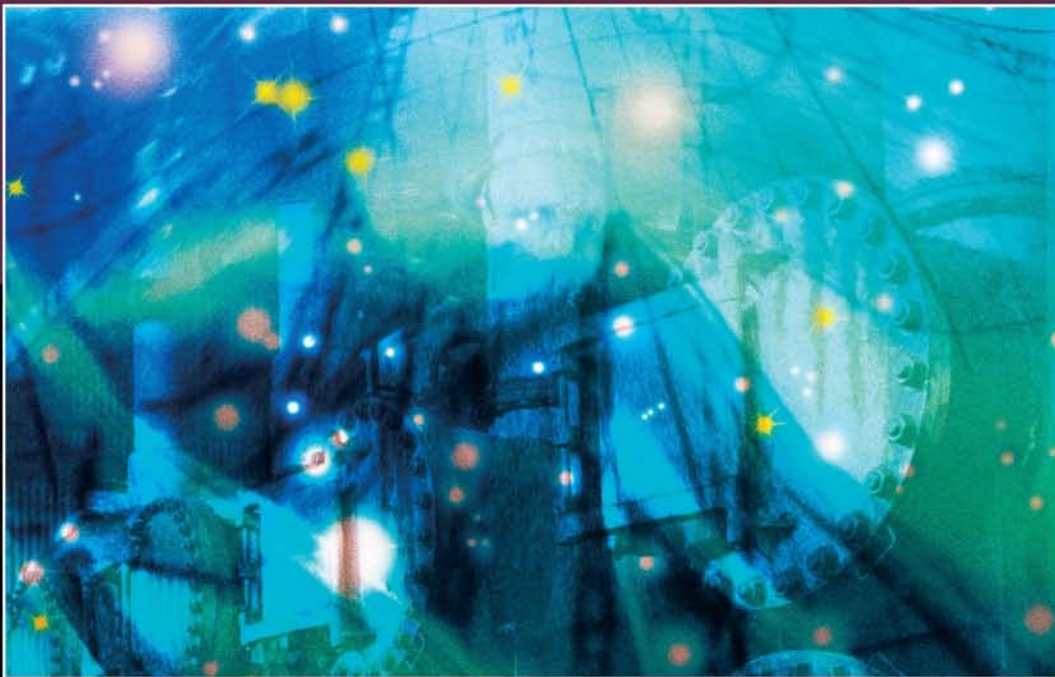


SOFTWARE ENGINEERING & TESTING

B. B. Agarwal, S. P. Tayal, M. Gupta



C O M P U T E R S C I E N C E S E R I E S

SOFTWARE ENGINEERING & TESTING

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

The CD-ROM that accompanies this book may only be used on a single PC. This license does not permit its use on the Internet or on a network (of any kind). By purchasing or using this book/CD-ROM package (the “Work”), you agree that this license grants permission to use the products contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained on the CD-ROM. Use of third party software contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the publisher or the owner of the software in order to reproduce or network any portion of the textual material or software (in any media) that is contained in the Work.

Jones and Bartlett Publishers, LLC (“the Publisher”) and anyone involved in the creation, writing, or production of the accompanying algorithms, code, or computer programs (“the software”) or any of the third party software contained on the CD-ROM or any of the textual material in the book, cannot and do not warrant the performance or results that might be obtained by using the software or contents of the book. The authors, developers, and the publisher have used their best efforts to insure the accuracy and functionality of the textual material and programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold “as is” without warranty (except for defective materials used in manufacturing the disc or due to faulty workmanship).

The authors, developers, and the publisher of any third party software, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or the CD-ROM, and only at the discretion of the Publisher.

The use of “implied warranty” and certain “exclusions” vary from state to state, and might not apply to the purchaser of this product.

SOFTWARE ENGINEERING & TESTING

An Introduction

**B. B. AGARWAL
S. P. TAYAL
M. GUPTA**



JONES AND BARTLETT PUBLISHERS

Sudbury, Massachusetts

BOSTON TORONTO LONDON SINGAPORE

World Headquarters

Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jbpub.com
www.jbpub.com

Jones and Bartlett Publishers
Canada
6339 Ormindale Way
Mississauga, Ontario L5V 1J2
Canada

Jones and Bartlett Publishers
International
Barb House, Barb Mews
London W6 7PA
United Kingdom

Jones and Bartlett's books and products are available through most bookstores and online booksellers. To contact Jones and Bartlett Publishers directly, call 800-832-0034, fax 978-443-8000, or visit our website www.jbpub.com.

Substantial discounts on bulk quantities of Jones and Bartlett's publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones and Bartlett via the above contact information or send an email to specialsales@jbpub.com.

Copyright © 2010 by Jones and Bartlett Publishers, LLC
Original Copyright © 2008 by Laxmi Publications Pvt. Ltd.

All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarked or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc., is not an attempt to infringe on the property of others.

ISBN: 978-1-934015-55-1

Cover Design: Tyler Creative

Library of Congress Cataloging-in-Publication Data

Agarwal, B. B.

Software engineering and testing / B.B. Agarwal, S.P. Tayal, M. Gupta.
p. cm.

ISBN 978-1-934015-55-1 (hardcover)

ISBN 978-0-7637-8302-0 (e)

1. Software engineering. 2. Computer software--Testing. I. Tayal, S.P.
II. Gupta, M. (Mahesh) 1975- III. Title.
QA76.758.A3945 2010
005.1--dc22

2008055318

6048 0569

Printed in the United States of America

13 12 11 10 09 10 9 8 7 6 5 4 3 2 1

TABLE OF CONTENTS

PART I: SOFTWARE ENGINEERING AND TESTING

Chapter 1. Introduction to Software Engineering	3
1.1 Introduction to Software	3
1.2 Types of Software	5
1.3 Classes of Software	8
1.4 Introduction to Software Engineering	9
1.5 Software Components	11
1.6 Software Characteristics	12
1.7 Software Crisis	13
1.8 Software Myths	15
1.9 Software Applications	15
1.10 Software-Engineering Processes	18
1.11 Evolution of Software	20
1.12 Comparison of Software Engineering and Related Fields	22
1.13 Some Terminologies	25
1.14 Programs Versus Software Products	26
Chapter 2. Software-Development Life-Cycle Models	29
2.1 Software-Development Life-Cycle	29
2.2 Waterfall Model	36
2.3 Prototyping Model	41
2.4 Spiral Model	44
2.5 Evolutionary Development Model	46
2.6 Iterative-Enhancement Model	47
2.7 RAD Model	49
2.8 Comparison of Various Process Models	50

Chapter 3. Introduction to Software Requirements Specification	53
3.1 Requirement Engineering	53
3.2 Process of Requirements Engineering	55
3.3 Information Modeling	61
3.4 Data-Flow Diagrams	62
3.5 Decision Tables	67
3.6 SRS Document	70
3.7 IEEE Standards for SRS Documents	73
3.8 SRS Validation	75
3.9 Components of SRS	75
3.10 Characteristics of SRS	78
3.11 Entity-Relationship Diagram	79
Chapter 4. Software Reliability and Quality Assurance	85
4.1 Verification and Validation	85
4.2 Software Quality Assurance	87
4.3 Software Quality	89
4.4 Capability Maturity Model (SEI-CMM)	96
4.5 International Standard Organization (ISO)	99
4.6 Comparison of ISO-9000 Certification and the SEI-CMM	106
4.7 Reliability Issues	107
4.8 Reliability Metrics	110
4.9 Reliability Growth Modeling	112
4.10 Reliability Assessment	115
Chapter 5. System Design	117
5.1 System/Software Design	117
5.2 Architectural Design	123
5.3 Low-Level Design	125
5.4 Coupling and Cohesion	136
5.5 Functional-Oriented Versus The Object-Oriented Approach	143
5.6 Design Specifications	144
5.7 Verification for Design	145
5.8 Monitoring and Control for Design	146

Chapter 6. Software Measurement and Metrics	149
6.1 Software Metrics	149
6.2 Halstead's Software Science	151
6.3 Function-Point Based Measures	154
6.4 Cyclomatic Complexity	157
Chapter 7. Software Testing	161
7.1 Introduction to Testing	161
7.2 Testing Principles	162
7.3 Testing Objectives	163
7.4 Test Oracles	164
7.5 Levels of Testing	165
7.6 White-Box Testing/Structural Testing	173
7.7 Functional/Black-Box Testing	175
7.8 Test Plan	178
7.9 Test-Case Design	179
Chapter 8. Software-Testing Strategies	181
8.1 Static-Testing Strategies	181
8.2 Debugging	186
8.3 Error, Fault, and Failure	189
Chapter 9. Software Maintenance and Project Management	193
9.1 Software as an Evolution Entity	193
9.2 Software-Configuration Management Activities	193
9.3 Change-Control Process	197
9.4 Software-Version Control	199
9.5 Software-Configuration Management	200
9.6 Need for Maintenance	202
9.7 Categories of Maintenance	203
9.8 Maintenance Costs	204
9.9 Software-Project Estimation	207
9.10 Constructive Cost Model (COCOMO)	211
9.11 Software-Risk Analysis and Management	215

Chapter 10. Computer-Aided Software Engineering	223
10.1 Case and its Scope	223
10.2 Levels of Case	224
10.3 Architecture of Case Environment	224
10.4 Building Blocks for Case	226
10.5 Case Support in Software Life-Cycle	227
10.6 Objectives of Case	228
10.7 Case Repository	229
10.8 Characteristics of Case Tools	231
10.9 Case Classification	231
10.10 Categories of Case Tools	233
10.11 Advantages of Case Tools	234
10.12 Disadvantages of Case Tools	235
10.13 Reverse Software Engineering	235
10.14 Software Re-Engineering	240
Chapter 11. Coding	247
11.1 Information Hiding	247
11.2 Programming Style	248
11.3 Internal Documentation	250
11.4 Monitoring and Control for Coding	251
11.5 Structured Programming	252
11.6 Fourth-Generation Techniques	255
PART II: SOFTWARE DEVELOPMENT AND APPLICATIONS	
Chapter 12. Introduction to Software Development	261
12.1 Program Phase	262
12.2 How to Write a Good Program	262
12.3 Programming Tools	263
Chapter 13. Visual Basic 6.0	265
13.1 Hardware and Software Requirements for Visual Basic	266
13.2 Application Types	266
13.3 Compilation in Visual Basic	268

13.4 Visual Basic Terminology	268
13.5 Integrated Development Environment (IDE)	269
Chapter 14. Controls in Visual Basic	273
14.1 Tool-Box Controls	276
Chapter 15. Variables and Operators in Visual Basic	297
15.1 Variable Naming Conventions	297
15.2 Variable Declaration	297
15.3 Scope of Variables	298
15.4 Logical Operators	298
15.5 Logical Operators	299
15.6 If-Else Statement	301
15.7 Do While Statement	301
15.8 For Loop	302
15.9 With-End With Statement	302
Chapter 16. Functions in Visual Basic	303
Chapter 17. Introduction to Databases	315
17.1 Tables	316
17.2 Structure of a Database	317
17.3 Keys	317
17.4 Data Integrity	318
Chapter 18. MS Access 2000	319
18.1 Creating a Database in MS Access 2000	319
18.2 Data Types	324
18.3 Field Properties	324
18.4 Saving the Table	327
18.5 Modifying the Table	328
18.6 Importing the Table	328
Chapter 19. Oracle	329
19.1 Starting with Oracle 8	329
19.2 How to Create a New User	331

19.3 User Creation by Navigator	332
19.4 Data Types in Oracle	335
19.5 Syntax and Query in Oracle	336
19.6 Functions	344
19.7 Primary Keys	345
19.8 Data Export	346
19.9 Data Import	347
Chapter 20. SQL Server 2000	349
20.1 What's New in Microsoft SQL Server 2000?	349
20.2 Starting Microsoft SQL Server 2000	349
20.3 Installation of SQL Server 2000	351
20.4 Creating a Database	353
20.5 How to Create a Database Using Enterprise Manager	354
20.6 Create a Database Using the Create Database Wizard in Enterprise Manager	358
20.7 Creating a New Table	358
20.8 Data Types	361
20.9 Query Analyzer	368
20.10 How to use Query Analyzer	368
20.11 Generating an SQL Script	370
20.12 How to use the Script	374
20.13 Attaching a Database	376
20.14 Detaching a Database	378
20.15 Copy Database Wizard	380
20.16 Importing and Exporting a Database	380
20.17 SQL Server Service Manager	386
Chapter 21. Programming in Visual Basic with MS Access 2000	391
21.1 Saving Projects and Forms	393
21.2 Database Designing	399
21.3 Use of App.Path	412
Chapter 22. Programming with Oracle and SQL Server 2000	413
22.1 Table Creation	413

22.2 Data Links	414
22.3 Working with the Project	418
22.4 Data Export at Runtime	420
22.5 Working in a Project with an SQL Server 2000 Database	420
Chapter 23. Graphs	421
Chapter 24. Data Reports	425
24.1 Data Report Creation	425
24.2 Data Environment and the Connection	425
24.3 Data Report Designing	430
24.4 Data Report Controls	433
24.5 Calling a Report	436
24.6 Retrieval of Selected Data in the Data Report	436
24.7 Index Number of Data Report Section	439
24.8 Grouping in Data Reports	440
Chapter 25. Crystal Reports	447
25.1 Advantages over Visual Basic Data Reports	448
25.2 Starting with Crystal Report 8.5	448
25.3 Creating Reports Using DSN of the SQL Server 2000 Database	451
25.4 Creating Connection Using DSN	456
Chapter 26. Error Handling	465
26.1 Key Handling	465
26.2 Key Locking at Key Press Event	469
26.3 Other Error-Handling Methods	470
26.4 Some Common Errors	471
26.5 Precautions	479
Chapter 27. Creating the Setup Package	481
27.1 How to Create a Setup	481
Index	493

PART I

SOFTWARE ENGINEERING AND TESTING

The role of software engineering cannot be neglected in the field of software development. The advent of computers introduced the need for software and the quality of software introduced the need for software engineering. Software engineering has come a long way since 1968, when the term was first used at a NATO conference, and software itself has entered our lives in ways that few had anticipated, even a decade ago. So a firm grounding in software-engineering theory and practice is essential for understanding how to build good error-free software at an inexpensive price and with less time and for evaluating the risks and opportunities that software presents in our everyday lives.

Chapter 1

INTRODUCTION TO SOFTWARE ENGINEERING

1.1 INTRODUCTION TO SOFTWARE

Software is described by its capabilities. The capabilities relate to the functions it executes, the features it provides, and the facilities it offers. Software written for sales-order processing would have different functions to process different types of sales orders from different market segments. The features, for example, would be to handle multi-currency computing, updating of product, sales, and tax status in MIS reports and books of accounts. The facilities could be printing of sales orders, e-mails to customers, reports, and advice to the store departments to dispatch the goods. The facilities and features could be optional and based on customer choices.

Software is developed keeping in mind certain hardware and operating system considerations, known as the platform. Hence, software is described along with its capabilities and the platform specifications that are required to run it.

1.1.1 Definition of Software

Software is a set of instructions used to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software. It also includes a set of documents, such as the software manual, meant to help users understand the software system. Today's software is comprised of Source Code, Executables, Design Documents, Operations, and System Manuals and Installation and Implementation Manuals.

Software includes:

- (i) Instructions (computer programs) that when executed provide desired functions and performance.
- (ii) Data structures that enable the programs to adequately manipulate information.
- (iii) Documents that describe the operation and use of the programs.

OR

The term software refers to the set of computer programs, procedures, and associated documents (flowcharts, manuals, etc.) that describe the programs and how they are to be used. To be precise, software is a collection of programs whose objective is to enhance the capabilities of the hardware.

OR

Definition of software given by the IEEE:

Software is the collection of computer programs, procedure rules and associated documentation and data.

1.1.2 Importance of Software

Computer software has become a driving force.

- It is the engine that drives business decisionmaking.
- It serves as the basis for modern scientific investigation and engineering problem-solving.
- It is embedded in all kinds of systems, such as transportation, medical, telecommunications, military, industrial processes, entertainment, office products, etc.

It is important as it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities. Software's impact on our society and culture is significant. As software importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs.

1.2 TYPES OF SOFTWARE

Computer software is often divided into two categories:

1. **System software.** This software includes the operating system and all utilities that enable the computer to function.
2. **Application software.** These consist of programs that do real work for users. For example, word processors, spreadsheets, and database management systems fall under the category of applications software.

System software are low-level programs that interact with the computer at a very basic level and include the operating system and utilities for managing resources. In contrast, application software includes database programs, word processors, and spreadsheets. Application software sits above system software because it needs the help of system software to run. Figure 1.1 gives an overview of software classification and its types.

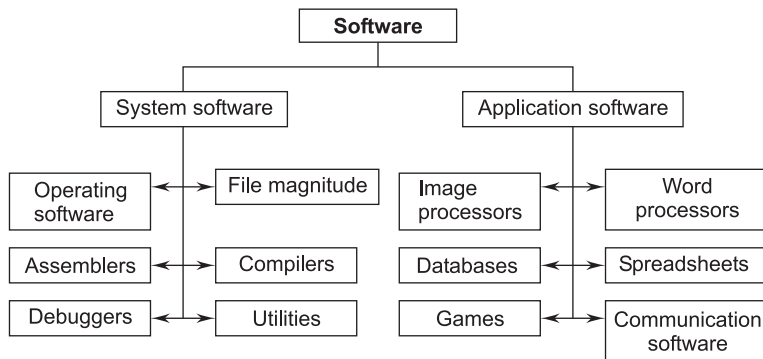


FIGURE 1.1 Types of Software

The following is an overview of the different classes of software:

1. **Operating Systems.** The operating system provides interface between the user and the hardware. It manages hardware, such as memory, CPU, input/output devices, files, etc., for the user. Most commonly used operating systems include Microsoft Windows, DOS, XENIX, Mac OS, OS/2, Unix MVS, etc.
2. **Utilities.** Utilities are programs that perform the specification tasks related to managing system resources. The operating system includes a number of utilities for managing disk printers and other devices.
3. **Compilers and Interpreters.** A compiler is a program that translates source code into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and

reorganizing the instructions. Thus, a compiler differs from an interpreter, which analyzes and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

4. **Word Processors.** A word processor is a program that enables you to perform word-processing functions. Word processors use a computer to create, edit, and print documents. Of all computer applications, word processors are the most common.

To perform word processing, you need a computer, the word-processing software (word processor), and a printer. A word processor enables you to create a document, store it electronically on a disk, display it on a screen, modify it by entering commands and characters from the keyboard, and print it on a printer.

The great advantage of word processing over using a typewriter is that you can make changes without retyping the entire document. If you make a typing mistake, you simply back up the cursor and correct your mistake. If you want to delete a paragraph, you simply remove it, without leaving a trace. It is equally easy to insert a word, sentence, or paragraph in the middle of a document. Word processors also make it easy to move sections of text from one place to another within a document, or between documents. When you have made all the changes you want, you can send the file to a printer to get a hardcopy. Some of the commonly used word processors are Microsoft Word, WordStar, WordPerfect, AmiPro, etc.

5. **Spreadsheets.** A spreadsheet is a table of values arranged in rows and columns. Each value can have a predefined relationship to the other values. If you change one value, therefore, you may need to change other values as well.

Spreadsheet applications (often referred to simply as spreadsheets) are computer programs that let you create and manipulate spreadsheets electronically. In a spreadsheet application, each value sits in a cell. You can define what type of data is in each cell and how different cells depend on one another. The relationships between cells are called formulas, and the names of the cells are called labels.

Once you have defined the cells and the formulas for linking them together, you can enter your data. You can then modify selected values to see how all the other values change accordingly. This enables you to study various what-if scenarios.

There are a number of spreadsheet applications on the market, Lotus 1-2-3 and Excel being among the most famous. These applications support graphic features that enable you to produce charts and graphs from the data.

Some spreadsheets are multi-dimensional, meaning that you can link one spreadsheet to another. A three-dimensional spreadsheet, for example, is like a stack of spreadsheets all connected by formulae. A change made in one spreadsheet automatically affects other spreadsheets.

6. **Presentation Graphics.** Presentation graphics enable users to create highly stylized images for slide shows and reports. The software includes functions for creating various types of charts and graphs and for inserting text in a variety of fonts. Most systems enable you to import data from a spreadsheet application to create the charts and graphs. Presentation graphics are often called business graphics. Some of the popular presentation graphics software include Microsoft PowerPoint, Lotus Freelance Graphics, Harvard Presentation Graphics, etc.
7. **Database Management System (DBMS).** A DBMS is a collection of programs that enable you to store, modify, and extract information from a database. There are many different types of DBMSs, ranging from small systems that run on personal computers to huge systems that run on mainframes. The following are some examples of database applications: computerized library systems, automated teller machines, flight and railway reservation systems, computerized inventory systems, etc.

From a technical standpoint, a DBMS can differ widely. The terms relational, network, flat, and hierarchical all refer to the way a DBMS organizes information internally. The internal organization can affect how quickly and flexibly you can extract information.

Requests for information from a database are made in the form of a query, which is a stylized question. Different DBMSs support different query languages, although there is a semi-standardized query language called SQL (structured query language). Sophisticated languages for managing database systems are called fourth-generation languages, or 4GLs for short.

The information from a database can be presented in a variety of formats. Most DBMSs include a report-writer program that enables you to output data in the form of a report. Many DBMSs also include a graphics component that enables you to output information in the form of graphs and charts. Some examples of database management systems are IDMS, IMS, DB2, Oracle, Sybase, Informix, Ingress, MS-SQL Server, MS Access, etc.

8. **Image Processors.** Image processors or graphics programs enable you to create, edit, manipulate, add special effects, view, and print and save images, and include the following:

(i) *Paint Programs.* A paint program is a graphics program that enables you to draw pictures on the display screen, which is represented as bitmaps (bit-mapped graphics). Most paint programs provide the tools in the form of icons. By selecting an icon, you can perform functions associated with the tool. In addition to these tools, paint programs also provide easy ways to draw common shapes, such as straight lines, rectangles, circles, and ovals.

