER 7

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 7.1. (d) | 7.2. (a) | 7.3. (d) | 7.4. (d) | 7.5. (b) | 7.6. (d) |
| 7.7. (c) | 7.8. (d) | 7.9. (a) | 7.10. (c) | 7.11. (d) | 7.12. (d) |
| 7.13. (b) | 7.14. (c) | 7.15. (a) | 7.16. (d) | 7.17. (c) | 7.18. (b) |
| 7.19. (a) | 7.20. (b) | 7.21. (a) | 7.22. (d) | 7.23. (c) | 7.24. (b) |
| 7.25. (a) | 7.26. (c) | 7.27. (a) | 7.28. (a) | 7.29. (c) | 7.30. (b) |
| 7.31. (d) | 7.32. (b) | 7.33. (b) | 7.34. (d) | 7.35. (c) | 7.36. (c) |
| 7.37. (a) | 7.38. (c) | 7.39. (c) | 7.40. (d) | 7.41. (a) | 7.42. (d) |
| 7.43. (c) | 7.44. (c) | 7.45. (a) | 7.46. (c) | 7.47. (a) | 7.48. (a) |
| 7.49. (a) | 7.50. (b) | | | | |

CHAPTER 8

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 8.1. (d) | 8.2. (c) | 8.3. (b) | 8.4. (b) | 8.5. (b) | 8.6. (c) |
| 8.7. (d) | 8.8. (a) | 8.9. (c) | 8.10. (a) | 8.11. (a) | 8.12. (b) |
| 8.13. (c) | 8.14. (d) | 8.15. (b) | 8.16. (a) | 8.17. (a) | 8.18. (a) |
| 8.19. (b) | 8.20. (b) | 8.21. (d) | 8.22. (a) | 8.23. (b) | 8.24. (d) |
| 8.25. (c) | 8.26. (b) | 8.27. (b) | 8.28. (a) | 8.29. (d) | 8.30. (a) |
| 8.31. (b) | 8.32. (d) | 8.33. (a) | 8.34. (b) | 8.35. (a) | 8.36. (a) |
| 8.37. (d) | 8.38. (c) | 8.39. (a) | 8.40. (c) | 8.41. (d) | 8.42. (b) |
| 8.43. (c) | 8.44. (d) | 8.45. (c) | 8.46. (d) | 8.47. (b) | 8.48. (b) |
| 8.49. (d) | 8.50. (b) | | | | |

CHAPTER 9

- | | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| 9.1. (c) | 9.2. (d) | 9.3. (b) | 9.4. (a) | 9.5. (a) | 9.6. (d) |
| 9.7. (d) | 9.8. (c) | 9.9. (d) | 9.10. (a) | 9.11. (b) | 9.12. (a) |
| 9.13. (c) | 9.14. (c) | 9.15. (a) | 9.16. (d) | 9.17. (c) | 9.18. (d) |
| 9.19. (d) | 9.20. (a) | 9.21. (d) | 9.22. (c) | 9.23. (d) | 9.24. (b) |
| 9.25. (b) | | | | | |

Affiliated Books
Distribution
Services ©
1970 Ginn & Co.

SOFTWARE ENGINEERING

Programs • Documentation • Operating Procedures

This book is designed as a textbook for the first course in Software Engineering for undergraduate and postgraduate students. This may also be helpful for software professionals to help them practice the software engineering concepts.

The Second Edition is an attempt to bridge the gap between "What is taught in the classroom" and "What is practiced in the industry". The concepts are discussed with the help of real life examples and numerical problems.

This book explains the basic principles of Software Engineering in a clear and systematic manner. A contemporary approach is adopted throughout the book. After introducing the fundamental concepts, the book presents a detailed discussion of software requirements analysis & specifications. Various norms and models of Software Project Planning are discussed next, followed by a comprehensive account of Software Metrics.

Suitable examples, illustrations, exercises, multiple choice questions and answers are included throughout the book to facilitate an easier understanding of the subject.

Prof. K.K. Aggarwal is the Vice-Chancellor of Guru Gobind Singh Indraprastha University, Delhi. He graduated in Electronics and Communication Engineering from Punjab University and obtained Masters degree in Advanced Electronics from Kurukshetra University securing *First position in both*. Later, he did his Ph.D. in Reliability Evaluation and Optimization also from Kurukshetra University. In 1975, he rose to the level of *Professor at an age of 27½ years*, probably the youngest person in the world to have achieved this level. He has been the Chairman of the Department of Electronics, Communications & Computer Engineering at Regional Engineering College, Kurukshetra for a long time and worked as Dean (Academic) for that institution. He was also Director, "Centre for Excellence for Manpower Development in Reliability Engineering" established by the Ministry of Human Resource Development at that College. Before taking up the *present assignment* in December 1998, Prof. Aggarwal was *Pro Vice-Chancellor*, Guru Jambheshwar University (Technical University of Haryana), Hisar for a period of three years.

He has been *President of the Institution of Electronics and Telecommunication Engineers (IETE)* for the period 2002-2004.

Prof. Yogesh Singh is a Professor & the Dean of University School of Information Technology and also the Dean of University School of Engineering & Technology, Guru Gobind Singh Indraprastha University, Delhi. He received his M.Tech. and Ph.D. (Computer Engineering) degrees from National Institute of Technology, Kurukshetra (previously known as Regional Engineering College, Kurukshetra). Prior to this, he was Founder Chairman, Department of Computer Science & Engineering, Guru Jambheshwar University, Hisar, Haryana. His area of research is Software Engineering focusing on Planning, Testing, Metrics and Neural Networks. He has more than 150 publications in International/National Journals and Conferences.

He was the Principal Investigator of the successfully implemented MHRD-AICTE Project entitled "*Experimentation & Development of Software Reliability & Complexity Measurement Techniques*". He is a member of IT-Task force and a member of its Core-group on E-Education, Govt. of NCT of Delhi.

ISBN 81-224-1638-1



PUBLISHING FOR ONE WORLD

9 788122 416381

NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS

New Delhi • Bangalore • Chennai • Cochin • Guwahati • Hyderabad

Jalandhar • Kolkata • Lucknow • Mumbai • Ranchi

www.newagepublishers.com