Sophisticated paint applications are often called image-editing programs. These applications support many of the features of draw programs, such as the ability to work with objects. Each object, however, is represented as a bitmap rather than as a vector image.

(ii) *Draw Programs.* A draw program is another graphics program that enables you to draw pictures, then store the images in files, merge them into documents, and print them. Unlike paint programs, which represent images as bitmaps, draw programs use vector graphics, which makes it easy to scale images to different sizes. In addition, graphics produced with a draw program have no inherent resolution. Rather, they can be represented at any resolution, which makes them ideal for high-resolution output.

(iii) *Image Editors.* An image editor is a graphics program that provides a variety of special features for altering bit-mapped images. The difference between image editors and paint programs is not always clear-cut, but in general, image editors are specialized for modifying bit-mapped images, such as scanned photographs, whereas paint programs are specialized for creating images. In addition to offering a host of filters and image transformation algorithms, image editors also enable you to create and superimpose layers.

1.3 CLASSES OF SOFTWARE

Software is classified into the following two classes:

1. **Generic Software.** Generic software is designed for a broad customer market whose requirements are very common, fairly stable, and well-understood by the software engineer.

These products are sold in the open market, and there could be several competitive products on the market. Database products, browsers, ERP/CRM

and CAD/CAM packages, OS and system software are examples of generic software.

2. **Customized Software.** Customized products are those that are developed for a customer where domain, environment, and requirements are unique to that customer and cannot be satisfied by generic products.

Legacy systems, software written for specific business processes that are typical of the specific industry, are used when a customized software product is needed. Process-control systems, traffic-management systems, hospital-management systems, and manufacturing-process control systems require customized software.

The developer manages a generic product and the customer manages a customized product. In other words, requirements and specifications of a generic product are controlled by the developer, whereas in the case of a customized product, these are controlled by the customer and influenced by the practices of that industry.

1.4 INTRODUCTION TO SOFTWARE ENGINEERING

A few important definitions given by several authors and institutions are as follows:

IEEE Comprehensive Definition. *Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, i.e., the application of engineering to software.*

Other Definitions. *Software Engineering deals with cost-effective solutions to practical problems by applying scientific knowledge in building software artifacts in the service of mankind.*

OR

Software Engineering is the application of methods and scientific knowledge to create practical cost-effective solutions for the design, construction, operation and maintenance of software.

OR

Software Engineering is a discipline whose aim is the production of fault free software that satisfies the user's needs and that is delivered on time and within budget.

OR

The term Software Engineering refers to a movement, methods and techniques aimed at making software development more systematic. Software methodologies, such as, OMG's UML and software tools (CASE tools) that help developers model application designs and then generate code are all closely associated with Software Engineering.

OR

Software Engineering is an engineering discipline which is concerned with all aspects of software production.

1.4.1 Software-Engineering Principles

Software-engineering principles deal with both the process of software engineering and the final product. The right process will help produce the right product, but the desired product will also affect the choice of which process to use. A traditional problem in software engineering has been the emphasis on either the process or the product to the exclusion of the other. Both are important.

The principles we develop are general enough to be applicable throughout the process of software construction and management. Principles, however, are not sufficient to drive software development. In fact, they are general and abstract statements describing desirable properties of software processes and products. But, to apply principles, the software engineer should be equipped with appropriate methods and specific techniques that help incorporate the desired properties into processes and products.

In principle, we should distinguish between methods and techniques. Methods are general guidelines that govern the execution of some activity; they are rigorous, systematic, and disciplined approaches. Techniques are more technical and mechanical than methods; often, they also have more restricted applicability. In general, however, the difference between the two is not sharp. We will therefore use the two terms interchangeably.

Sometimes, methods and techniques are packaged together to form a methodology. The purpose of a methodology is to promote a certain approach to solving a problem by preselecting the methods and techniques to be used. Tools, in turn, are developed to support the application of techniques, methods, and methodologies.

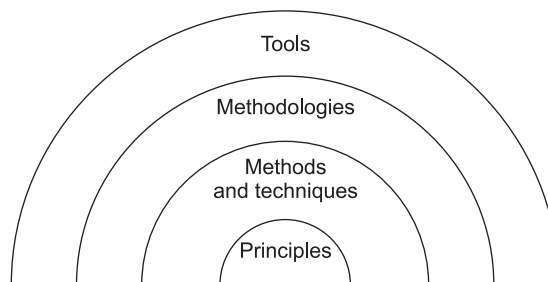


FIGURE 1.2 Relationship Between Principles, Techniques, Methodologies, and Tools

Figure 1.2 shows the relationship between principles, methods, methodologies, and tools. Each layer in the figure is based on the layer(s) below it and is more susceptible to change, due to the passage of time. This figure shows clearly that principles are the basis of all methods, techniques, methodologies, and tools.

1.5 SOFTWARE COMPONENTS

A software component is a system element offering a predefined service and is able to communicate with other components. Clemens Szyperski and David Messerschmitt give the following five criteria for what a software component shall be to fulfill the definition:

- Multiple-use
- Non-context-specific
- Composable with other components
- Encapsulated, i.e., non-investigable through its interfaces
- A unit of independent deployment and versioning

A simpler definition can be: A component is an object written to a specification. It does not matter what the specification is, COM, Java Beans, etc., as long as the object adheres to the specification. It is only by adhering to the specification that the object becomes a component and gains features, such as reusability and so forth.

Software components often take the form of objects or collections of objects (from object-oriented programming) in some binary or textual form, adhering to some interface description language (IDL) so that the component may exist autonomously from other components on a computer.

When a component is to be accessed or shared across execution contexts or network links, some form of serialization (also known as marshalling) is employed to turn the component or one of its interfaces into a bitstream.

It takes significant effort and awareness to write a software component that is effectively reusable. The component needs:

- to be fully documented;
- to be more thoroughly tested;
- to have robust input validity checking;
- to pass back useful error messages as appropriate;
- to be built with an awareness that it will be put to unforeseen uses;
- a mechanism for compensating developers who invest the (substantial) effort implied above.

1.6 SOFTWARE CHARACTERISTICS

The key characteristics of software are as follows:

1. **Most software is custom-built, rather than assembled from existing components.** Most software continues to be custom built, although recent developments tend to be component-based. Modern reusable components encapsulate both data and the processing applied to data, enabling the software engineer to create new applications from reusable parts. For example, today a GUI is built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing details required to build the interface are contained in a library of reusable components for interface construction.
2. **Software is developed or engineered; it is not manufactured in the classical sense.** Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software. Both activities depend on people, but the relationship between people applied and work accomplished is entirely different. Both require the construction of a “product.” But the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.
3. **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. Reuse of components from libraries help in reduction of effort. Now-a-days, we reuse not only algorithms but also data structures.

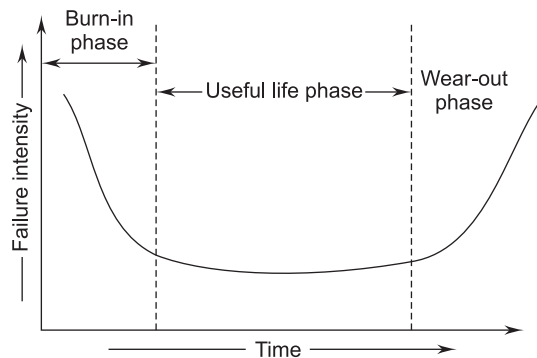


FIGURE 1.3 Bath-tub Curve

4. **Software doesn't wear out.** There is a well-known "bath-tub curve" in reliability studies for hardware products. Figure 1.3 depicts the failure rate as a function of time for hardware. The relationship is often called the "bath-tub curve."

There are three phases for the life of a hardware product. The initial phase is the burn-in phase, where failure intensity is high. The product is tested in the industry before delivery. Due to testing and fixing faults, failure intensity will come down initially and may stabilize after a certain time. The second phase is the useful life phase where failure intensity is approximately constant and is called the useful life of a product. After a few years, again failure intensity will increase due to wearing out of components. This phase is called the wear-out phase. We do not have this phase for software, as it does not wear out. The curve for software is given in Figure 1.4.

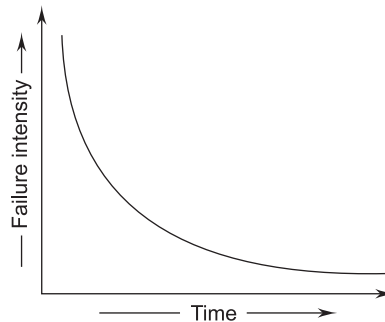


FIGURE 1.4 Software Curve

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

1.7 SOFTWARE CRISIS

The software crisis has been with us since the 1970s. 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."

During software development, then, many problems are raised and that set of problems is known as the software crisis. When software is being developed, problems are encountered associated with the development steps. Now we will

discuss the problems and causes of the software crises encountered in different stages of software development.

Problems

- Schedule and cost estimates are often grossly inaccurate.
- The “productivity” of software people hasn’t kept pace with the demand for their services.
- The quality of software is sometimes less than adequate.
- With no solid indication of productivity, we can’t accurately evaluate the efficiency of new, tools, methods, or standards.
- Communication between the customer and software developer is often poor.
- Software maintenance tasks devour the majority of all software funds.

Causes

- The quality of the software is not good because most developers use historical data to develop the software.
- If there is delay in any process or stage (i.e., analysis, design, coding & testing) then scheduling does not match with actual timing.
- Communication between managers and customers, software developers, support staff, etc., can break down because the special characteristics of software and the problems associated with its development are misunderstood.
- The software people responsible for tapping the potential often change when it is discussed and resist change when it is introduced.

Software Crisis from the Programmer’s Point-of-View

- Problem of compatibility.
- Problem of portability.
- Problem in documentation.
- Problem of piracy of software.
- Problem in coordination of work of different people.
- Problem of proper maintenance.

Software Crisis form the User’s Point-of-View

- Software cost is very high.
- Hardware goes down.
- Lack of specialization in development.

- Problem of different versions of software.
- Problem of views.
- Problem of bugs.

1.8 SOFTWARE MYTHS

The following are different myths about software:

- If we get behind schedule, we can add more programmers and catch up.
- If we decide to outsource the software project to a third party, we can just relax and let that firm build it.
- Project requirement continuously changes, but changes can be easily accommodated because software is flexible.
- The only deliverable work product for a successful project is the working program.
- Software with more features is better software.
- Once we write the program and get it to work, our job is done.
- Until we get the program running, we have no way of assessing its quality.
- Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.
- A general statement of objectives is sufficient to begin writing programs; we can fill in the details later.
- We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

1.9 SOFTWARE APPLICATIONS

Software applications are grouped into eight areas for convenience as shown in Figure 1.5.

1. **System Software.** System software is a collection of programs used to run the system as an assistance to other software programs. The compilers, editors, utilities, operating system components, drivers, and interfaces are examples of system software. This software resides in the computer system and consumes its resources. A computer system without system software cannot function.
System software directly interacts with the hardware, heavy usage by multiple users, concurrent operations that require scheduling, resource sharing, and

sophisticated process management, complex data structures, and multiple external interfaces.

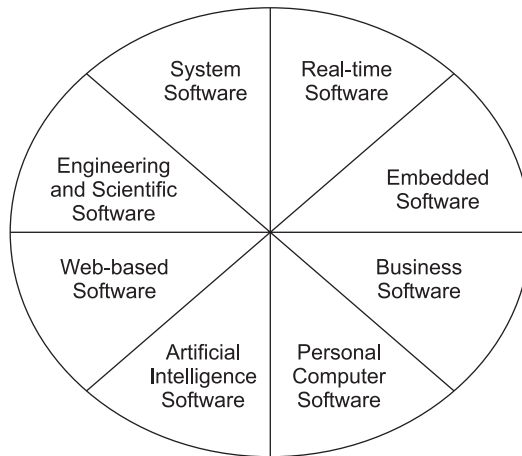


FIGURE 1.5 Software Applications

- 2. Real-time Software.** Real-time software deals with a changing environment. First, it collects the input and converts it from analog to a digital, control component that responds to the external environment and performs the action.

The software is used to monitor, control, and analyze real-world events as they occur. Examples include Rocket launching, games, etc.

- 3. Embedded Software.** Software, when written to perform certain functions under control conditions and is further embedded into hardware as a part of large systems, is called embedded software.

The software resides in the Read-Only-Memory (ROM) and is used to control the various functions of the resident products. The products could be a car, washing machine, microwave oven, industrial processing products, gas stations, satellites, and a host of other products, where the need is to acquire input, analyze, identify status, and decide and take action that allows the product to perform in a predetermined manner. Because of its role and performance, it is also called intelligent software.

- 4. Business Software.** Software designed to process business applications is called business software. Business software can be a data- and information-processing application. It can drive the business process through transaction processing in on-line or in real-time mode.

This software is used for specific operations, such as accounting packages, management information systems, payroll packages, and inventory management. Business software restructures existing data in order to facilitate business

operations or management decision-making. It also encompasses interactive computing. It is integrated software related to a particular field.

5. **Personal Computer Software.** Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external networks, or database access are only a few of hundreds of applications.
6. **Artificial-intelligence Software.** Artificial-intelligence software uses non-numerical algorithms, which use the data and information generated in the system to solve complex problems. These problem scenarios are not generally amenable to problem-solving procedures, and require specific analysis and interpretation of the problem to solve it.

Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving and game playing, and signal-processing software.

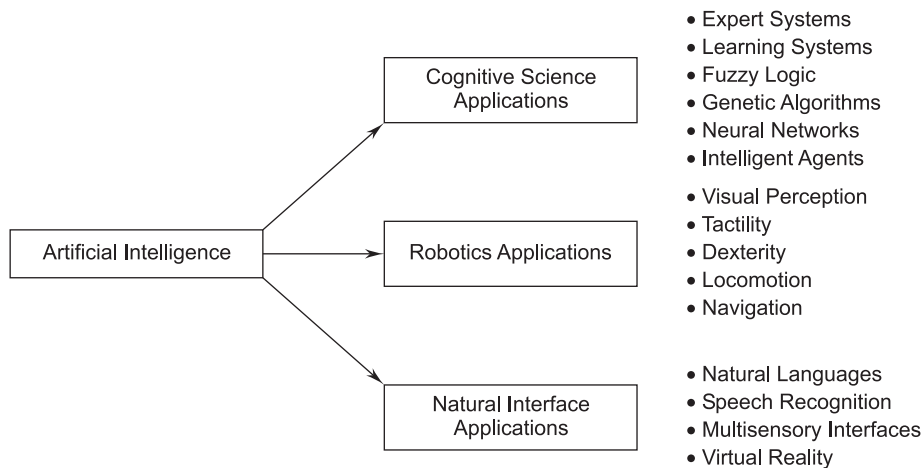


FIGURE 1.6 Application Areas of Artificial Intelligence

7. **Web-based Software.** Web-based software includes the languages by which web pages are processed, i.e., HTML, Java, CGI, Perl, DHTML, etc.
8. **Engineering and Scientific Software.** The design and engineering of scientific software deals with processing requirements in their specific fields. They are written for specific applications using the principles and formulae of each field.

These software applications service the need for drawing, drafting, modeling, lead calculations, specifications-building, and so on. Dedicated software is available for stress analysis or for analysis of engineering data, statistical data for inter-

pretation, and decision-making. CAD/CAM/CAE packages, SPSS, MATLAB, and circuit analyzers are typical examples of such software.

1.10 SOFTWARE-ENGINEERING PROCESSES

1.10.1 Process

A process is a series of steps involving activities, constraints, and resources that produce an intended output of some kind.

Any process has the following characteristics:

- The process prescribes all of the major process activities.
- The process uses resources, subject to a set of constraints (such as a schedule), and produces intermediate and final products.
- The process may be composed of sub-processes that are linked in some way. The process may be defined as a hierarchy of processes, organized so that each sub-process has its own process model.
- Each process activity has entry and exit criteria, so that we know when the activity begins and ends.
- The activities are organized in a sequence, so that it is clear when one activity is performed relative to the other activities.
- Every process has a set of guiding principles that explain the goals of each activity.
- Constraints or controls may apply to an activity, resource, or product. For example, the budget or schedule may constrain the length of time an activity may take or a tool may limit the way in which a resource may be used.

1.10.2 What is a Software Process?

A software process is the related set of activities and processes that are involved in developing and evolving a software system.

OR

A set of activities whose goal is the development or evolution of software.

OR

A software process is a set of activities and associated results, which produce a software product.

These activities are mostly carried out by software engineers. There are four fundamental process activities (covered later in the book), which are common to all software processes. These activities are:

1. **Software specifications:** The functionality of the software and constraints on its operation must be defined.
2. **Software development:** Software that meets the specifications must be produced.
3. **Software validation:** The software must be validated to ensure that it does what the customer wants.
4. **Software evolution:** The software must evolve to meet changing customer needs.

Different software processes organize these activities in different ways and are described at different levels of detail. The timing of the activities varies, as does the results of each activity. Different organizations may use different processes to produce the same type of product. However, some processes are more suitable than others for some types of applications. If an inappropriate process is used, this will probably reduce the quality or the usefulness of the software product to be developed.

A software process can be characterized as shown in Figure 1.7. A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of task sets—each a collection of software-engineering work tasks, project milestones, software work products and deliverables, and quality-assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software-configuration management and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

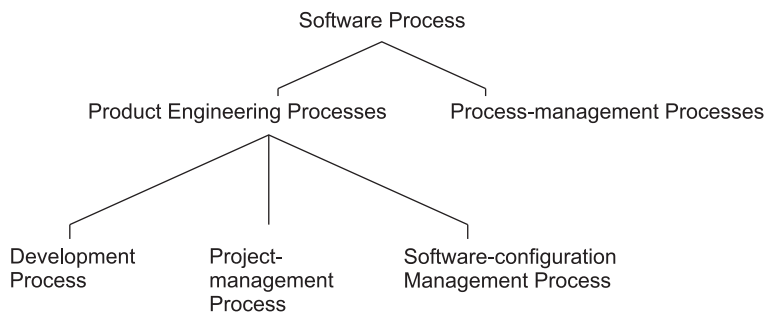


FIGURE 1.7 The Software Process

Thus, the software industry considers software development a process. According to Booch and Rumbaugh, *"A process defines who is doing what, when and how to reach a certain goal."* Software engineering is a field, which combines processes, methods, and tools for the development of software. The concept of process is the main step in the software engineering approach. Thus, a software process is a set of activities. When those activities are performed in specific sequence in accordance with ordering constraints, the desired results are produced.

1.11 EVOLUTION OF SOFTWARE

Software-engineering principles have evolved over the past more than 50 years from art to an engineering discipline. This can be shown with the help of Figure 1.8.

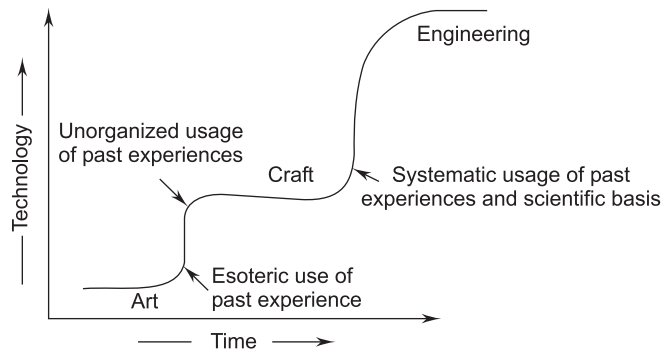


FIGURE 1.8 Evolution of Art to an Engineering Discipline

Development in the field of software and hardware computing made a significant change in the twentieth century. We can divide the software development process into four eras:

1. **Early Era.** During the early eras general-purpose hardware became commonplace. Software, on the other hand, was custom-designed for each application and was relatively limited in distribution. Most software was developed and ultimately used by the same person or organization.

This era was from 1950 to 1960 and includes:

- Limited distribution
- Custom software
- Batch orientation

2. **Second Era.** The second era in the evolution of computer systems introduced new concepts of human-machine interaction. Interactive techniques opened a new world of applications and new levels of hardware and software sophistication. Real-time software deals with the changing environment and is multi-user in which many users can perform or work with a software at the same time.

This era was from 1960 to 1972 and includes:

- Multi-users
- Databases
- Real-time software
- Product software
- Multi-programming

3. **Third Era.** In the second era software was custom designed and limited in distribution but in the third era software was consumer designed and the distribution was not limited. The cost of the hardware was also very low in this era.

This era was from 1973 to 1985 and includes:

- Embedded intelligence
- Consumer impact
- Distributed systems
- Low-cost hardware

4. **Fourth Era.** The fourth era in the evolution of computer systems moves us away from individual computers and computer programs and toward the collective impact of computers and software. As the fourth era progresses, new technologies have begun to emerge.

This era is from 1985 to present and includes:

- Powerful desktop systems
- Expert systems
- Artificial intelligence
- Network computers
- Parallel computing
- Object-oriented technology

At this time the concept of software making includes object-oriented technology, network computing, etc.

1.12 COMPARISON OF SOFTWARE ENGINEERING AND RELATED FIELDS

The relationships between software engineering and the fields of computer science and traditional engineering has been debated for decades. Software engineering resembles all of these fields, but important distinctions exist.

1.12.1 Comparing Computer Science

Many compare software engineering to computer science and information science like they compare traditional engineering to physics and chemistry.

About half of all software engineers earn computer science degrees. Yet on the job, practitioners do applied software engineering, which differs from doing theoretical computer science.

TABLE 1.1

Issue	Software Engineering	Computer Science
Ideal	Constructing software applications for real-world use for today	Finding eternal truths about problems and algorithms for posterity
Results	Working applications (such as office suites and video games) that deliver value to users	Computational complexity and correctness of algorithms (such as Shell sort) and analysis of problems (such as the traveling salesman problem)
Budgets and Schedules	Projects (such as upgrading an office suite) have fixed budgets and schedules	Projects (such as solving $P = NP?$) have open-ended budgets and schedules
Change	Applications evolve as user needs and expectations evolve, and as SE technologies and practices evolve	When computer science problems are solved, the solution will never change
Additional Skills	Domain knowledge	Mathematics
Notable Educators and Researchers	Barry Boehm, Fred Brooks, and David Parnas	Edsger Dijkstra, Donald Knuth, and Alan Turing

Notable Practitioners	Dan Bricklin, Steve McConnell	Not applicable
Practitioners in U.S.	680,000	25,000
Practitioners in Rest of World	1,400,000?	50,000?

1.12.2 Comparing Engineering

The software-engineering community is about 60% as large as the rest of the engineering community combined.

Software engineers aspire to build low-cost, reliable, safe products; much like engineers in other disciplines do. Software engineers borrow many metaphors and techniques from other engineering disciplines, including requirements analysis, quality control, and project-management techniques. Engineers in other disciplines also borrow many tools and practices from software engineers. Yet, there are also some differences between software engineering and other engineering disciplines.

TABLE 1.2

Issue	Software Engineering	Engineering
Foundations	Based on computer science, information science, and discrete math.	Based on science, mathematics, and empirical knowledge.
Cost	Compilers and computers are cheap, so software engineering and consulting are often more than half of the cost of a project. Minor software engineering cost-overruns can adversely affect the total project cost.	In some projects, construction and manufacturing costs can be high, so engineering may only be 15% of the cost of a project. Major engineering cost overruns may not affect the total project cost.
Replication	Replication (copying CDs or downloading files) is trivial. Most development effort goes into building new (unproven) systems or changing old designs and adding features.	Radically new or one-of-a-kind systems can require significant development effort to create a new design or change an existing design. Other kinds of systems may require less development effort, but more attention to issues such as manufacturability.

Innovation	Software engineers often apply new and untested elements in software projects.	Engineers generally try to apply known and tested principles, and limit the use of untested innovations to only those necessary to create a product that meets its requirements.
Duration	Software engineers emphasize projects that will live for years or decades.	Some engineers solve long-range problems (bridges and dams) that endure for centuries.
Management Status	Few software engineers manage anyone.	Engineers in some disciplines, such as civil engineering, manage construction, manufacturing, or maintenance crews.
Blame	Software engineers must blame themselves for project problems.	Engineers in some fields can often blame construction, manufacturing, or maintenance crews for project problems.
Practitioners in U.S.	611,900 software engineers	1,157,020 total non-software engineers
Age	Software engineering is about 50 years old.	Engineering as a whole is thousands of years old.
Title Regulations	Software engineers are typically self-appointed. A computer-science degree is common but not at all a formal requirement.	In many jurisdictions it is illegal to call yourself an engineer without specific formal education and/or accreditation by governmental or engineering association bodies.
Analysis Methodology	Methods for formally verifying correctness are developed in computer science, but they are rarely used by software engineers. The issue remains controversial.	Some engineering disciplines are based on a closed system theory and can in theory prove formal correctness of a design. In practice, a lack of computing power or input data can make such proofs of correctness intractable, leading many engineers to use a pragmatic mix of analytical approximations and empirical test data to ensure that a product will meet its requirements.

Synthesis Methodology	SEs struggle to synthesize (build to order) a result according to requirements.	Engineers have nominally refined synthesis techniques over the ages to provide exactly this. However, this has not prevented some notable engineering failures, such as the collapse of the Tacoma Narrows Bridge, the sinking of the Titanic, and the Pentium FDIV bug. In addition, new technologies inevitably result in new challenges that cannot be met using existing techniques.
Research during Projects	Software engineering is often busy with researching the unknown (e.g., to derive an algorithm) right in the middle of a project.	Traditional engineering nominally separates these activities. A project is supposed to apply research results in known or new clever ways to build the desired result. However, ground-breaking engineering projects, such as Project Apollo often include a lot of research into the unknown.
Codified	Software engineering has just recently started to codify and teach best practices in the form of design patterns.	Some engineering disciplines have thousands of years of best practice experience handed over from generation to generation via a field's literature, standards, rules, and regulations. Newer disciplines, such as electronic engineering and computer engineering have codified their own best practices as they have developed.

1.13 SOME TERMINOLOGIES

The following are some of the terminologies frequently used in the field of software engineering:

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. Finalization of specifications is a milestone. Completion of design documentation is another milestone. The milestones are essential for project planning and management.

2. **Product and Process.** What is delivered to the customer is called the product. It may include source-code specification documents, manuals, documentation, etc. Basically, it is nothing but a set of deliverables only.

A 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.

3. **Measures, Metrics, and Indicators.** In software engineering measures provide a quantitative indication of amount, dimension, capacity, or size of a given attribute of a product.

Metrics are a quantitative measure of the degree to which a system, component, or process possesses a given attribute of a product. An indicator is a combination of metrics.

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).

1.14 PROGRAMS VERSUS SOFTWARE PRODUCTS

1.14.1 Programs

A program is a subset of software and it becomes software only if documentation and operating procedure manuals are prepared. A program is a combination of source code and object code.

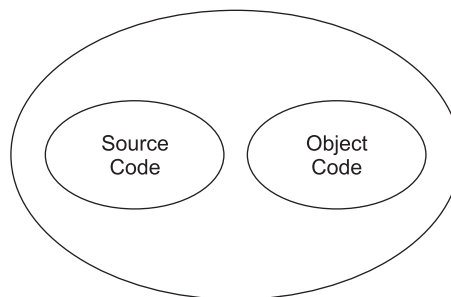


FIGURE 1.9 Program = Source Code + Object Code

1.14.2 Software Products

A software product consists not only of the program code but also of all the associated documents, such as the requirements specification documents, the design documents, the test documents, and the operating procedures which include user manuals and operational manuals.

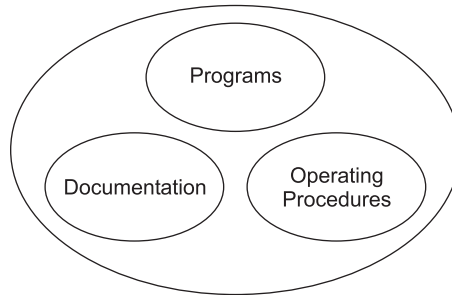


FIGURE 1.10 Software = Program + Documentation + Operating Procedures

1.14.3 Programs Versus Software Products

The various differences between a program product and a software product are given in Table 1.3.

TABLE 1.3

S. No.	Programs	Software Products
1.	Programs are developed by individuals for their personal use	A software product is usually developed by a group of engineers working as a team
2.	Usually small in size	Usually large in size
3.	Single user	Large number of users
4.	Single developer	Team of developers
5.	Lack proper documentation	Good documentation support
6.	Adhoc development	Systematic development
7.	Lack of user interface	Good user interface
8.	Have limited functionality	Exhibit more functionality

EXERCISES

1. Define software.
2. What is software engineering?
3. Describe the evolving role of software.
4. What are the different myths and realities about software?
5. Give the various application areas of software.
6. What is the bath-tub curve?
7. Discuss the characteristics of software.
8. What characteristics of software make it different from other engineering products (for example, hardware)?
9. Explain some characteristics of software. Also discuss some software components.
10. Explain the statement “software does not wear out”.
11. Discuss the evolution of software engineering in the last 50 years.
12. What are the different software components?
13. What are the symptoms of the present software crisis? What factors have contributed to the present software crisis? What are possible solutions to the present software crisis?
14. What can you learn from a software crisis?
15. What is a software crisis? Explain the problems of a software crisis.
16. What are software myths?
17. Explain in detail the software-engineering process.
18. Distinguish between a program and a software product.
19. Discuss the two well-known principles in software engineering used to tackle the complexity of the development of large programs.
20. What is the difference between software engineering and conventional engineering?

Chapter 2

SOFTWARE-DEVELOPMENT LIFE-CYCLE MODELS

2.1 SOFTWARE-DEVELOPMENT LIFE-CYCLE

The software-development life-cycle is used to facilitate the development of a large software product in a systematic, well-defined, and cost-effective way.

An information system goes through a series of phases from conception to implementation. This process is called the Software-Development Life-Cycle. Various reasons for using a life-cycle model include:

- Helps to understand the entire process
- Enforces a structured approach to development
- Enables planning of resources in advance
- Enables subsequent controls of them
- Aids management to track progress of the system

The software development life-cycle consists of several phases and these phases need to be identified along with defining the entry and exit criteria for every phase. A phase can begin only when the corresponding phase-entry criteria are

satisfied. Similarly, a phase can be considered to be complete only when the corresponding exit criteria are satisfied. If there is no clear indication of the entry and exit for every phase, it becomes very difficult to track the progress of the project.

The software development life-cycle can be divided into 5-9 phases, i.e., it must have a minimum of five phases and a maximum of nine phases. On average it has seven or eight phases. These are:

- Project initiation and planning/Recognition of need/Preliminary investigation
- Project identification and selection/Feasibility study
- Project analysis
- System design
- Coding
- Testing
- Implementation
- Maintenance

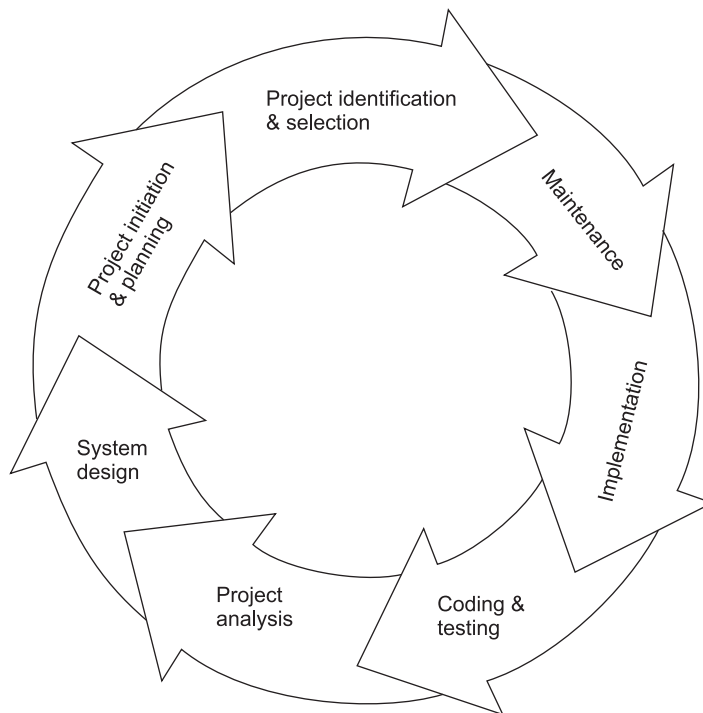


FIGURE 2.1 Software-Development Life-Cycle

1. **Recognition of Need.** Recognition of need is nothing but the problem definition. It is the decision about problems in the existing system and the impetus for system change. The first stage of any project or system-development life-cycle is called the preliminary investigation. It is a brief investigation of the system under consideration. This investigation provides the organization's computer steering committee and any project team a set of terms or references for more detailed work. This is carried out by a senior manager and will result in a study proposal. At this stage the need for changes in the existing system are identified and shortcomings of the existing system are detected. These are stated clearly providing the basis for the initial or feasibility study.
2. **Feasibility Study.** A feasibility study is a preliminary study which investigates the information needs of prospective users and determines the resource requirements, costs, benefits, and feasibility of a proposed project.

The goal of feasibility studies is to evaluate alternative systems and to propose the most feasible and desirable systems for development. The feasibility of a proposed system can be evaluated in terms of four major categories, as illustrated in Table 2.1.

- (i) *Organizational Feasibility.* Organizational feasibility is how well a proposed information system supports the objectives of the organization and is a strategic plan for an information system. For example, projects that do not directly contribute to meeting an organization's strategic objectives are typically not funded.
- (ii) *Economic Feasibility.* Economic feasibility is concerned with whether expected cost savings, increased revenue, increased profits, reductions in required investments, and other types of benefits will exceed the costs of developing and operating a proposed system. For example, if a project can't cover its development costs, it won't be approved, unless mandated by government regulations or other considerations.
- (iii) *Technical Feasibility.* Technical feasibility can be demonstrated if reliable hardware and software capable of meeting the needs of a proposed system can be acquired or developed by the business in the required time.
- (iv) *Operational Feasibility.* Operational feasibility is the willingness and ability of management, employees, customers, suppliers, and others to operate, use, and support a proposed system. For example, if the software for a new system is too difficult to use, employees may make too many errors and avoid using it. Thus, it would fail to show operational feasibility.

TABLE 2.1 Key Features of Categories of Feasibility

Organizational Feasibility	Economic Feasibility
How well the proposed system supports the strategic objectives of the organization	Cost savings Increased revenue Decreased investment Increased profits
Technical Feasibility	Operational Feasibility
Hardware, software, and network capability, reliability and availability	End-user acceptance Management support Customer, supplier, and government requirements

3. **Project Analysis.** Project analysis is a detailed study of the various operations performed by a system and their relationships within and outside the system. Detailed investigation should be conducted with personnel closely involved with the area under investigation, according to the precise terms of reference arising out of the initial study reports. The tasks to be carried out should be clearly defined such as:

- Examine and document the relevant aspects of the existing system, its shortcomings and problems.
- Analyze the findings and record the results.
- Define and document in an outline the proposed system.
- Test the proposed design against the known facts.
- Produce a detailed report to support the proposals.
- Estimate the resources required to design and implement the system.

The objectives at this stage are to provide solutions to stated problems, usually in the form of specifications to meet the users requirements and to make recommendations for a new computer-based system. Analysis is an iterative and progressive process, examining information flows and evaluating various alternative design solutions until a preferred solution is available. This is documented as the system proposal.

4. **System Design.** System design is the most creative and challenging phase of the system-development life-cycle. The term design describes the final system and process by which it is developed. Different stages of the design phase are shown in Figure 2.2. This phase is a very important phase of the life-cycle. This is a creative as well as a technical activity including the following tasks:

- Appraising the terms of reference

- Appraising the analysis of the existing system, particularly problem areas
- Defining precisely the required system output
- Determining data required to produce the output
- Deciding the medium and opening the files
- Devising processing methods and using software to handle files and to produce output
- Determining methods of data capture and data input
- Designing the output forms
- Defining detailed critical procedures
- Calculating timings of processing and data movements
- Documenting all aspects of design

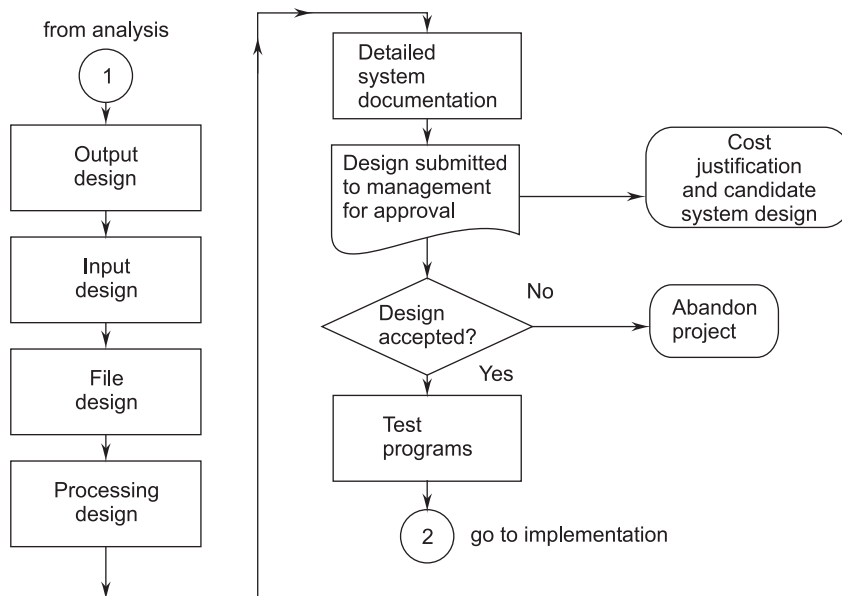


FIGURE 2.2 Cycle of Design Phase

5. **Coding.** The goal of the coding phase is to translate the design of the system into code in a given programming language. In this phase the aim is to implement the design in the best possible manner. This phase affects both testing and maintenance phases. Well-written code can reduce the testing and maintenance effort. Hence, during coding the focus is on developing programs that are easy to read and understand and not simply on developing programs that are simple to write.

Coding can be subject to company-wide standards that may define the entire layout of programs, such as headers for comments in every unit, naming conventions for variables, classes and functions, the maximum number of lines in each component, and other aspects of standardization.

Structured programming helps the understandability of programs. The goal of structured programming is to linearize the control flow in the program. Single entry-single exit constructs should be used. The constructs include selection (if-then-else) and iteration (while, repeat-unit).

6. **Testing.** Testing is the major quality-control measure used during software development. Its basic function is to detect errors in the software. Thus, the goal of testing is to uncover requirement, design, and coding errors in the program.

Testing is an extremely critical and time-consuming activity. It requires proper planning of the overall testing process. During the testing of the unit, the specified test cases are executed and the actual results are compared with the expected output. The final output of the testing phase is the test report and the error report, or a set of such reports (one for each unit tested). Each test report contains the set of test cases and the result of executing the code with these test cases. The error report describes the errors encountered and the action taken to remove the errors.

Testing cannot show the absence of defects; it can show only software errors present. During the testing phase emphasis should be on the following:

- Tests should be planned long before testing begins.
 - All tests should be traceable to customer requirements.
 - Tracing should begin “in the small” and progress toward testing “in the large.”
 - For most effective testing, independent, third parties should conduct testing.
7. **Implementation.** The implementation phase is less creative than system design. It is mainly concerned with user training, site selection, and preparation and file conversion. Once the system has been designed, it is ready for implementation. Implementation is concerned with those tasks leading immediately to a fully operational system. It involves programmers, users, and operations management, but its planning and timing is a prime function of a systems analyst. It includes the final testing of the complete system to user satisfaction, and supervision of initial operation of the system. Implementation of the system also includes providing security to the system.

Types of Implementation

There are three types of implementation:

- Implementation of a computer system to replace a manual system.
 - Implementation of a new computer system to replace an existing one.
 - Implementation of a modified application (software) to replace an existing one using the same computer.
8. **Maintenance.** Maintenance is an important part of the SDLC. If there is any error to correct or change then it is done in the maintenance phase. Maintenance of software is also a very necessary aspect related to software development. Many times maintenance may consume more time than the time consumed in the development. Also, the cost of maintenance varies from 50% to 80% of the total development cost.

Maintenance is not as rewarding or exciting as developing the systems. It may have problems such as:

- Availability of only a few maintenance tools.
- User may not accept the cost of maintenance.
- Standards and guidelines of project may be poorly defined.
- A good test plan is lacking.
- Maintenance is viewed as a necessary evil often delegated to junior programmers.
- Most programmers view maintenance as low-level drudgery.

Types of Maintenance

Maintenance may be classified as:

- (i) *Corrective Maintenance.* Corrective maintenance means repairing processing or performance failures or making changes because of previously uncorrected problems.
- (ii) *Adaptive Maintenance.* Adaptive maintenance means changing the program function. This is done to adapt to the external environment change. For example, the current system was designed so that it calculates taxes on profits after deducting the dividend on equity shares. The government has issued orders now to include the dividend in the company profit for tax calculation. This function needs to be changed to adapt to the new system.
- (iii) *Perfective Maintenance.* Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's

additional or changing needs. For example, earlier data was sent from stores to headquarters on magnetic media but after the stores were electronically linked via leased lines, the software was enhanced to send data via leased lines.

As maintenance is very costly and very essential, efforts have been done to reduce its costs. One way to reduce the costs is through maintenance management and software modification audits. Software modification consists of program rewriting and system-level-upgrading.

- (iv) *Preventive Maintenance*. Preventive maintenance is the process by which we prevent our system from being obsolete. Preventive maintenance involves the concept of re-engineering and reverse engineering in which an old system with an old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

2.2 WATERFALL MODEL

The waterfall model is a very common software development process model. The waterfall model was popularized in the 1970s and permeates most current software-engineering textbooks and standard industrial practices. Its first appearance in the literature dates back to the late 1950s as the result of experience gained in the development of a large air-defense software system called SAGE (Semi-Automated Ground Environment).

The waterfall model is illustrated in Figure 2.3. Because of the cascade from one phase to another, this model is known as the *waterfall model* or *software life-cycle*.

As the figure shows, the process is structured as a cascade of phases, where the output of one phase constitutes the input to the next one. Each phase, in turn, is structured as a set of activities that might be executed by different people concurrently.

The phases shown in the figure are the following:

- Feasibility study
- Requirements analysis and specification
- Design and specification
- Coding and module testing
- Integration and system testing
- Delivery
- Maintenance

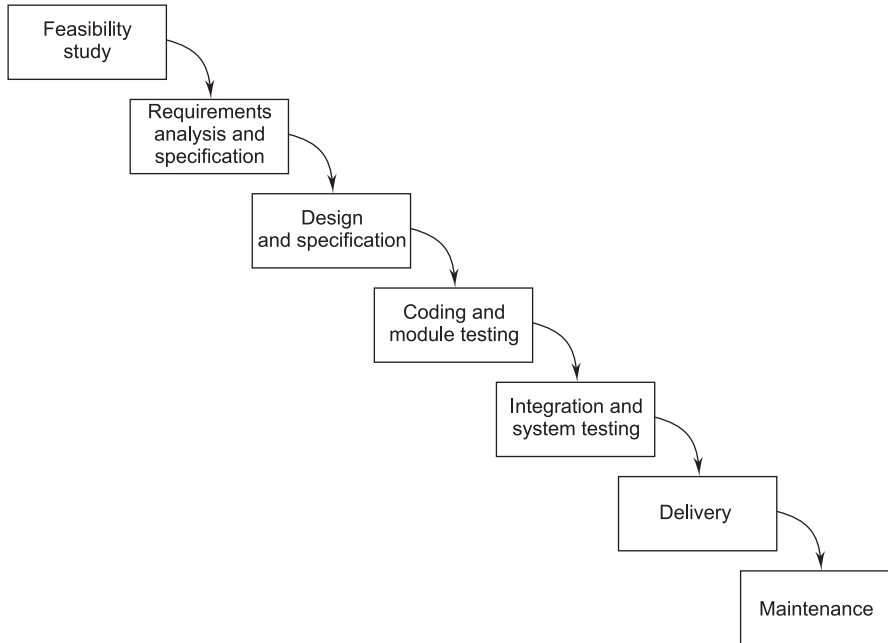


FIGURE 2.3 Waterfall Model

1. **Feasibility Study.** The first phase is the feasibility study. The purpose of this phase is to produce a feasibility study document that evaluates the costs and benefits of the proposed application. To do so, it is first necessary to analyze the problem, at least at a global level. Obviously, the more we understand the problem, the better we can identify alternative solutions, their costs, and their potential benefits to the user. Therefore, ideally, one should perform as much analysis of the problem as is needed to do a well-founded feasibility study. The feasibility study is usually done within limited time bounds and under pressure. Often, its result is an offer to the potential customer. Since we cannot be sure that the offer will be accepted, economic reasons prevent us from investing too many resources into analyzing the problem.

In sum, the feasibility study tries to anticipate future scenarios of software development. Its result is a document that should contain at least the following items:

- A definition of the problem.
- Determination of technical and economic viability.
- Alternative solutions and their expected benefits.

- Required resources, costs, and delivery dates in each proposed alternative solution.

At the end of this phase, a report called a feasibility study is prepared by a group of software engineers. The client or the customer is also consulted through a questionnaire. This report determines whether the project is feasible or not. After being successful in the feasibility study, the requirement analysis is carried out.

2. **Requirement Analysis and Specification.** This phase exactly tells the requirements and needs of the project. This is a very important and critical phase in the waterfall model.

The purpose of a requirements analysis is to identify the qualities required of the application, in terms of functionality, performance, ease of use, portability, and so on.

The requirements describe the “what” of a system, not the “how.” This phase produces a large document and contains a description of what the system will do without describing how it will be done. The resultant document is known as the software requirement specification (SRS) document.

An SRS document must contain the following:

- Detailed statement of problem.
- Possible alternative solution to problem.
- Functional requirements of the software system.
- Constraints on the software system.

The SRS document must be precise, consistent, and complete. There is no scope for any ambiguity or contradiction in the SRS document. An SRS document may be organized as a problem statement, introduction to the problem, functional requirements of the system, non-functional requirements of the system, behavioral descriptions, and validation criteria.

3. **Design and Specification.** The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different design approaches are available: the traditional design approach and the object-oriented design approach.

- (i) *Traditional Design Approach.* The traditional design approach is currently being used by many software-development houses. Traditional design consists of two different activities; first a structured analysis of the

requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

Structured design is undertaken once the structured analysis activity is complete. Structured design consists of two main activities: architectural design (also called high-level design) and detailed design (also called low-level design). High-level design involves decomposing the system into modules, and representing the interfaces and the invocation relationships among the modules. During detailed design, internals of the individual modules are designed in greater detail (e.g., the data structures and algorithms of the modules are designed and documented).

- (ii) *Object-Oriented Design Approach*. This is a new paradigm. Various objects in the system are identified. After the identification of objects, the relationships among them are also explored. The OOD approach has several benefits, such as lower development time and effort and better maintainability.

4. **Coding and Module Testing.** Coding and module testing is the phase in which we actually write programs using a programming language. It was the only recognized development phase in early development processes, but it is just one of several phases in a waterfall process. The output of this phase is an implemented and tested collection of modules.

Coding can be subject to company-wide standards, which may define the entire layout of programs, such as the headers for comments in every unit, naming conventions for variables and sub-programs, the maximum number of lines in each component, and other aspects that the company deems worthy of standardization.

Module testing is also often subject to company standards, including a precise definition of a test plan, the definition of testing criteria to be followed (e.g., black-box versus white-box, or a mixture of the two), the definition of completion criteria (when to stop testing), and the management of test cases. Debugging is a related activity performed in this phase.

Module testing is the main quality-control activity that is carried out in this phase. Other such activities may include code inspections to check adherence to coding standards and, more generally, to check for a disciplined programming style, as well as checking of software qualities other than functional correctness (e.g., performance), although this is often better done at a later stage of coding.

5. **Integration and System Testing.** During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot (can you guess the reason for this?). Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out.

The objective of system testing is to determine whether the software system performs per the requirements mentioned in the SRS document. This testing is known as system testing. A fully developed software product is system tested. The system testing is done in three phases: Alpha, Beta, and Acceptance Testing.

- *Alpha Testing* is conducted by the software-development team at the developer's site.
- *Beta Testing* is performed by a group of friendly customers in the presence of the software-development team.
- *Acceptance Testing* is performed by the customers themselves. If the software is successful in acceptance testing, the product is installed at the customer's site.

6. **Delivery and Maintenance.** The delivery of software is often done in two stages. In the first stage, the application is distributed among a selected group of customers prior to its official release. The purpose of this procedure is to perform a kind of controlled experiment to determine, on the basis of feedback from users, whether any changes are necessary prior to the official release. In the second stage, the product is distributed to the customers.

We define maintenance as the set of activities that are performed after the system is delivered to the customer. Basically, maintenance consists of correcting any remaining errors in the system (corrective maintenance), adapting the application to changes in the environment (adaptive maintenance), and improving, changing, or adding features and qualities to the application (perfective maintenance). Recall that the cost of maintenance is often more than 60% of the total cost of software and that about 20% of maintenance costs may be attributed to corrective and adaptive maintenance, while over 50% is attributable to perfective maintenance. Based on this breakdown, we observed that evolution is probably a better term than maintenance, although the latter is used more widely.

2.2.1 Advantages of Waterfall Model

The various advantages of the waterfall model include:

- It is a linear model.
- It is a segmental model.
- It is systematic and sequential.
- It is a simple one.
- It has proper documentation.

2.2.2 Disadvantages of Waterfall Model

The various disadvantages of the waterfall model include:

- It is difficult to define all requirements at the beginning of a project.
- This model is not suitable for accommodating any change.
- A working version of the system is not seen until late in the project's life.
- It does not scale up well to large projects.
- It involves heavy documentation.
- We cannot go backward in the SDLC.
- There is no sample model for clearly realizing the customer's needs.
- There is no risk analysis.
- If there is any mistake or error in any phase then we cannot make good software.
- It is a document-driven process that requires formal documents at the end of each phase.

2.3 PROTOTYPING MODEL

It always happens that a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these and many other situations, a prototyping paradigm may offer the best approach.

Prototyping begins with communication. The developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A quick design then occurs. The quick design focuses on the representation of those aspects

of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype. The prototype is evaluated by the customer/user to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools that enable working programs to be generated quickly.

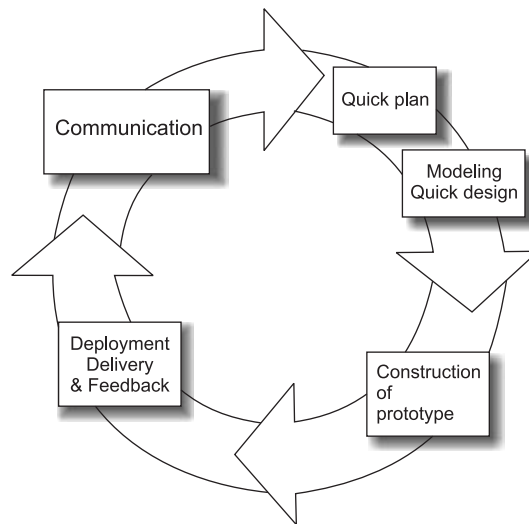


FIGURE 2.4 Prototyping Model

2.3.1 Reasons for Using Prototyping Model

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is valuable for gaining a better understanding of the customer's needs. The prototype model is very useful in developing the Graphical User Interface (GUI) part of a system.

The prototyping model can be used when the technical solutions are unclear to the development team. Often, major design decisions depend on issues, such as the response time of a hardware controller or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues. The third reason for developing a prototype is that it is impossible

to “get it right” the first time and one must plan to throw away the first product in order to develop a good product later, as advocated.

2.3.2 Controlling Changes During Prototyping

A major problem with prototyping is controlling changes to the prototype following suggestions by the users. One approach has been to categorize changes as belonging to one of three types:

- **Cosmetic (about 35% of changes)**

These are simply changes to the layout of the screen. They are:

- (a) Implemented.
- (b) Recorded.

- **Local (about 60% of changes)**

These involve changes to the way the screen is processed but do not affect other parts of the system. They are:

- (a) Implemented.
- (b) Recorded.
- (c) Backed-up so that they can be removed at a later stage is necessary.
- (d) Inspected retrospectively.

- **Global (about 5% of changes)**

These are changes that affect more than one part of the processing. All changes here have to be the subject of a design review before they can be implemented.

2.3.3 Advantages of Prototyping Models

- Suitable for large systems for which there is no manual process to define the requirements.
- User training to use the system.
- User services determination.
- System training.
- Quality of software is good.
- Requirements are not frozen.

2.3.4 Limitations of Prototyping Model

- It is difficult to find all the requirements of the software initially.
- It is very difficult to predict how the system will work after development.

These two limitations are removed in the prototyping model.

The first limitation is removed by unfreezing the requirements before any design or coding can proceed.

The second limitation is removed by making a throw-away prototype to understand the requirements.

2.4 SPIRAL MODEL

The spiral model, originally proposed by Boehm, is an evolutionary software model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear segmental model.

The goal of the spiral model of the software production process is to provide a framework for designing such processes, guided by the risk levels in the projects at hand. As opposed to the previously presented models, the spiral model may be viewed as a metamodel, because it can accommodate any process-development model. By using it as a reference, one may choose the most appropriate development model (e.g., evolutionary versus waterfall). The guiding principle behind such a choice is the level of risk; accordingly, the spiral model provides a view of the production process that supports risk management.

Let us present a few definitions. Risks are potentially adverse circumstances that may impair the development process and the quality of products. Boehm [1989] defines risk management as a *discipline whose objectives are to identify, address, and eliminate software risk items before they become either threats to successful software operation or a major source of expensive software rework*. The spiral model focuses on identifying and eliminating high-risk problems by careful process design, rather than treating both trivial and severe problems uniformly.

The spiral model is recommended where the requirements and solutions call for developing full-fledged, large, complex systems with many features and facilities from scratch. It is used when experimenting on technology, trying out new skills, and when the user is not able to offer requirements in clear terms. It is also useful when the requirements are not clear and when the solution intended has multi-users, multi-functions, multi-features, multi-location applications to be used on multiple platforms, where seamless integration, interfacing, data migration, and replication are the issues. The radial dimension of a cycle represents the cumulative costs, and the angular dimension represents the progress made in completing each cycle of the spiral.

Each phase in this model is split into four sectors (or quadrants) as shown in Figure 2.5.

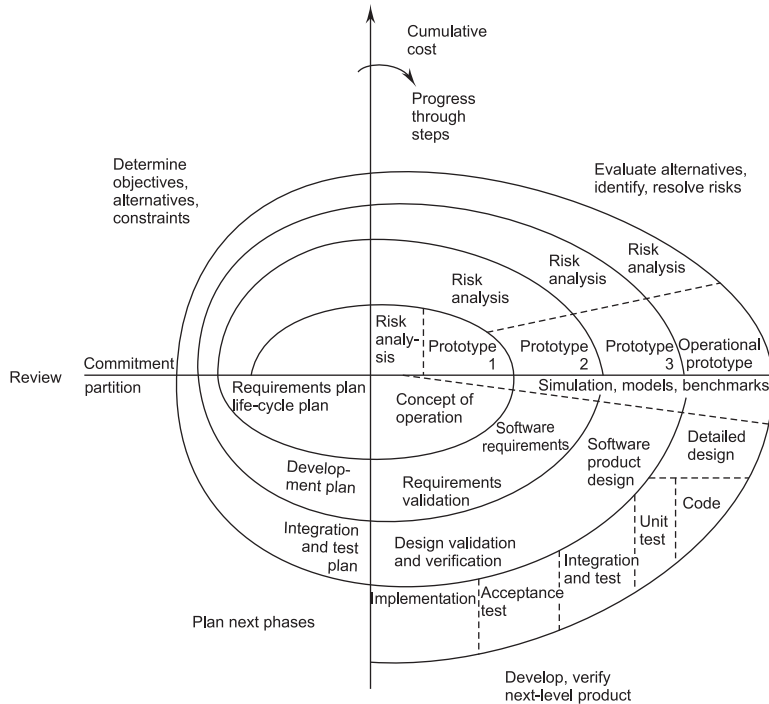


FIGURE 2.5 The Spiral Model (From Boehm [1988])

The first quadrant identifies the objectives of the phase and the alternative solutions possible for the phase under consideration.

In the second quadrant we evaluate different alternatives based on the objectives and constraints. The evaluation is based on the risk perceptions for the project.

The third quadrant is the next step that emphasizes development of strategies that resolve the uncertainties and risks. This may involve activities, such as benchmarking, simulation, and prototyping.

In the last or fourth quadrant we determine the objective that should be fulfilled in the next cycle of our software development in order to build the complete system.

2.4.1 Characteristics of Spiral Model

The main characteristic of the Spiral Model is that it is cyclic and not linear like the waterfall model (see Figure 2.5). Each cycle of the spiral consists of four stages, and each stage is represented by one quadrant of the Cartesian diagram. The radius of the spiral represents the cost accumulated so far in the process; the angular dimension represents the progress in the process.

2.4.2 Limitations of Spiral Model

There are some limitations of the spiral model which include:

- No strict standards for software development.
- No particular beginning or end of a particular phase.

2.4.3 Advantages of Spiral Model

There are some advantages of the spiral model which include:

- It is risk-driven model.
- It is very flexible.
- Less documentation is needed.
- It uses prototyping.

2.4.4 Disadvantages of Spiral Model

The spiral model has some disadvantages that need to be resolved before it can be a universally applied life-cycle model. The disadvantages 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.

2.5 EVOLUTIONARY DEVELOPMENT MODEL

In the Evolutionary Model, development engineering effort is made first to establish correct, precise requirement definitions and system scope, as agreed by all the users across the organization. This is achieved through application of iterative processes to evolve a system most suited to the given circumstances. The process is *iterative* as the software engineer goes through a repetitive process of requirement called Analysis-Design-Testing through Prototype-Implementation-Assessment-Evaluation until all users and stakeholders are satisfied.

This model differs from the 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.

2.5.1 Need of an Evolutionary Model

The various reasons why there exists a need for an evolutionary model include:

- Business and product requirements often change as development proceeds.
- Tight market deadlines make completion of a comprehensive software product impossible but a limited version must be introduced to meet competitive and business pressures.

- A set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined.

2.5.2 Uses of Evolutionary Model

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.

2.6 ITERATIVE-ENHANCEMENT MODEL

The iterative-enhancement model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. In this model, the software is broken down into several modules, which are incrementally developed and delivered. First, the development team develops the core module of the system and then it is later refined into increasing levels of capability of adding new functionalities in successive versions.

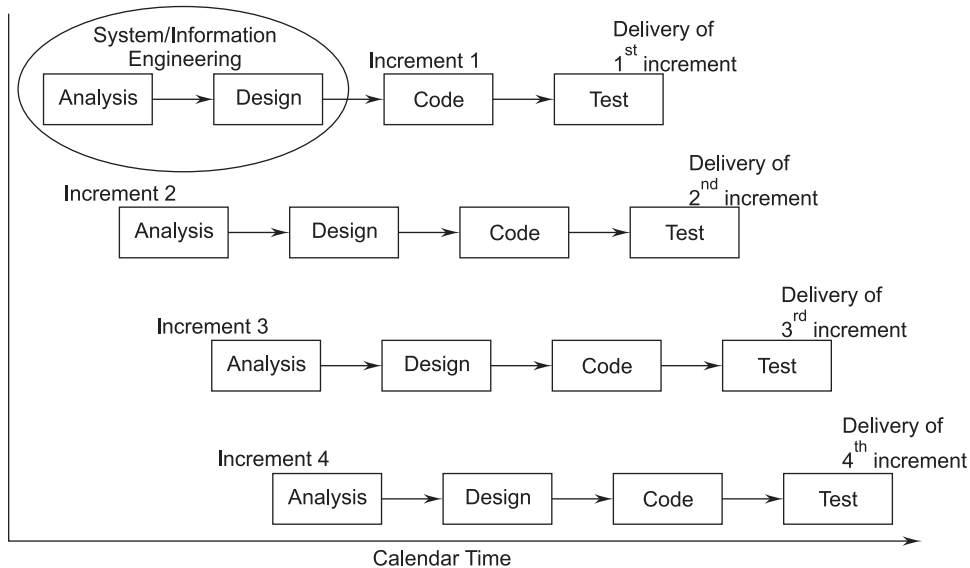


FIGURE 2.6 Iterative-Enhancement Model

Each linear sequence produces a deliverable *increment* of the software. For example, word-processing software developed using the iterative paradigm might deliver basis file management, editing, and document production functions in the

first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment could incorporate the prototyping paradigm.

When an iterative-enhancement model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, other unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

2.6.1 Advantages of Iterative-Enhancement Model

The various advantages of following the approach of the iterative-enhancement model are as follows:

- The feedback from early increments improve the later stages.
- The possibility of changes in requirements is reduced because of the shorter time span between the design of a component and its delivery.
- Users get benefits earlier than with a conventional approach.
- Early delivery of some useful components improves cash flow, because you get some return on investment early on.
- Smaller sub-projects are easier to control and manage.
- ‘Gold-plating,’ that is the requesting of features that are unnecessary and not in fact used, is less as users will know that if a feature is not in the current increment then it can be included in the next.
- The project can be temporarily abandoned if more urgent work crops up.
- Job satisfaction is increased for developers who see their labors bearing fruit at regular, short intervals.

2.6.2 Disadvantages of Iterative-Enhancement Model

The various disadvantages of the iterative-enhancement model include:

- Software breakage, that is, later increments may require modifications to earlier increments.
- Programmers may be more productive working on one large system than on a series of smaller ones.

- Some problems are difficult to divide into functional units (modules), which can be incrementally developed and delivered.

2.7 RAD MODEL

The RAD (Rapid Application Development Model) model is proposed when requirements and solutions can be modularized as independent system or software components, each of which can be developed by different teams. After these smaller system components are developed, they are integrated to produce the large software system solution. The modularization could be on a functional, technology, or architectural basis, such as front-end-back-end, client side-server side, and so on.

RAD becomes faster if the software engineer uses the component's technology such that the components are really available for reuse. Since the development is distributed into component-development teams, the teams work in tandem and total development is completed in a short period (i.e., 60 to 90 days). Figure 2.7 shows the RAD model.

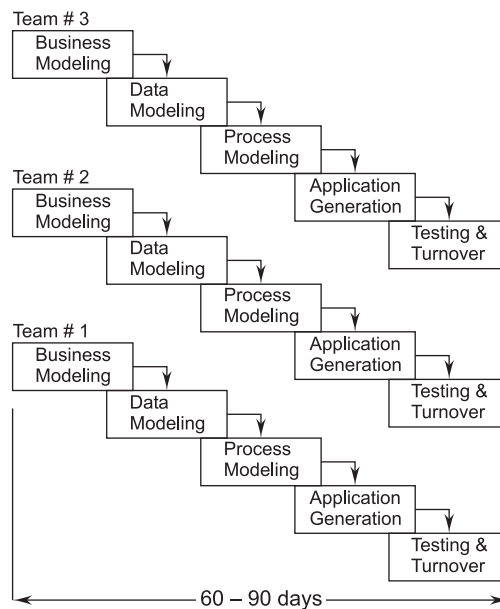


FIGURE 2.7 The RAD Model

1. **Business Modeling.** It covers the following functions:

- Where does information come from and go?

- Who processes it?
 - What information drives the business process?
 - What information is generated?
2. **Data Modeling.** The information flow defined as part of the business-modeling phase is refined into a set of data objects that are needed to support the business. The characteristics of each object are identified and the relationships between these objects are defined.
 3. **Process Modeling.** In this model, information flows from object to object to implement a business function. To add, modify, delete, or retain a data object, there is a need for description which is done in this phase.
 4. **Application Generation.** RAD assumes the use of fourth-generation techniques. The RAD process works to reuse existing program components or create reusable components. To facilitate the construction of the software using the above cases, automated tools are used.
 5. **Testing and Turnover.** In this phase we have to test the programs, but we use some already existing programs which are already tested, so the time involved in testing is less. Only the new programs or components must be tested.

2.7.1 Disadvantages of RAD Model

The various disadvantages of the RAD model include:

- For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated timeframe, RAD projects will fail.
- If a system cannot be properly modularized, building the components necessary for RAD will be problematic.
- If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
- RAD may not be appropriate when technical risks are high (for example, when a new application makes heavy use of new technology).

2.8 COMPARISON OF VARIOUS PROCESS MODELS

The comparison between various process models is given in Table 2.2.

TABLE 2.2 Comparison of Process Models

Strengths	Weaknesses	Types of Projects
<p>WATERFALL</p> <p>Simple</p> <p>Easy to execute</p> <p>Intuitive and logical</p>	<p>All or nothing approach</p> <p>Requirements frozen early</p> <p>Disallows changes</p> <p>Cycle time too long</p> <p>May choose outdated hardware technology</p> <p>User feedback not allowed</p> <p>Encourages requirement bloating</p>	<p>For well-understood problems</p> <p>Short duration projects</p> <p>Automation of existing manual systems</p>
<p>PROTOTYPING</p> <p>Helps in requirements elicitation</p> <p>Reduces risk</p> <p>Leads to a better system</p>	<p>Front heavy process</p> <p>Possibly higher costs</p> <p>Disallows later changes</p>	<p>Systems with novice users</p> <p>When there are uncertainties in requirements</p>
<p>ITERATIVE ENHANCEMENT</p> <p>Regular/quick deliveries</p> <p>Reduces risk</p> <p>Accommodates changes</p> <p>Allows user feedback</p> <p>Allows reasonable exit points</p> <p>Avoids requirement bloating</p> <p>Prioritizes requirements</p>	<p>Each iteration can have planning overhead</p> <p>Costs may increase as work done in one iteration may have to be undone later</p> <p>System architecture and structure may suffer as frequent changes are made</p>	<p>For businesses where time is of essence</p> <p>Where risk of a long project cannot be taken</p> <p>Where requirements are not known</p>
<p>SPIRAL</p> <p>Controls project risks</p> <p>Very flexible</p> <p>Less documentation needed</p>	<p>No strict standards for software development</p> <p>No particular beginning or end of particular phase</p>	<p>Projects built on untested assumptions</p>

EXERCISES

1. Discuss the SDLC in brief.
2. Give the basic phases in the software-development life-cycle.
3. What are the different steps in the software-development life-cycle? What are the end products at each step?
4. What are the important activities that are carried out during the feasibility study phase?
5. Explain the different categories of maintenance in the software-development life-cycle.
6. What is the role of the testing phase in the software-development life-cycle?
7. Draw the schematic diagram of the waterfall model of software development. Also discuss its phases in brief.
8. Explain the waterfall model in detail with the help of a diagram. State its advantages and also its limitations.
9. What is a prototype? Draw the schematic diagram of the prototyping model of software development. Also discuss its phases in brief.
10. What are the major advantages of first constructing a working prototype before developing the actual product?
11. Write a short description of the evolutionary development model. Also state its advantages.
12. Explain the iterative-enhancement model with the help of a suitable example.
13. Which life-cycle model would you follow for developing software for each of the following applications? Justify your selection of model with an appropriate reason:
 - (i) A game
 - (ii) A new text editor
 - (iii) A compiler for a new language
 - (iv) A software for hospital management
14. How are the risks associated with a project handled in the spiral model of software development?
15. What are the major phases in the spiral model of software development? Explain.
16. List the various models of software development and explain the RAD model in detail.
17. Compare the spiral model with the prototyping model by giving their advantages and disadvantages.
18. What is the need and use of the evolutionary development model?
19. Discuss the advantages and disadvantages of various models of software engineering.
20. Give a short comparison between the various software-development life-cycle models.

INTRODUCTION TO SOFTWARE REQUIREMENTS SPECIFICATION

3.1 REQUIREMENT ENGINEERING

A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfill the system's purpose.

Figure 3.1 illustrates the process of determining the requirements for a software-based system.

Requirements describe the “what” of a system, not the “how.” Requirements engineering produces one large document, written in a natural language, and contains a description of what the system will do without describing how it will do it.

Requirements engineering is the systematic use of proven principles, techniques, and language tools for the cost-effective analysis, documentation, and on-going evaluation of the user's needs and the specifications of the external behavior of a system to satisfy those user needs. It can be defined as a discipline, which addresses requirements of objects all along a system-development process.

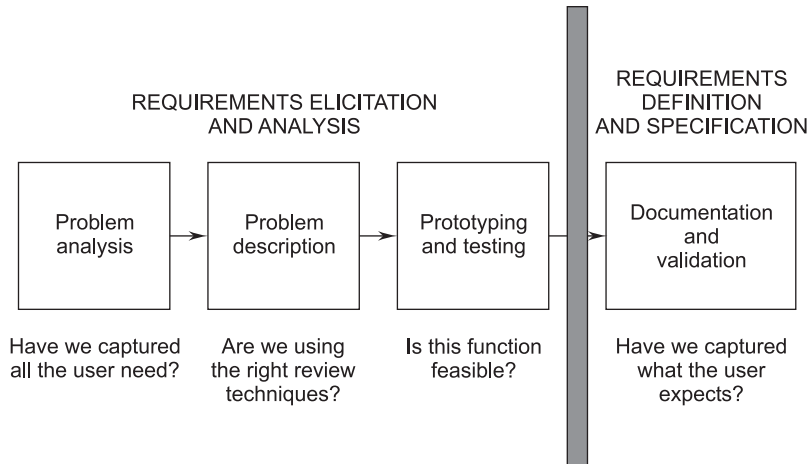


FIGURE 3.1 The Process of Determining Requirements

The input to requirements engineering is the problem statement prepared by the customer. The output of the Requirements Engineering (RE) process is a system requirements specification called the Requirement Definition and Description (RDD). The system requirements specification forms the basis for designing software solutions.

3.1.1 Types of Requirements

There are various categories of the requirements. On the basis of their priority, the requirements are classified into the following three types:

- Those that should be absolutely met.
- Those that are highly desirable but not necessary.
- Those that are possible but could be eliminated.

On the basis of their functionality, the requirements are classified into the following two types:

- (i) **Functional requirements.** They define factors, such as I/O formats, storage structure, computational capabilities, timing, and synchronization.
- (ii) **Non-functional requirements.** They define the properties or qualities of a product including usability, efficiency, performance, space, reliability, portability, etc.

3.2 PROCESS OF REQUIREMENTS ENGINEERING

Figure 3.2 illustrates the process steps of requirements engineering.

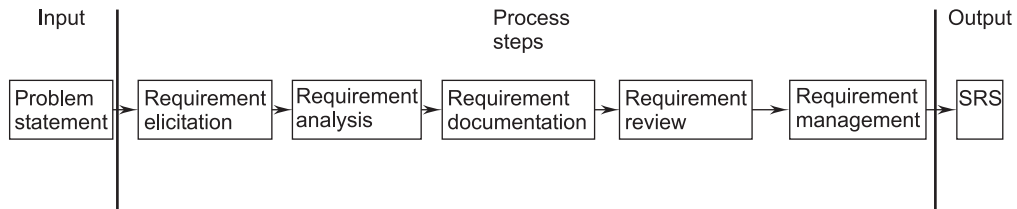


FIGURE 3.2 Process Steps of Requirements Engineering

Requirements engineering consists of the following processes:

- Requirements gathering (elicitation).
- Requirements analysis and modeling.
- Requirements documentation.
- Requirements review.
- Requirements management.

3.2.1 Requirement Elicitation and Analysis

Requirement Elicitation/Gathering. Requirement gathering is a communication process between the parties involved and affected in the problem situation. The tools in elicitation are meetings, interviews, video conferencing, e-mails, and existing documents study and facts findings. More than 90% to 95% elicitation should be complete in the initiation stage while the remaining 5% is completed during the development life-cycle. The requirements are gathered from various sources. The sources are:

- Customer (Initiator)
- End Users
- Primary Users
- Secondary Users
- Stakeholders

Requirement Analysis. Requirement analysis is a very important and essential activity after elicitation. In this phase, each requirement is analyzed from the point-of-view of validity, consistency, and feasibility for firm consideration in the RDD and then in the SRS. Validity confirms its relevance to goals and objectives and consistently confirms that it does not conflict with other requirements but supports others where necessary. Feasibility ensures that the necessary inputs are

available without bias and error, and technology support is possible to execute the requirement as a solution. This portion of the analysis confirms the place of the requirements in RDD on its own and along with others.

The second portion of analysis attempts to find for each requirement, its functionality, features, and facilities and the need for these under different conditions and constraints. Functionality states “how to achieve the requirement goal.” Features describe the attributes of functionality, and facilities provide its delivery, administration, and communication to other systems.

Process Model of Elicitation and Analysis

A generic process model of the elicitation and analysis process is shown in Figure 3.3. Each organization will have its own version or instantiation of this general model depending on local factors, such as the expertise of the staff, the type of system being developed, the standards used, etc.

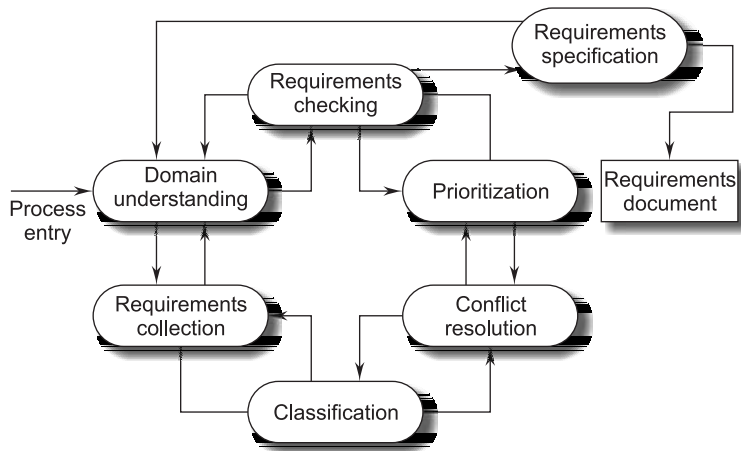


FIGURE 3.3 Requirements Elicitation and Analysis Process Model

The process activities are:

1. **Domain Understanding.** Analysts must develop their understanding of the application domain. For example, if a system for a supermarket is required, the analyst must find out how supermarkets operate.
2. **Requirements Collection.** This is the process of interacting with stakeholders in the system to discover their requirements. Obviously, domain understanding develops further during this activity.
3. **Classification.** This activity takes the unstructured collection of requirements and organizes them into coherent clusters.

4. **Conflict Resolution.** Inevitably, where multiple stakeholders are involved, requirements will conflict. This activity is concerned with findings and resolving these conflicts.
5. **Prioritization.** In any set of requirements some priorities will be more important than others. This stage involves interaction with stakeholders to discover the most important requirements.
6. **Requirements Checking.** The requirements are checked to discover if they are complete, consistent, and in accordance with what stakeholders really want from the system.

3.2.2 Requirements Documentation

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

The SRS is a specification for a particular software product, program, or set of programs that perform 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. In the first case, the SRS is used to define the needs and expectations of the users. In the second case, the SRS is written for a different purpose and serves as a contract document between customer and developer.

Thus, requirements must be written so they are meaningful not only to the customers but also to designers on the development team.

Requirements Definition Document

The system documentation contains a record of the requirements in the customer's terms. This requirements definition document describes what the customer would like to see.

- First, we outline the general purpose of the system. References to other related systems are included, and we incorporate any terms and abbreviations that may be useful.
- Next, we describe the background and objectives of system development. For example, if a system is to replace an existing approach, we explain why the existing system is unsatisfactory. Current methods and procedures are outlined in enough detail so we can isolate those elements with which the customer is happy from those that are disappointing.

- If the customer has a proposed new approach to solving the problem, we outline a description of the approach. Remember, though, that the purpose of the requirements document is to discuss the problem, not the solution; the focus should be on how the system is to meet the customer's needs.
- Once we record this overview of the problem, we describe the detailed characteristics of the proposed system. We define the system boundaries and interfaces across it. The system functions are also explained. Also, we include a complete list of data elements and classes and their characteristics.
- Finally, we discuss the environment in which the system will operate. We include requirements for support, security, and privacy and any special hardware or software constraints should be addressed.

3.2.3 Requirements Review

A requirements review is a manual process, which involves multiple readers from both client and contractor staff checking the requirements document for anomalies and omissions.

A requirements review can be informal or formal.

1. **Informal Review.** Informal reviews simply involve contractors discussing requirements with as many system stakeholders as possible. It is surprising how often communication between system developers and stakeholders ends after elicitation and there is no conformation that the documented requirements are what the stakeholders really said they wanted. Many problems can be detected simply by talking about the system to stakeholders before making a commitment to a formal review.
2. **Formal Review.** In a formal requirements review, the development team should 'walk' the client through the system requirements, explaining the implications of each requirement. The review team should check each requirement for consistency and should check the requirements as a whole for completeness. Reviewers may also check for:
 - (a) *Verifiability*: are the requirements as stated realistically testable?
 - (b) *Comprehensibility*: is the requirement property understood by the procurers or end-users of the system?
 - (c) *Traceability*: is the origin of the requirement clearly stated? You may have to go back to the source of the requirement to assess the impact of a change. Traceability is important as it allows the impact of change on the rest of the system to be assessed. We discuss it in more detail in the following section.
 - (d) *Adaptability*: is the requirement adaptable? That is, can the requirement be changed without large-scale effects on other system requirements?

Conflicts, contradictions, errors, and omissions in the requirements should be pointed out during the review and formally recorded. It is then up to the users, the system procurer, and the system developer to negotiate a solution to these identified problems.

3.2.4 Requirements Management

Requirements define the capability that the software system solution must deliver and the intended results that must result on its application to business problems. In order to generate such requirements, a systematic approach is necessary, through a formal management process called Requirements Management.

Requirements management is defined as a systematic approach to eliciting, organizing, and documenting the requirements of the system, and a process that establishes and maintains agreement between the customer and project team on the changing requirements of the system.

Classes of Requirements Management

From an evolution perspective, requirements fall into two classes:

1. *Enduring requirements.* These are relatively stable requirements which derive from the core activity of the organization and which relate directly to the domain of the system. For example, in a hospital there will always be requirements concerned with patients, doctors, nurses, treatments, etc. These requirements may be derived from domain models that show the entities and relations which characterize an application domain.

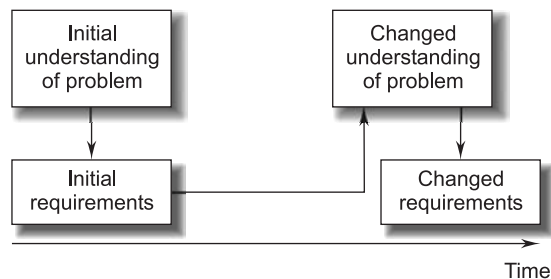


FIGURE 3.4 Requirements Evolution

2. *Volatile requirements.* These are requirements which are likely to change during the system development or after the system has been put into operation. Examples of volatile requirements are requirements resulting from government health-care policies.

Requirements Management Planning

Planning is an essential first stage in the requirements management process. Requirements management is very expensive and, for each project, the planning stage establishes the level of requirements management detail required. During the requirements management stage, you have to decide on:

1. *Requirements identification.* Each requirement must be uniquely identified so that it can be cross-referred by other requirements and so that it may be used in traceability assessments.
2. *A change management process.* This is the set of activities that assess the impact and cost of changes. We discuss it in more detail in the following section.
3. *Traceability policies.* These policies define the relationships between requirements and between the requirements and the system design that should be recorded and how these records should be maintained.
4. *CASE tool support.* Requirements management involves the processing of large amounts of information about the requirements. Tools, which may be used, range from specialist requirements management system to spreadsheets and simple database systems.

The process steps and their outputs which when implemented will lead to the most acceptable RDD and SRS through requirements management are given in Table 3.1.

TABLE 3.1 Process Steps for Requirements Management

Steps	Outputs
Study the domain	Domain knowledge improved for better solutions.
Analyze the problems	Increased understanding of the problem.
Understand user needs	Real and genuine needs that need solving identified.
Determine and define the system and system scope	First prototype system model and its scope definition.
Manage scope	System scope: Feasible and deliverable.
Refine and evaluate the system definition	Broader system scope defined in terms of deliverables.
Build the right system	A system accepted by the users.

3.3 INFORMATION MODELING

The Information Flow Model (IFM) is used to understand the sources and destination of information flow, which is required to execute the business process as shown in Figure 3.5.

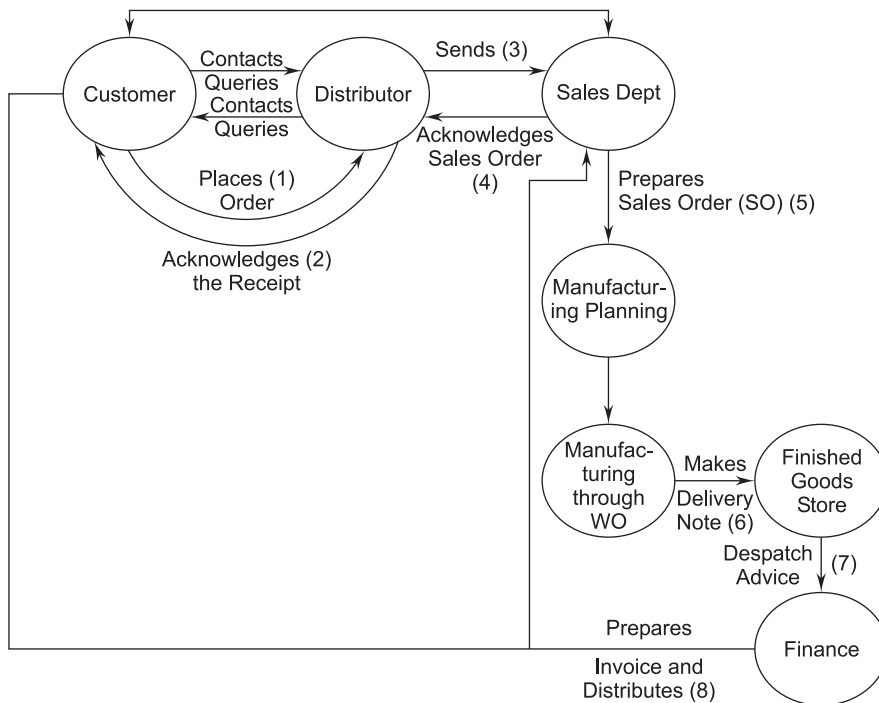


FIGURE 3.5 Information Flow Model

In IFM, information or data generators and processors are brought together to explain the flow. This could be documents, e-mail, or voicemail. The contents of the flow could be text, images, or diagrams. The purpose of the flow is to take the process further to its logical conclusion. For example, a customer order is to be processed for delivery or to be rejected, and necessary data or information input has to be provided progressively in the process.

IFM is generally a high-level model showing main flows, internal flows of information from sources, such as product catalogs, and manufacturing schedules. Customer profiles and accounting information are not shown. These are presumed to be present.

3.4 DATA-FLOW DIAGRAMS

Data-Flow Diagrams (DFD) are also known as data-flow graphs or bubble charts. A DFD serves the purpose of clarifying system requirements and identifying major transformations. DFDs show the flow of data through a system. It is an important modeling tool that allows us to picture a system as a network of functional processes.

Data-flow diagrams are well-known and widely used for specifying the functions of an information system. They describe systems as collections of data that are manipulated by functions. Data can be organized in several ways: they can be stored in data repositories, they can flow in data flows, and they can be transferred to or from the external environment.

One of the reasons for the success of DFDs is that they can be expressed by means of an attractive graphical notation that makes them easy to use.

3.4.1 Symbols Used for Constructing DFDs

There are different types of symbols used to construct DFDs. The meaning of each symbol is explained below:

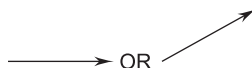
1. **Function symbol.** A function is represented using a circle. This symbol is called a process or a bubble and performs some processing of input data.



2. **External entity.** A square defines a source or destination of system data. External entities represent any entity that supplies or receives information from the system but is not a part of the system.



3. **Data-flow symbol.** A directed arc or arrow is used as a data-flow symbol. A data-flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.



4. **Data-store symbol.** A data-store symbol is represented using two parallel lines. A logical file can represent either a data-store symbol, which can represent either a data structure, or a physical file on disk. Each data store is connected

to a process by means of a data-flow symbol. The direction of the data-flow arrow shows whether data is being read from or written into a data store.

5. **Output Symbol.** It is used to represent data acquisition and production during human-computer interaction.



3.4.2 Example DFD

Example 3.1. Figure 3.6 shows how the symbols can be composed to form a DFD. The DFD describes the arithmetic expression

$$(a + b) * (c + a * d)$$

assuming that the data a , b , c , and d are read from a terminal and the result is printed. The figure shows that the arrow can be “forked” to represent the fact that the same datum is used in different places.

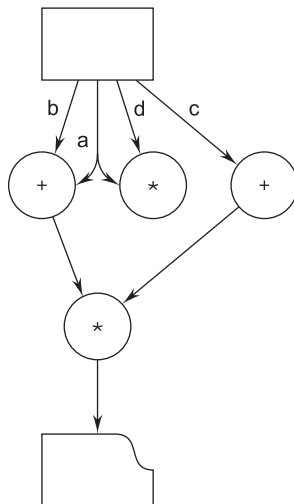


FIGURE 3.6 A DFD For Specifying the Arithmetic Expression $(a + b) * (c + a * d)$

Example 3.2. Figure 3.7 describes a simplified information system for a public library. The data and functions shown are not necessarily computer data and computer functions. The DFD describes physical objects, such as books and shelves, together with data stores that are likely to be, but are not necessarily, realized as computer files. Getting a book from the shelf can be done either automatically—by

a robot—or manually. In both cases, the action of getting a book is represented by a function depicted by a bubble. The figure could even represent the organization of a library with no computerized procedures.

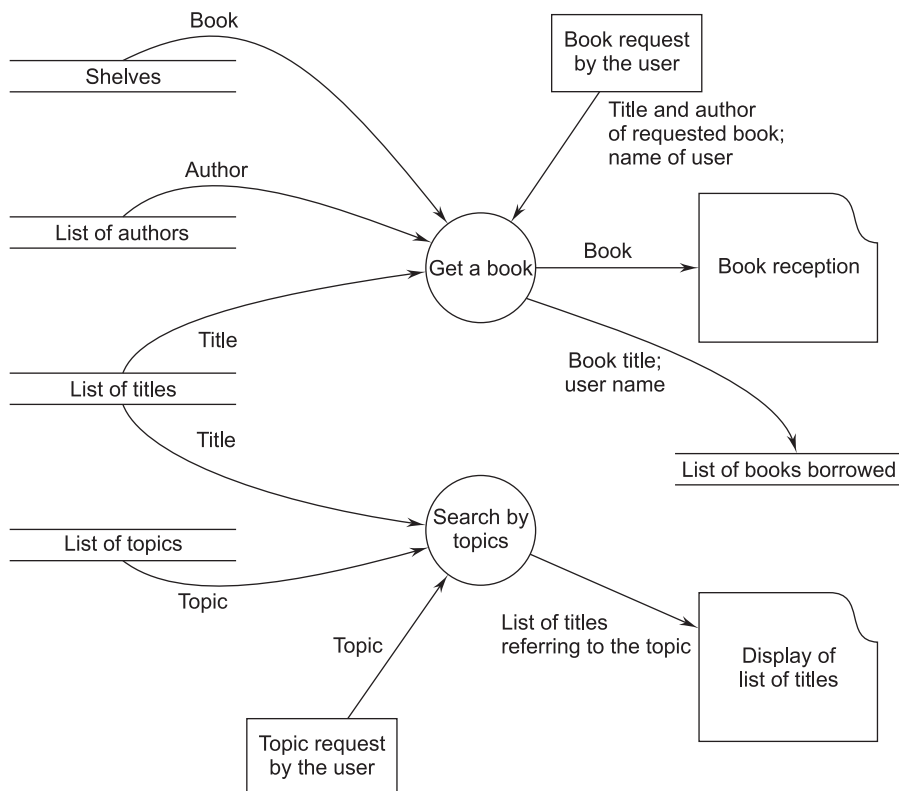


FIGURE 3.7 A DFD Describing a Simplified Library Information System

Figure 3.7 also describes the fact that, in order to obtain a book, the following are necessary: an explicit user request consisting of the title and the name of the author of the book and the user's name; access to the shelves that contain the books; a list of authors; and a list of titles. These provide the information necessary to find the book.

3.4.3 Levels of a DFD

There are different levels of a data-flow diagram. The initial level is called the context level or fundamental system model or a 0-level DFD. If we expand the 0-level processes then we get the 1st-level DFD and if we further expand the 1st-level processes then we get the 2nd-level DFD and so on.

Example 3.3. The 0th and 1st levels of the DFD of a Production Management System are shown in Figure 3.8 (a) and (b).

Let us discuss the data-flow diagram of the Production Management System.

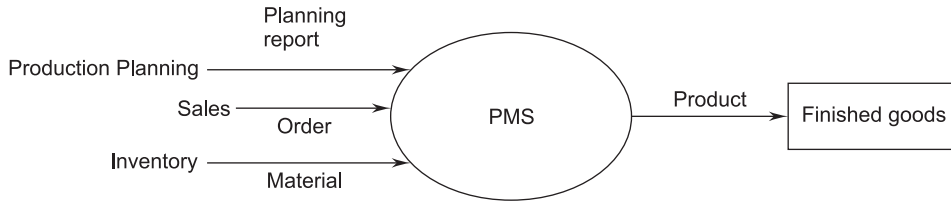


FIGURE 3.8 (a) Level 0 DFD of PMS

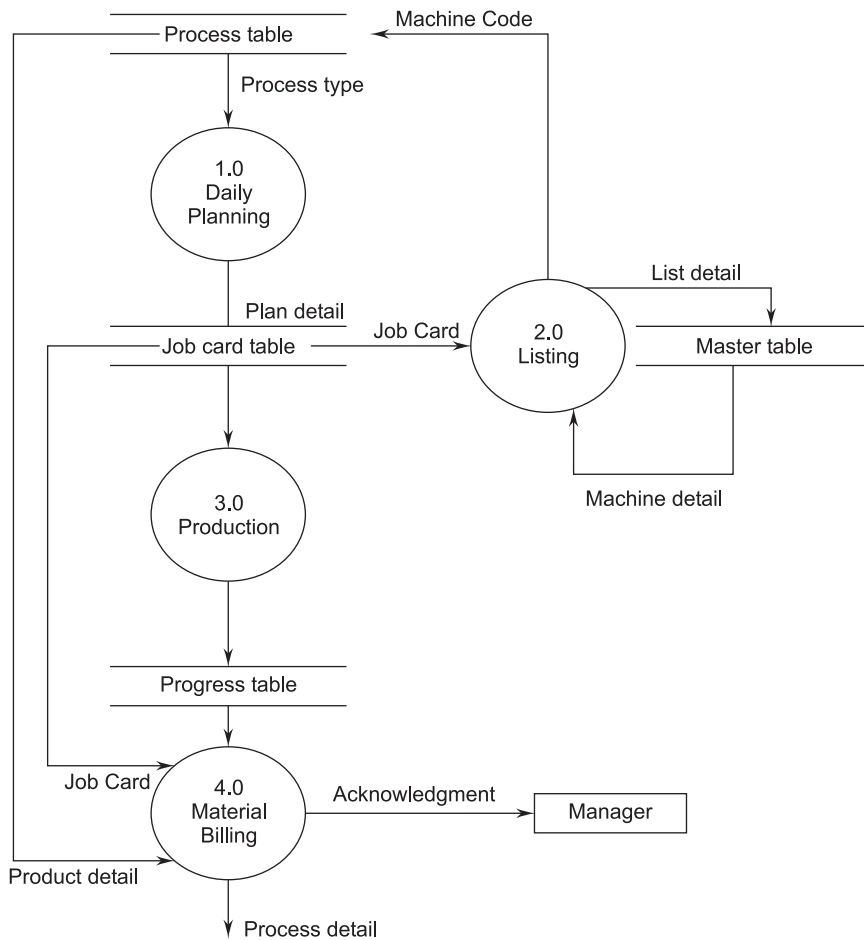


FIGURE 3.8 (b) Level 1 DFD of PMS

Data-flow diagrams can be expressed using informal notations, as illustrated in Figure 3.9 (a), or special symbols can be used to denote processing nodes, data sources, data sinks, and data stores, as illustrated in Figure 3.9 (b).

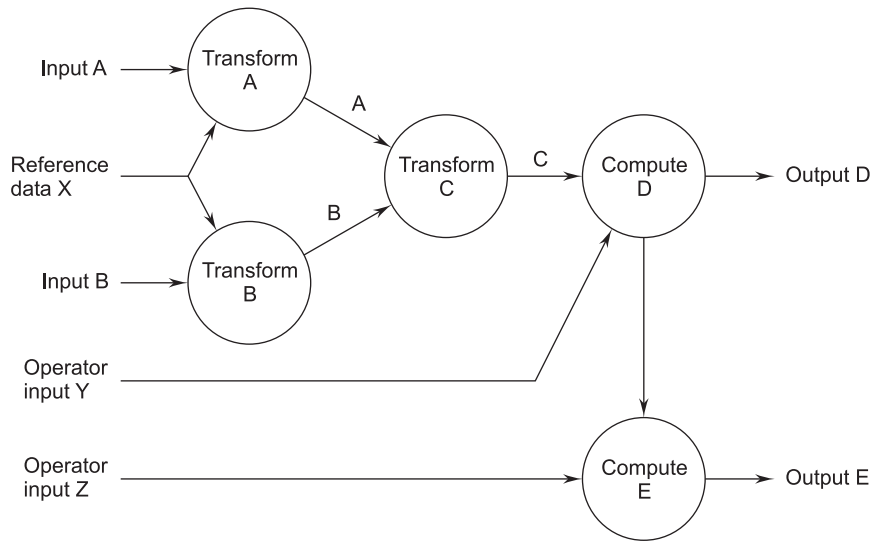


FIGURE 3.9 (a) An Informal DFD or Bubble Chart

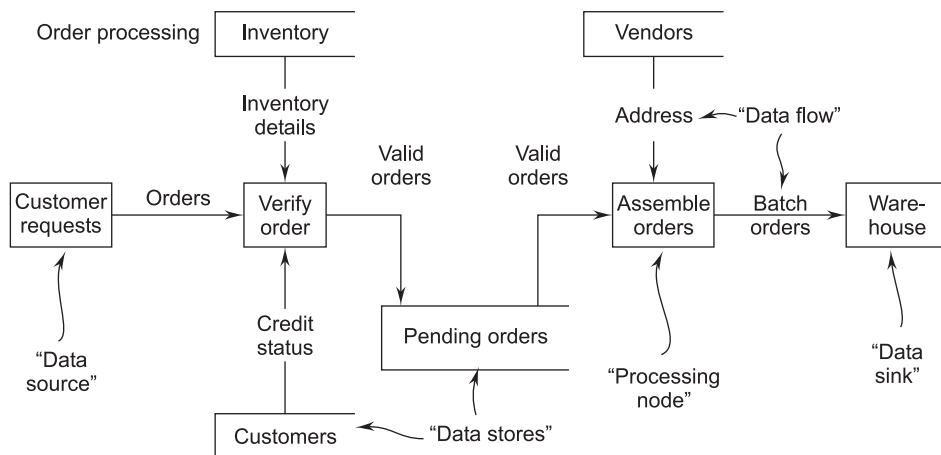


FIGURE 3.9 (b) A Formal DFD or Bubble Chart

3.4.4 General Guidelines and Rules for Constructing DFDs

The following guidelines will help avoid constructing DFDs that are quite simply wrong or incomplete.

- Remember that a DFD is not a flowchart.
- All names should be unique.
- Processes are always running; they do not start or stop.
- All data flows are named.
- Keep a note of all the processes and external entities. Give unique names to them. Identify the manner in which they interact with each other.
- Do number processes.
- Avoid complex DFDs (if possible).
- The DFD should be internally consistent.
- Every process should have a minimum of one input and one output.
- Only data needed to perform the process should be an input to the process.
- The direction of flow is from source to destination.

3.5 DECISION TABLES

Sometimes it is convenient to describe a system as a set of possible conditions satisfied by the system at a given time, rules for reacting to stimuli when certain sets of those conditions are met, and actions to be taken as a result.

Decision tables provide a mechanism for recording complex decision logic. Decision tables are widely used in data-processing applications and have extensively developed literature. As illustrated in Table 3.2, a decision table is segmented into four quadrants: condition stub, condition entry, action stub, and action entry.

TABLE 3.2 Basic Elements of a Decision Table

	Decision rules			
	Rule 1	Rule 2	Rule 3	Rule 4
(Condition stub)		(Condition entries)		
(Action stub)		(Action entries)		

The condition stub contains all of the conditions being examined. Condition entries are used to combine conditions into decision rules. The action stub describes

the actions to be taken in response to decision rules, and the action entry quadrant relates decision rules to actions.

TABLE 3.3 Limited-Entry Decision Table

	1	2	3	4
Credit limit is satisfactory	Y	N	N	N
Pay experience is favorable	-	Y	N	N
Special clearance is obtained	-	-	Y	N
Perform approve order	X	X	X	
Go to reject order				X

Table 3.3 illustrates the format of a limited-entry decision table (entries are limited to Y, N, -, and X). In a limited-entry decision table, Y denotes "yes," N denotes "no," - denotes "don't care," and X denotes "perform action." According to Table 3.3, orders are approved if the credit limit is not exceeded, or if the credit limit is exceeded but past experience is good, or if a special arrangement has been made. If none of these conditions hold, the order is rejected.

The (Y, N, -) entries in each column of the condition entry quadrant form a decision rule. If more than one decision rule has identical (Y, N, -) entries, the table is said to be ambiguous. Ambiguous pairs of decision rules that specify identical actions are said to be redundant, and those specifying different actions are contradictory. Contradictory rules permit specification of non-deterministic and concurrent actions. Table 3.4 illustrates redundant rules (R3 and R4) and contradictory rules (R2 and R3, and R2 and R4).

TABLE 3.4 An Ambiguous Decision Table

	Decision rules			
	Rule 1	Rule 2	Rule 3	Rule 4
C1	Y	Y	Y	Y
C2	Y	N	N	N
C3	N	N	N	N
A1	X			
A2		X		
A3			X	X

A decision table is complete if every possible set of conditions has a corresponding action prescribed. There are 2^N combinations of conditions in a table that has N conditions entries. Failure to specify an action for any one of the combinations results in an incomplete decision table. For example, in Table 3.5 the combination (N, N, N) for conditions C1, C2, and C3 has no action specified. Note also that the condition (Y, Y, N) specifies both actions A1 and A2. These multiple specified actions may be desired, or they may indicate a specification error.

TABLE 3.5 An Incomplete and Over-Specified Decision Table

C1	Y		N
C2		Y	N
C3			Y
A1		X	
A2	X		
A3			X

Figure 3.5 illustrates the use of a Karnaugh map to check a decision table for completeness and multiple specified actions. The specification is incomplete if there are any blank entries in the Karnaugh map. The specification is multiply specified if there are any multiple entries in the Karnaugh map.

3.5.1 Advantages of Decision Tables

The various advantages of decision tables include:

- Decision rules are clearly structured.
- Managers can be relieved from decision-making.
- Consistency in decision-making.
- Communication is easier between managers and analysts.
- Documentation is easily prepared, changed, or updated.
- Easy to use.
- Easier to draw or modify compared to flowcharts.
- Facilitate more compact documentation.
- Easier to follow a particular path down one column than through complex and lengthy flowcharts.

3.5.2 Disadvantages of Decision Tables

The various disadvantages of decision tables include:

- Impose an additional burden.
- Do not depict the flow.
- Not easy to translate.
- Cannot list all the alternatives.

3.6 SRS DOCUMENT

An SRS document is generated as the output of requirements analysis. Requirements analysis involves obtaining a clear and thorough understanding of the product to be developed. Thus, the SRS should be a consistent, correct, unambiguous, and complete document. The developer of the system can prepare an SRS after detailed communication with the customer. An SRS clearly defines the following:

External interfaces of the system: They identify the information that is to flow 'from and to' to the system.

Functional and non-functional requirements of the system: They are the findings of the run-time requirements.

The *functional requirements* of the system as documented in the SRS document should clearly describe each function, which the system would support along with the corresponding input and output data set.

The *non-functional requirements* deal with the characteristics of the system that cannot be expressed as functions. Examples of non-functional requirements include aspects concerning maintainability, portability, and usability. The non-functional requirements include aspects concerning maintainability, portability, and usability. The non-functional requirements may also include reliability issues, accuracy of results, human-computer interface issues, and constraints on the system implementation.

There are many ways to structure a requirements document. There is no single method that is suitable for all projects. The IEEE and U.S. Department of Defense have proposed a candidate format for representing the SRS. The general outline of the SRS document is given below:

3.6.1 Organization of an SRS Document

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 Operating Environment
 - 2.6 User Environment
 - 2.7 Assumptions and Dependencies
 - 2.8 Apportioning of Requirements
- 3. *Specific Requirements*
 - 3.1 External Interfaces
 - (i) User Interface
 - (ii) Hardware Interface
 - (iii) Software Interface
 - (iv) Communication Interface
 - 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 Attribute

- 3.6.1 Reliability
- 3.6.2 Availability
- 3.6.3 Security
- 3.6.4 Maintainability
- 3.6.5 Portability
- 3.7 Organizing the Specific Requirements
 - 3.7.1 System Mode
 - 3.7.2 User Class
 - 3.7.3 Objects
 - 3.7.4 Feature
 - 3.7.5 Stimulus
 - 3.7.6 Response
 - 3.7.7 Functional Hierarchy
- 3.8 Additional Comments
- 4. *Supporting Information*
 - 4.1 Table of Contents and Index
 - 4.2 Appendixes

3.6.2 Uses for SRS Documents

The following are a few major uses for SRS documents:

- Project managers base their plans and estimates of schedule, effort, and resources on it.
- The development team needs it to develop the product.
- The testing group needs it to generate test plans based on the described external behavior.
- The maintenance and product support staff need it to understand what the software product is supposed to do.
- The publications group writes documents, manuals, etc., from it.
- Customers rely on it to know what product they can expect.
- Training personnel can use it to help develop educational material for the software product.
- The maintenance and product support staff need it to understand what the software product is supposed to do.

3.7 IEEE STANDARDS FOR SRS DOCUMENTS

IEEE standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within the IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of the IEEE that have expressed an interest in participating in the development of the standard. Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard.

Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art.

Users are cautioned to check to determine that they have the latest edition of any IEEE Standard. Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

3.7.1 IEEE Recommended Approaches for SRS

This recommended practice describes recommended approaches for the specification of software requirements. It is based on a model in which the result of the software requirements specification process is an unambiguous and complete specification document. It should help:

1. Software customers to accurately describe what they wish to obtain;
2. Software suppliers to understand exactly what the customer wants;
3. Individuals to accomplish the following goals:
 - (i) Develop a standard software requirements specification (SRS) outline for their own organizations;
 - (ii) Define the format and content of their specific software requirements specifications;
4. Develop additional local supporting items, such as an SRS quality checklist, or an SRS writer's handbook.

3.7.2 Benefits of SRS

To the customers, suppliers, and other individuals, a good SRS should provide several specific benefits, such as the following:

1. **Establish the basis for agreement between the customers and the suppliers on what the software product is to do.**

The complete description of the functions to be performed by the software specified in the SRS will assist potential users in determining if the software specified meets their needs or how the software must be modified to meet their needs.

2. **Reduce the development effort.** The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.
3. **Provide a basis for estimating costs and schedules.** The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.
4. **Provide a baseline for validation and verification.** Organizations can develop their validation and verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.
5. **Facilitate transfer.** The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.
6. **Serves as a basis for enhancement.** Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

3.7.3 IEEE Recommended Practice for Software Requirements Specification

1. Overview
2. References
3. Definitions
4. Considerations for producing a good SRS
5. The parts of an SRS

3.8 SRS VALIDATION

It is extremely important to detect errors in the requirements document before going to other phases of system development. The major objective of SRS validation is to ensure that user requirements are complete and correctly recorded in the SRS and it is free from errors. It is also needed to check that the SRS itself is of good quality. Some of the most common types of errors in the SRS include:

1. **Omission.** Some user requirement is not included in the SRS. This error directly affects the external completeness of the system.
2. **Inconsistency.** Due to contradictions in requirements or incompatibility of state requirements.
3. **Incorrect fact.** Some facts recorded in the SRS are not correct.
4. **Ambiguity.** Some requirements have multiple meanings.

Besides improving the quality of the SRS, SRS validation should uncover and rectify all possible types of errors.

3.9 COMPONENTS OF SRS

The following requirements are used in the specification of the SRS:

1. Functional requirements
2. Performance requirements
3. Design constraints
4. External interface requirements

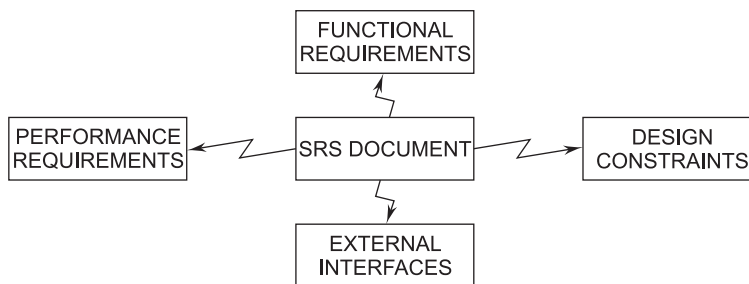


FIGURE 3.10 Components of SRS Document

1. **Functional Requirements.** Functional requirements specify which outputs should be produced from the given inputs. They describe the relationship between the input and output of the system. For each functional requirement,

a detailed description of all the data inputs and their sources, the units of measure, and the range of valid inputs must be specified.

All the operations to be performed on the input data to obtain should be specified. This includes specifying the validity checks on the input and output data, parameters affected by the operation, and equations or other logical operations that must be used to transform the inputs into corresponding outputs. For example, if there is a formula for computing the output, it should be specified. Care must be taken not to specify any algorithms that are not part of the system but that may be needed to implement the system. These decisions should be left for the designer. In addition some abnormal input, system behavior for invalid inputs, must be specified.

2. **Performance Requirements.** This part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: static and dynamic.

Static requirements are those that do not impose constraints on the execution characteristics of the system. These include requirements, such as the number of terminals to be supported, the number of simultaneous users to be supported, and the number of files that the system has to process and their sizes. These are also called capacity requirements of the system.

Dynamic requirements specify constraints on the execution behavior of the system. These typically include response time and throughput constraints on the system. Response time is the expected time for the completion of an operation under specified circumstances. Throughput is the expected number of operations that can be performed in a unit time. For example, the SRS may specify the number of transactions that must be processed per unit time, or what the response time for a particular command should be. Acceptable ranges of the different performance parameters should be specified, as well as acceptable performance for both normal and peak workload conditions.

3. **Design Constraints.** There are a number of factors in the client's environment that may restrict the choices of a designer. Such factors include standards that must be followed, resource limits, operating environment, reliability and security requirements, and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints, including:

- (i) *Standards Compliance.* This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit-tracing requirements, which require certain kinds of changes, or operations that must be recorded in an audit file.

- (ii) *Hardware Limitations.* The software may have to operate on some existing or predetermined hardware, thus imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating system available on the system, languages supported, and limits on primary and secondary storage.
 - (iii) *Reliability and Fault Tolerance.* Fault-tolerance requirements can place a major constraint on how the system is to be designed. Fault-tolerance requirements often make the system more complex and expensive. Recovery requirements must specify what the system should do if some fault occurs. Recovery requirements are often an integral part of the design constraints.
 - (iv) *Security.* Security requirements are particularly significant in defense systems and many database systems. Security requirements place restrictions on the use of certain commands, control access to data, provide different kinds of access requirements for different people, require the use of passwords and cryptography techniques, and maintain a log of activities in the system. Given the current security needs even of common systems, they may also require proper assessment of security threats, proper programming techniques, and use of tools to detect flaws, such as buffer overflow.
4. **External Interface Requirements.** All the interactions of the software with people, hardware, and other software should be clearly specified. For the user interface, the characteristics of each user interface of the software product should be specified. User interface is becoming increasingly important and must be given proper attention. A preliminary user manual should be created with all user commands, screen formats, an explanation of how the system will appear to the user, and feedback and error messages. Like other specifications these requirements should be precise and verifiable. So, a statement like “the system should be user friendly” should be avoided and statements like “commands should be no longer than six characters” or “commands names should reflect the function they perform” should be used.

For hardware interface requirements, the SRS should specify the logical characteristics of each interface between the software product and the hardware components. If the software is to execute on existing hardware or on predetermined hardware, all the characteristics of the hardware, including memory restrictions, should be specified. In addition, the current use and load characteristics of the hardware should be given.

The interface requirement should specify the interface with other software the system will use or that will use the system. This includes the interface with the operating system and other applications. The message content and format of each interface should be specified.

3.10 CHARACTERISTICS OF SRS

A good SRS document has certain characteristics that must be present. The characteristics are:

1. **Correctness.** An SRS is correct if every requirement included in the SRS represents something required in the final system.
2. **Completeness.** An SRS is complete when it is documented after:
 - (i) The involvement of all types of concerned personnel.
 - (ii) Focusing on all problems, goals, and objectives, and not only on functions and features.
 - (iii) Correct definition of scope and boundaries of the software and system.
3. **Unambiguous.** An SRS is unambiguous if and only if every requirement stated has one and only one interpretation. Requirements are often written in a natural language. The SRS writer has to be especially careful to ensure that there are no ambiguities. One way to avoid ambiguities is to use some formal requirements specification language. The major disadvantage of using formal languages is the large effort required to write an SRS, the high cost of doing so, and the increased difficulty of reading and understanding formally stated requirements (particularly by the users and clients).
4. **Verifiable.** An SRS is verifiable if and only if there exists some cost-effective process that can check whether the final product meets the requirements.
5. **Modifiable.** An SRS is modifiable if its structure and style are such that any necessary change can be made easily while preserving completeness and consistency. The presence of redundancy is a major hindrance to modifiability, as it can easily lead to errors. For example, assume that a requirement is stated in two places and that the requirement later needs to be changed. If only one occurrence of the requirement is modified, the resulting SRS will be inconsistent.
6. **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:
 - (i) *Backward traceability.* This depends upon each requirement explicitly referencing its source in earlier documents.
 - (ii) *Forward traceability.* This depends upon each requirement in the SRS having a unique name or reference number.
7. **Consistency.** Consistency in the SRS is essential to achieve correct results across the system. This is achieved by:
 - (i) The use of standard terms and definitions.
 - (ii) The consistent application of business rules in all functionality.
 - (iii) The use of a data dictionary.

- (iv) The lack of consistency results in an incorrect SRS and failure in deliverables to customer.
- 8. **Testability.** An SRS should be written in such a way that it is possible to create a test plan to confirm whether specifications can be met and requirements can be delivered. This is achieved by:
 - (i) Considering functional and non-functional requirements.
 - (ii) Determining features and facilities required for each requirement.
 - (iii) Ensuring that 'users' and 'stakeholders' freeze the requirement.
- 9. **Clarity.** An SRS is clear when it has a single interpretation for the author (analysis), the user, the end user, the stakeholder, the developer, the tester, and the customer. This is possible if the language of the SRS is unambiguous. Clarity can be ascertained after reviewing the SRS by a third party. It can be enhanced if the SRS includes diagrams, models, and charts.
- 10. **Feasibility.** RDD-SRS needs to be confirmed on technical and operational feasibility. The SRS often assumes the use of technology and tools based on the information given by their vendors. It needs to be confirmed whether the technology is capable enough to deliver what is expected in the SRS. The operational feasibility must be checked through environment checking. It is assumed that sources of data, user capability, system culture, work culture, and other such aspects satisfy the expectation of the developer. These must be confirmed before development launch.

3.11 ENTITY-RELATIONSHIP DIAGRAM

The Entity-Relationship (E-R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database. It is a data-oriented model of a system, whereas a DFD is a process-oriented model. It has three main components: data entities, their relationships, and their associated attributes.

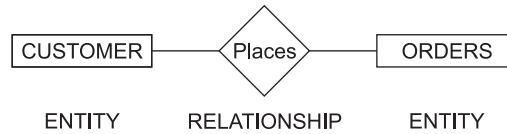
1. **Entity.** It is the most elementary process of an organization about which data is to be maintained. Every entity has a unique identity, which distinguishes it from other entities. An entity type is the description of all entities to which a common definition and common relationships and attributes apply. It is represented by a rectangular box with the name of the entity written inside. For example, invoice object has various elements, such as invoice number, date, quantity, discount, total-price, etc.

INVOICE

CUSTOMER

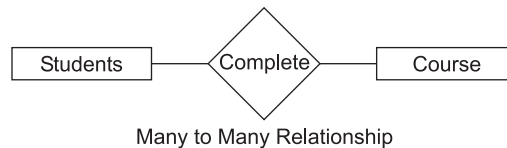
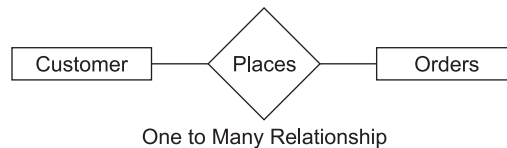
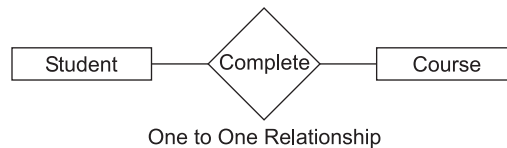
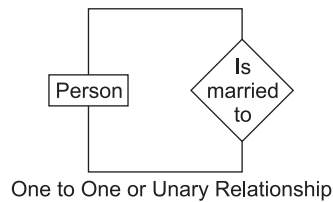
ENTITY TYPES

2. **Relationship.** Entities are connected to each other by relationships. It indicates how two entities are associated. A diamond notation with the name of the relationship is represented as written inside.



The number of entity types that participate in the relationship is called the degree of the relationship (e.g., customer places order).

The above two relationships have degree two because they involve two entity types—customer and orders. The three most common relationships in the E-R diagram are unary, i.e., degree one, binary, i.e., degree two, and ternary, i.e., degree three.



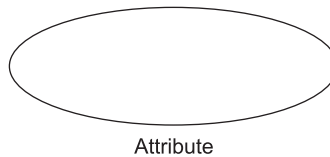
Binary Relationship

Cardinality and optionally: The cardinality represents the relationship between two entities. Consider the one to many relationships between two entities, class and student. Here, the cardinality of a relationship is the number of instances of entity student that can be associated with each instance of entity class. This is shown in Figure 3.11.



FIGURE 3.11 One to Many Cardinality Relationship

3. **Attribute.** 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. It is represented by an oval-shaped box with the name of the attribute written inside it. The notation for attribute is



3.11.1 Types of Attributes

1. **Simple Attribute.** There is no need to sub-divide a simple attribute into component attributes. For example, if there is no need to refer to the individual components of an address (zip, street, and so on), then the whole address is designated as a simple attribute.
2. **Composite Attribute.** On the other hand, a composite attribute can be divided into sub-parts. For example, an attribute name could be structured as a composite attribute consisting of first_name, middle_initial, and last_name.
3. **Single Valued Attribute.** The attribute in our examples all have a single value for a particular entity. For instance, the loan_number attribute for a specific loan entity refers to only one loan number. Such attributes are said to be single valued.
4. **Multivalued Attribute.** There may be instances where an attribute has a set of values for a specific entity. Consider an employee entity set with the attribute phone_number. An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be multi-valued.
5. **Derived Attribute.** The value for this type of attribute can be derived from the values of other related attributes or entities. For instance, let us say that the customer entity set has an attribute loans_held, which represents how many loans a customer has from the bank. We can derive the value for this attribute by counting the number of loan entities associated with that customer.

The minimum cardinality of a relationship is the minimum number of instances of the second entity (student, in this case) with each instance of the first entity (class, in this case).

In a situation where there can be no instance of the second entity, then it is called an optional relationship. For example, if a college does not offer a particular course then it will be considered optional with respect to the relationship 'offers.' This relationship is shown in Figure 3.12.



FIGURE 3.12 Minimum Cardinality Relationship

When the minimum cardinality of a relationship is one, then the second entity is called a mandatory participant in the relationship. The maximum cardinality is the maximum number of instances of the second entity. The modified E-R diagram is shown in Figure 3.13.

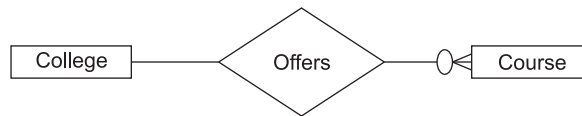


FIGURE 3.13 Modified E-R Diagram Representing Cardinalities

The relationship cardinalities are shown in Figure 3.14.

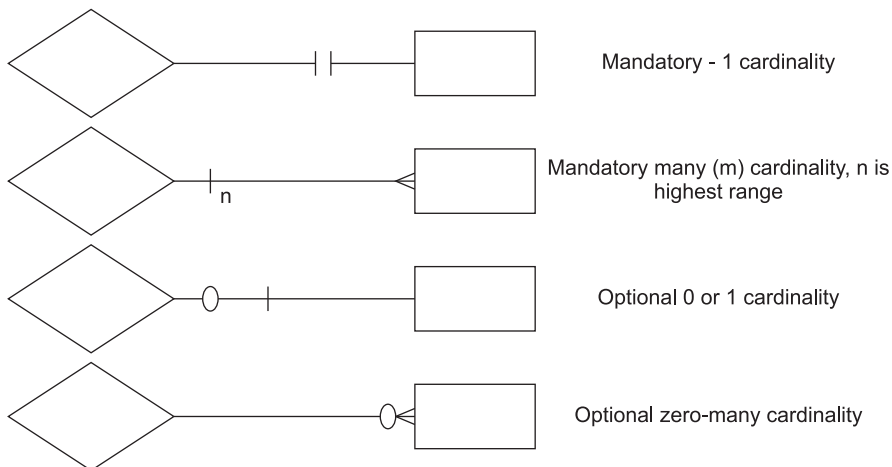


FIGURE 3.14 Relationship Cardinalities

EXERCISES

1. What does the term requirements mean? Explain the process of determining the requirements for a software-based system.
2. Discuss the significance and use of requirement engineering. What are the problems in the formulation of requirements?
3. What are the crucial process steps of requirement engineering? Discuss with the help of a diagram.
4. Describe the various steps of requirements engineering. Is it essential to follow these steps?
5. Explain the importance of requirements. How many types of requirements are possible and why?
6. What is requirements elicitation? Discuss any two techniques in detail.
7. List the requirements elicitation techniques. Which one is most popular and why?
8. Explain requirements elicitation and the analysis process model with the help of a suitable diagram.
9. What is requirements management? What are the steps of planning in requirement management?
10. What is the traceability of a requirement? Why is traceability important?
11. Explain why a many to many relationship is to be modeled as an associative entity.
12. Explain the relationship between minimum cardinality and optional and mandatory participation.
13. Define:
 - (i) Data-flow diagram
 - (ii) Decision table
 - (iii) E-R diagram
14. What are the similarities between data-flow and E-R diagrams?
15. Draw the E-R diagram for a hotel reception desk management.
16. Draw a DFD for borrowing a book in a library if: "A borrower can borrow a book if it is available, otherwise he can reserve the book if he so wishes. He can borrow a maximum of three books."
17. What is an SRS? What are the components of an SRS?
18. Discuss the characteristics of a good SRS document.
19. What is SRS validation?
20. List the seven desirable characteristics of a good Software Requirements Specification (SRS) document.
21. Discuss the organization of an SRS. List some important reasons for this organization.
22. Define information modeling.
23. What is a DFD? Explain some of the symbols used to draw a DFD.
24. State some of the advantages and disadvantages of data-flow diagrams.
25. Explain the IEEE standards for an SRS.
26. According to IEEE standards what are several specific benefits a good SRS should provide?

SOFTWARE RELIABILITY AND QUALITY ASSURANCE

4.1 VERIFICATION AND VALIDATION

Verification and validation (V and V) is the name given to the checking and analysis process that ensures that software conforms to its specifications and meets the needs of the customers who are paying for that software. Verification and validation is a whole life-cycle process. It starts with requirements reviews and continues through design reviews and code inspection to product testing. There should be V and V activities at each stage of the software-development process. These activities ensure that the results of process activities are as specified.

Verification and validation is not the same thing although they are easily confused. The difference between them is succinctly expressed by Boehm (1979):

- ‘**Validation:** Are we building the right product?’
- ‘**Verification:** Are we building the product right?’

These definitions tell us that the role of verification involves checking that the software conforms to its specifications. You should check that the system meets its

specified functional and non-functional requirements. Validation, however, is a more general process; you should ensure that the software meets the expectations of the customer. It goes beyond checking conformance of the system to its specifications to showing that the software does what the customer expects.

Within the V and V process, *two techniques* of system checking and analysis may be used.

1. **Software inspections.** Software inspection analyzes and checks system representations, such as the requirements document, design diagrams, and the program source code. They may be applied at all stages of the process. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Software inspections and automated analyzes are static V and V techniques as they do not require the system to be executed.
2. **Software testing.** Software testing involves executing an implementation of the software with test data and examining the outputs of the software and its operational behavior to check that it is performing as required. Testing is a dynamic technique of verification and validation because it works with an executable representation of the system.

4.1.1 Verification

Verification is the process of determining whether the output of one phase of software development confirms to that of its previous phase.

OR

Verification involves checking of functional and non-functional requirements to ensure that the software confirms to its specifications.

4.1.2 Validation

Validation is the process of determining whether a fully developed system confirms to its requirement specifications.

OR

Validation is an analysis process that is done after checking conformance of the system to its specifications.

Thus, the goal of the verification and validation process is to establish confidence in the customer that the software system is 'fit for the customer.' It doesn't mean that the software system is free from errors.

4.2 SOFTWARE QUALITY ASSURANCE

The aim of the Software Quality Assurance (SQA) process is to develop a high-quality software product. *Software Quality Assurance is a set of activities designed to evaluate the process by which software is developed and/or maintained.*

Quality assurance is a *planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements (IEEE83).*

The purpose of a software quality assurance group is to provide assurance that the procedures, tools, and techniques used during product development and modification are adequate to provide the desired level of confidence in the work products.

The process of the SQA:

1. Defines the requirements for software controlled system fault/failure detection, isolation, and recovery;
2. Reviews the software-development processes and products for software-error prevention and/or controlled change to reduced functionality states; and
3. Defines the process for measuring and analyzing defects as well as reliability and maintainability factors.

4.2.1 SQA Objectives

The various objectives of SQA are as follows:

- Quality management approach.
- Measurement and reporting mechanisms.
- Effective software-engineering technology.
- A procedure to assure compliance with software-development standards where applicable.
- A multi-testing strategy is drawn.
- Formal technical reviews that are applied throughout the software process.

4.2.2 SQA Goals

The major goals of SQA are as follows:

- SQA activities are planned.
- Non-compliance issues that cannot be resolved within the software project are addressed by senior management.

- Adherence of software products and activities to the applicable standards, procedures, and requirements is verified objectively.
- Affected groups and individuals are informed of SQA activities and results.

4.2.3 SQA Plan

An SQA plan defines the quality processes and procedures that should be used. This involves selecting and instantiating standards for products and processes and defining the required quality attributes of the system.

The SQA plan provides a roadmap for instituting software quality assurance. Developed by the SQA group (or the software team if a SQA group does not exist), the plan serves as a template for SQA activities that are instituted for each software project.

The quality plan should select those organizational standards that are appropriate to a particular product and development process. New standards may have to be defined if the project uses new methods and tools.

An outline structure for a quality plan includes:

1. *Product introduction*: A description of the product, its intended markets, and the quality expectations for the product.
2. *Product plans*: The critical release dates and responsibilities for the product along with plans for distribution and product servicing.
3. *Process descriptions*: The development and service processes that should be used for product development and management.
4. *Quality goals*: The quality goals and plans for the product including an identification and justification of critical product quality attributes.
5. *Risks and risk management*: The key risks that might affect product quality and the actions to address these risks.

Preparation of a Software Quality Assurance Plan for each software project is a primary responsibility of the software quality assurance group. Topics in a Software Quality Assurance Plan include:

- Purpose-scope of plan;
- List of references to other documents;
- Management, including organization, tasks, and responsibilities;
- Documentation to be produced;
- Standards, practices, and conventions;
- Reviews and audits;
- Testing;

- Problem reporting and corrective action;
- Tools, techniques, and methodologies;
- Code, media, and supplier control;
- Records collection, maintenance, and retention;
- Training;
- Risk management—the methods of risk management that are to be used.

4.3 SOFTWARE QUALITY

We examine the qualities that are pertinent to software products and software production processes. These qualities will become our goals in the practice of software engineering.

The basic goal of software engineering is to produce quality software. Software quality is a broad and important field of software engineering addressed by several standardization bodies, such as ISO, IEEE, ANSI, etc.

4.3.1 Definition of Software Quality

Software quality is the:

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

The above definition emphasizes three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned. If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

4.3.2 Classification of Software Qualities

There are many desirable software qualities. Some of these apply both to the product and to the process used to produce the product. The user wants the software products to be reliable, efficient, and easy to use. The producer of the software wants it to be verifiable, maintainable, portable, and extensible. The

manager of the software project wants the process of software development to be productive and easy to control.

In this section, we consider two different classifications of software-related qualities: internal versus external and product versus process.

External versus Internal Qualities

We can divide software qualities into external and internal qualities. The external qualities are visible to the users of the system: the internal qualities are those that concern the developers of the system. In general, users of the software only care about the external qualities, but it is the internal qualities, which deal largely with the structure of the software, that help developers achieve the external qualities. For example, the internal quality of verifiability is necessary for achieving the external quality of reliability. In many cases, however, the qualities are related closely and the distinction between internal and external is not sharp.

Product and Process Qualities

We use a process to produce the software product. We can also attribute some qualities to the process, although process qualities often are closely related to product qualities. For example, if the process requires careful planning of system test data before any design and development of the system starts, products reliability will increase. Some qualities, such as efficiency, apply both to the product and to the process.

It is interesting to examine the word product here. It usually refers to what is delivered to the customer. Even though this is an acceptable definition from the customer's perspective, it is not adequate for the developer who requires a general definition of a software product that encompasses not only the object code and the user manual that are delivered to the customer but also the requirements, design, source code, test data, etc. In fact, it is possible to deliver different subsets of the same product to different customers.

4.3.3 Software Quality Attributes

Software quality is comprised of six main attributes (called characteristics) as shown in Figure 4.1. These six attributes have detailed characteristics which are considered the basic ones and which can and should be measured using suitable metrics. At the top level, for software products, these attributes can be defined as follows:

1. **Functionality:** The capability to provide functions which meet stated and implied needs when the software is used.

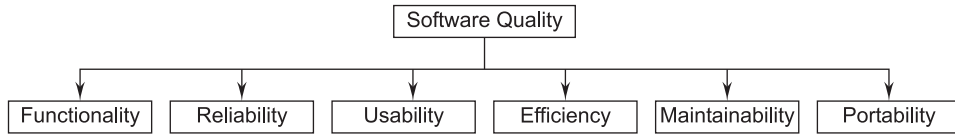


FIGURE 4.1 Software Quality Attributes

2. **Reliability:** The capability to maintain a specified level of performance.
3. **Usability:** The capability to be understood, learned, and used.
4. **Efficiency:** The capability to provide appropriate performance relative to the amount of resources used.
5. **Maintainability:** The capability to be modified for purposes of making corrections, improvements, or adaptation.
6. **Portability:** The capability to be adapted for different specified environments without applying actions or means other than those provided for this purpose in the product.

4.3.4 McCall's Quality Factors

McCall, Richards, and Walters [MCC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 4.2, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments.

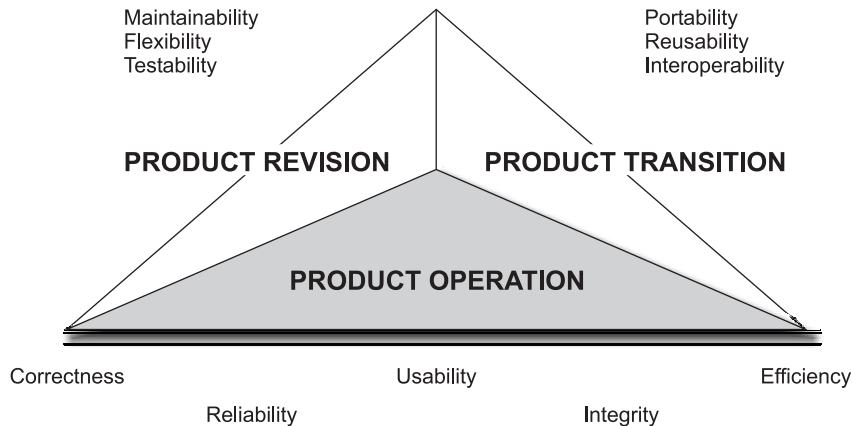


FIGURE 4.2 McCall's Software Quality Factors

One attempt to identify specific product qualities that are appropriate to software has been that of James A. McCall. He grouped software qualities into three sets of quality factors:

- Product operation qualities;
- Product revision qualities; and
- Product transition qualities.

The definitions below are those given by McCall, but the reader may come across others. These are not all inclusive: sometimes other qualities might be of interest.

Product Operation Quality Factors

- *Correctness*: The extent to which a program satisfies its specifications and fulfills the user's objectives.
- *Reliability*: The extent to which a program can be expected to perform its intended function with required precision.
- *Efficiency*: The amount of computer resources required by the software.
- *Integrity*: The extent to which access to software or data by unauthorized persons can be controlled.
- *Usability*: The effort required for learning, operating, preparing input, and interpreting output.

Product Revision Quality Factors

- *Maintainability*: The effort required to locate and fix an error in an operational program.
- *Testability*: The effort required to test a program to ensure it performs its intended function.
- *Flexibility*: The efforts required to modify an operational program.

Product Transition Quality Factors

- *Portability*: The effort required for transferring a program from one hardware configuration and software system environment to another.
- *Reusability*: The extent to which a program can be used in other applications.
- *Interoperability*: The efforts required to couple one system to another.

4.3.5 Software Quality Criteria

The software quality criteria of various quality factors are depicted in Table 4.1.

TABLE 4.1 Software Quality Criteria

Quality Factor	Software Quality Criteria
Correctness	Traceability, consistency, completeness
Reliability	Error tolerance, consistency, accuracy, simplicity
Efficiency	Execution efficiency, storage efficiency
Integrity	Access control, access audit
Usability	Operability, training, communicativeness, input/output volume, input/output area
Maintainability	Consistency, simplicity, conciseness, modularity, self-descriptiveness
Testability	Simplicity, modularity, instrumentation, self-descriptiveness
Flexibility	Modularity, generality, expandability, self-descriptiveness
Portability	Modularity, self-descriptiveness, machine independence, software system independence
Reusability	Generality, modularity, software system independence, machine independence, self-descriptiveness
Interoperability	Modularity, communications commonality, data commonality

4.3.6 Representative Qualities

In this section, we present the most important qualities of software products and processes.

1. **Correctness.** A program is functionally correct if it behaves according to the specification of the functions it should provide (called functional requirements specifications). It is common simply to use the term “correct” rather than “functionally correct”; similarly, in this context, the term “specifications” implies “functional requirements specification.” We will follow this convention when the context is clear.

The definition of correctness assumes that a specification of the system is available and that it is possible to determine unambiguously whether or not a program meets the specifications. With most current software systems, no such specification exists. If a specification does exist, it is usually written in an informal style using natural language.

2. **Reliability.** Informally, software is reliable if the user can depend on it. The specialized literature on software reliability defines reliability in terms of statistical behavior—the probability that the software will operate as expected over a specified time interval.

Figure 4.3 illustrates the relationship between reliability and correctness. This figure shows that the set of all reliable programs includes the set of correct programs, but not vice versa.

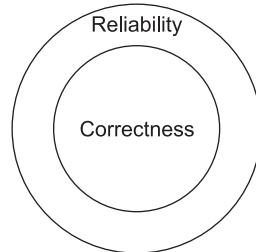


FIGURE 4.3 Relationship Between Correctness and Reliability in the Ideal Case

3. **Robustness.** A program is robust if it behaves “reasonably,” even in circumstances that were not anticipated in the requirements specification—for example, when it encounters incorrect input data or some hardware malfunction (say, disk crash).

Obviously, robustness is a difficult-to-define quality; after all, if we could state precisely what we should do to make an application robust, we would be able to specify its “reasonable” behavior completely. The robustness would become equivalent to correctness (or reliability in the sense of Figure 4.3).

4. **Performance.** Performance is important because it affects the usability of the system. If a software system is too slow, it reduces the productivity of the users, possibly to the point of not meeting their needs. If a software system uses too much disk space, it may be too expensive to run. If a software system uses too much memory, it may affect the other applications that are run on the same system, or it may run slowly while the operating system tries to balance the memory usage of the different applications. Performance is also important because it affects the scalability of a software system.
- 5 **Verifiability.** A software system is verifiable if its properties can be verified easily. For example, it is important to be able to verify the correctness or the performance of a software system.

Verifiability is usually an internal quality, although it sometimes becomes an external quality also. For example, in many security-critical applications, the customer requires the verifiability of certain properties. The highest level of the security standard for a trusted computer system requires the verifiability of the operating system kernel.

6. **Repairability.** A software system is repairable if it allows the correction of its defects with a limited amount of work. In many engineering products, repairability is a major design goal. For example, automobile engines are built with the parts that are most likely to fail as the most accessible. In computer hardware engineering, there is a subspecialty called Repairability, Availability, and Serviceability (RAS).
7. **Evolvability.** Like other engineering products, software products are modified over time to provide new functions or to change existing functions. Indeed, the fact that software is so malleable makes modifications extremely easy to apply to an implementation.

8. **Understandability.** Some software systems are easier to understand than others. Of course, some tasks are inherently more complex than others.

Given tasks of inherently similar difficulty, we can follow certain guidelines to produce more understandable designs and to write more understandable programs. For example, abstraction and modularity enhance a system's understandability.

9. **Interoperability.** "Interoperability" refers to the ability of a system to coexist and cooperate with other systems.

With interoperability, a vendor can produce different products and allow the user to combine them if necessary. This makes it easier for the vendor to produce the products, and it gives the user more freedom in exactly what functions to pay for and to combine. Interoperability can be achieved through standardization of interfaces.

10. **Productivity.** Productivity is a quality of the software-production process; it measures the efficiency of the process and, as we said before, is the performance quality applied to the process. An efficient process results in faster delivery of the product.

Productivity offers many trade-offs in the choice of a process. For example, a process that requires specialization of individual team members may lead to productivity in producing a certain product, but not in producing a variety of products. Software reuse is a technique that leads to the overall productivity of an organization that is involved in developing many products, but developing reusable modules is harder than developing modules for one's own use, thus reducing the productivity of the group that is developing reusable modules as part of their product development.

11. **Timeliness.** Timeliness is a process-related quality that refers to the ability to deliver a product on time.

Timeliness requires careful scheduling, accurate estimation of work, and clearly specified and verifiable milestones.

12. **Visibility.** A software-development process is visible if all of its steps and its current status are documented clearly. Another term used to characterize this property is transparency. The idea is that the steps and the status of the project are available and easily accessible for external examination.

4.3.7 Importance of Software Quality

We would expect quality to be a concern of all producers of goods and services. However, the special characteristics of software, and in particular, its intangibility and complexity, make special demands.

1. **Increasing Criticality of Software.** The final customer or user is naturally anxious about the general quality of software, especially its reliability. This is increasingly the case as organizations become more dependent on their computer systems and software is used more and more in areas which are safety critical; for example, to control aircraft.
2. **The Intangibility of Software.** This makes it difficult to know whether a particular task in a project has been completed satisfactorily. The results of these tasks can be made tangible by demanding that the developers produce 'deliverables' that can be examined for quality.
3. **Accumulating Errors During Software Development.** As computer system development is made up of a number of steps where the output from one step is the input to the next, the errors in the earlier deliverables will be added to those in the later steps leading to an accumulating detrimental effect, and generally, the later in a project that an error is found the more expensive it will be to fix. In addition, because the number of errors in the system is unknown the debugging phases of a project are particularly difficult to control.

For these reasons quality management is an essential part of effective overall project management.

4.4 CAPABILITY MATURITY MODEL (SEI-CMM)

SEI-CMM stands for Software Engineering Institute Capability Maturity Model. The CMM was developed by the Software Engineering Institute (SEI) of the Carnegie Mellon University in the USA, which is engaged in a long-term program of software-process improvement.

It is a model that helps in judging the maturity of a software process of an organization. It also helps to identify the main process that will help in increasing the maturity of these processes. It has become a standard for assessing and improving software processes.

SEI-CMM can be used in two ways: capability evaluation and software-process assessment. Capability evaluation and software-process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software-process capability of an organization. The results of capability evaluation indicate the likely contractor performance if the contractor is awarded a work. Therefore, the results of the software-process capability assessment can be used to select a contractor. On the other hand, software-process assessment is used by an organization with the objective to improve its process capability. Thus, this type of assessment is for purely internal use.

Five levels of process maturing have been proposed for the software industry by SEI-CMM, which indicate the sophistication and quality of their software production practices. These levels are defined as follows:

1. **Level 1 (Initial).** The software process is adhoc, and even chaotic at time. The organization whose success depends on individual effort and whose processes are not defined and documented come under this level.
Organizations at this level can benefit most by improving project management, quality assurance, and change control.
2. **Level 2 (Repeatable).** Here basic management practices, such as tracking costs, schedules, and functionality are established but not the procedures for doing it. Here, the efforts done previously for the success of a project may repeat.
Some of the characteristics of a process at this level are: project commitments are realistic and based on past experience with similar projects, costs and schedules are tracked and problems resolved when they arise, formal configuration control mechanisms are in place, and software project standards are defined and followed.
3. **Level 3 (Defined).** The organization-wide software process includes management and engineering procedures. These procedures are well-defined, documented, standardized, and integrated. All projects make use of the documented and approved version of the organization process for software development and maintenance. But the process and practices are not analyzed quantitatively. In this process both the development and management processes are formal. ISO 9000 aims at achieving this level.

4. **Level 4 (Managed).** At this level, the focus is on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as the average defect correction time, productivity, the average number of defects found per hour of inspection, the average number of failures detected during testing per LOC, and so forth. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than to improve the process.

Software processes and products are quantitatively understood, measured, and controlled using detailed procedures.

5. **Level 5 (Optimized).** At this level, an organization is committed to continuous process improvement. Process improvement is budgeted and planned and is an integral part of the organization's process. The organizations have the means to identify weaknesses and strengthen the process proactively, with the goal of preventing the occurrence of defects. Best software-engineering and management practices are used throughout the organization.

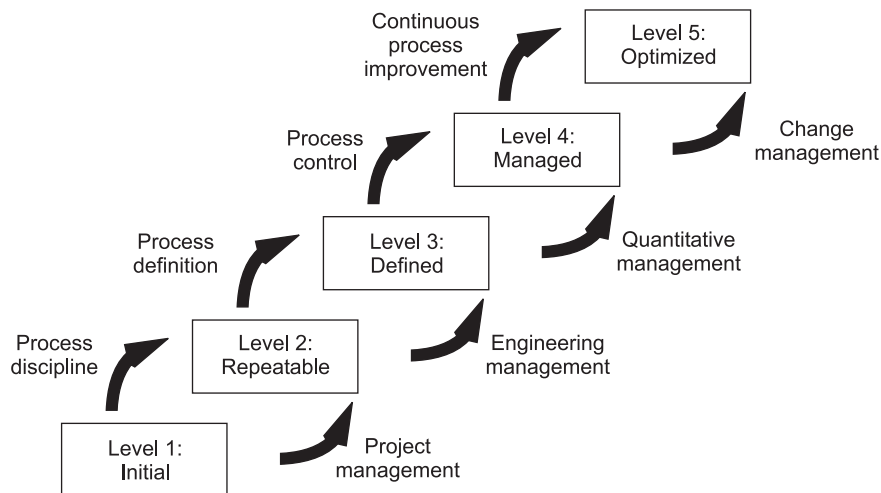


FIGURE 4.4 Software Engineering Institute Levels of Maturity

Except for Level 1, each maturity level is characterized by several Key Process Areas (KPAs) that indicate the areas an organization should focus on to improve its software process at the next level.

This is shown in Table 4.2:

TABLE 4.2

MM level	Characteristics	Focus	Key Process Areas
1 (Initial)	Chaotic, unrepeatable, high risk of non-performance	Competent people	Not applicable
2 (Repeatable)	Methodical in umbrella activities and processes. Performance is repeated but not improved	Project management	Configuration management, quality assurance, sub-contract management, project tracking and oversight, project planning
3 (Defined)	Improved performance in cost, schedule quality, and risk management	Definition of processes	Peer reviews, inter-group coordination, software product engineering, training program, integrated software management, and organization process definition
4 (Managed)	Learns from project experience, and continuous performance improvement in all key areas of the project	Product and process quality	Quality management, process measurement, and analysis
5 (Optimized)	All-around performance, improvement through learning. High performance in all quality attributes and risk management through RMMM	Continuous process improvement	Process change management, technology innovation, and defect prevention

4.5 INTERNATIONAL STANDARD ORGANIZATION (ISO)

4.5.1 Introduction to the ISO

The International Organization for Standardization (ISO) is a group of worldwide federations of national standards bodies from some 100 countries. The ISO is a non-governmental organization established in 1947.

The ISO-9000 standard specifies quality-assurance elements in generic terms, which can be applied to any business, regardless of the product or services being offered. In order to register for one of the quality-assurance system models contained in the ISO-9000, third-party auditors examine an organization's quality system and operations for compliance to the standard and for effective operations. Upon successful audit, the organization receives a certificate from a registered body represented by the auditors. Thereafter, semi-annual audits ensure conformance to the standard.

The ISO-9000 standard views an organization as a set of interrelated processes. In order to pass the criteria for ISO-9000 compliance, the processes must address the identified areas, and document and practice them. When a process is documented, it is better understood, controlled, and improved. However, the ISO-9000 standard does not specify how an organization should implement its quality system. Therefore, the biggest challenge is to design and implement a quality-assurance system that meets the standard and gels well with the products/services of the organization.

The ISO-9001 is a quality-assurance standard that is specific to software engineering. It specifies 20 standards with which an organization must comply for an effective implementation of the quality assurance system.

The ISO-9000 series of standards is a set of documents dealing with quality systems that can be used for quality assurance purposes. The ISO-9000 series is not just a 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.

4.5.2 ISO-9000 Mission

The mission of the ISO is to promote the development of standardization and related activities to facilitate the international exchange of goods and services, and to develop cooperation in the spheres of intellectual, scientific, technological, and economic activity. The ISO published its ISO-9000 standard in 1988. ISO-9000 consists of three standards for external quality assurance. These are ISO-9001, ISO-9002, and ISO-9003.

1. **ISO-9001.** The ISO-9001 is an international quality-management system. The ISO-9001 is mainly related to the software industry. It lays down the standards for designing, developing, servicing, and producing a standard quality of goods. It is also applicable to most software-development organizations.
2. **ISO-9002.** The ISO-9002 is basically related to manufacturing only and is silent on designing issues. Examples of this category of industries include steel- and car-manufacturing industries that buy the product and plant

designs from external sources and are involved in only manufacturing those products. Therefore, the ISO-9002 is not applicable to software-development organizations.

3. **ISO-9003.** The ISO-9003 standard applies to the service industry. The organizations who are involved in only installation of products, services, and testing of products are eligible for the ISO-9003 certification.

4.5.3 Why is ISO certification required by the software industry?

There are several reasons why the software industry must get an ISO certification. Some of the important reasons include:

- It is sign of customer confidence. This certification has become a standard for international bidding.
- It is a motivating factor for business organizations.
- It makes processes more focused, efficient, and cost-effective.
- It helps in designing high-quality repeatable software products.
- It highlights weaknesses and suggests corrective measures for improvements.
- It facilitates the development of optimal processes and total quality measurement.
- It emphasizes the need for proper documentation.

4.5.4 How does an organization obtain ISO-9000 certification?

An organization intending to obtain ISO-9000 certification applies to a ISO-9000 registrar for registration. The ISO-9000 registration process consists of the following stages:

1. **Application.** Once an organization decides to go for ISO-9000 certification, it applies to a registrar for registration.
2. **Pre-assessment.** During this stage, the registrar makes a rough assessment of the organization.
3. **Document Review and Adequacy of Audit.** During this stage, the registrar reviews the documents submitted by the organization and makes suggestions for possible improvements.
4. **Compliance Audit.** During this stage, the registrar checks whether the suggestions made by it during review have been complied by the organization or not.
5. **Registration.** The registrar awards the ISO-9000 certificate after successful completion of all previous phases.

6. *Continued Surveillance.* The registrar continues to monitor the organization, though only periodically.

4.5.5 Benefits of ISO-9000 Certification

Benefits of ISO-9000 certification include:

1. *Continuous Improvement.* Business ISO-9000 certification forces an organization to focus on “how they do business.” Each procedure and work instruction must be documented and thus becomes the springboard for continuous improvement.
2. *Eliminate Variation.* Documented processes are the basis for repetition and help eliminate variation within the process. As variation is eliminated, efficiency improves. As efficiency improves, the cost of quality is reduced.
3. *Higher Real and Perceived Quality*
 - (i) With the development of solid corrective and preventative measures, permanent, company-wide solutions to quality problems are found.
 - (ii) This results in higher quality.
4. *Boost Employee Morale.* Employee morale is increased as they are asked to take control of their processes and document their work processes.
5. *Improved Customer Satisfaction.* Customer satisfaction, and more importantly customer loyalty, grows as a company transforms from a reactive organization to a proactive, preventative organization. It becomes a company people want to do business with.
6. *Increased Employee Participation.* Reduced problems resulting from increased employee participation, involvement, awareness, and systematic employee training.
7. *Better Product and Services.* Better products and services result from continuous improvement processes.
8. *Greater Quality Assurance.* Fosters the understanding that quality, in and of itself, is not limited to a quality department but is everyone’s responsibility.
9. *Improved Profit Levels.* Improved profit levels result as productivity improves and rework costs are reduced.
10. *Improved Communication.* Improved communications both internally and externally which improves quality, efficiency, on-time delivery, and customer/supplier relations.
11. *Reduced Cost.* ISO standards result in reduced costs of the product.
12. *Competitive Edge.* By offering higher customer services, ISO-9000 standards add a competitive edge.

4.5.6 Limitations of ISO-9000 Certification

The limitations of ISO-9000 certification include:

- ISO-9000 does not provide any guidelines for defining an appropriate process.
- ISO-9000 certification process is not foolproof and no international accrediting agency exists.
- ISO-9000 does not automatically lead to continuous process improvement, i.e., it does not automatically lead to TQM.

4.5.7 Uses of ISO

ISO certification is used as a reference model for a contract between the external independent parties. It must be remembered that ISO certification does not guarantee a high-quality product. It focuses mainly on the processes. ISO is now being used as the standard and basis of evaluation for international bidding. The ISO-9000 specifies a set of guidelines for repeatable organizations. A repeatable organization is one where the production process is person-independent.

4.5.8 Salient Features of ISO-9001 Requirements

The salient features of ISO-9001 requirements include:

- All documents concerned with the development of a software product should be properly managed, authorized, and controlled.
- Proper plans should be prepared and then progress against these plans should be monitored.
- Important documents should be independently checked and reviewed for effectiveness and correctness.
- The product should be tested against its specifications.

4.5.9 ISO-9126

Over the years, various lists of software quality characteristics have been put forward, such as those of McCall, described previously and of Boehm. A difficulty has been the lack of agreed definitions of the qualities of good software. The term 'maintainability' has been used, for example, to refer to the ease with which an error can be located and corrected in a piece of software, and also in a wider sense to include the ease of making any changes. For some, 'robustness' has meant the software's tolerance of incorrect input, while for others it has meant the ability to change program code without introducing unexpected errors. The ISO-9126 standard was published in 1991 to tackle the question of the definition of software

quality. This 13-page document was designed as a foundation upon which further, more detailed, standards could be built.

ISO-9126 identifies six software quality characteristics:

- *Functionality*, which covers the functions that a software product provides to satisfy user needs;
- *Reliability*, which relates to the capability of the software to maintain its level of performance;
- *Usability*, which relates to the effort needed to use the software;
- *Efficiency*, which relates to the physical resources used when the software is executed;
- *Maintainability*, which relates to the effort needed to make changes to the software;
- *Portability*, which relates to the ability of the software to be transferred to a different environment.

The ISO-9126 has sub-characteristics for each of the primary characteristics. It is indicative of the difficulties of gaining widespread agreement that these sub-characteristics are outside the main standards and are provided for 'information only.' They are useful as they clarify what is meant by the main characteristics.

Characteristic	Sub-characteristics
Functionality	Suitability Accuracy Interoperability Compliance Security

Compliance refers to the degree to which the software adheres to application-related standards or legal requirements. Typically these are auditing requirements.

Interoperability and security are good illustrations of the efforts of the ISO-9126 to clarify terminology. 'Interoperability' refers to the ability of the software to interact with other systems. The framers of the ISO-9126 have chosen this word rather than compatibility because the latter causes confusion with the characteristic referred to by the ISO-9126 as 'replace ability' (see below).

Characteristic	Sub-characteristics
Reliability	Maturity Fault tolerance Recoverability

Maturity refers to the frequency of failure due to faults in a software product, the implication being that the more the software has been used, the more faults will have been uncovered and removed. It is also interesting to note that recoverability has been clearly distinguished from security which describes the control of access to a system.

Characteristic	Sub-characteristics
Usability	Understandability Learnability Operability

Understandability is a pretty clear quality to grasp, although the definition “attributes that bear on the users’ efforts for recognizing the logical concept and its applicability” in our view actually makes it less clear!

Note how learnability is distinguished from operability. A software tool could be easy to learn but time-consuming to use because, perhaps, it uses a large number of nested menus. This might be fine for a package used intermittently, but not where the system is used for many hours each day. In this case, learnability has been incorporated at the expense of operability.

Characteristics	Sub-characteristics
Efficiency	Time behavior Resource behavior
Maintainability	Analyzability Changeability Stability Testability

Analyzability is the quality that McCall called diagnosability, the ease with which the cause of a failure can be determined. Changeability is the quality that others have called flexibility: the latter name is perhaps a better one as changeability has a slightly different connotation in plain English—it implies that the suppliers of the software are always changing it!

Stability, on the other hand, does not mean that the software never changes: It means that there is a low risk of a modification to the software having unexpected effects.

Characteristic	Sub-characteristics
Portability	Adaptability Installability Conformance Replaceability

Conformance, as distinguished from compliance, relates to those standards that have a bearing on portability. The use of a standard programming language common to many software/hardware environments would be an example of conformance. Replaceability refers to the factors that give 'upwards compatibility' between old software components and the new ones. Downwards compatibility is specifically excluded from the definition.

The ISO-9126 provides guidelines for the use of the quality characteristics. Variation in the importance of different quality characteristics depending on the type of product is stressed. Thus, reliability will be of particular concern with safety critical systems while efficiency will be important for some real-time systems. For interactive end-user systems, the key quality might be usability. Once the requirements for the software product have been established the following steps are suggested:

- Quality metrics selection
- Ratings level definition
- Assessment criteria definition

4.6 COMPARISON OF ISO-9000 CERTIFICATION AND THE SEI-CMM

Comparison of some of the key characteristics of ISO-9000 certification and the SEI-CMM model are as follows:

- The emphasis of the SEI-CMM is on continuous process improvement, whereas the ISO-9000 addresses the minimum criteria for an acceptable quality system.
- The Capability Maturity Model (CMM) is a five-level framework for measuring software-engineering practices, as they relate to a process. On the other hand the ISO-9000 defines a minimum level of generic attributes for a quality-management program.
- The ISO-9000 is awarded by an international standard body. Therefore, ISO-9000 certification can be quoted by an organization in official documents, in communication with external parties, and in tender quotations. However, SEI-CMM assessment is purely for internal use.
- The ISO-9000 standards were basically designed to audit manufacturing/service organizations, whereas the CMM was developed specifically for the software industry and thus addresses several issues specific to the software industry.

- The SEI-CMM focuses strictly on software, while the ISO-9000 has a much wider scope: hardware, software, processed materials, and services.
- The SEI-CMM model provides a list of key process areas (KPA) on which an organization at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement.
- The SEI-CMM model aims for achieving Total Quality Management (TQM), which is beyond quality assurance, whereas the ISO-9000 aims at Level 3 (defined level) of the SEI-CMM model.
- The ISO-9000 requires that procedures for handling, storage, packaging, and delivery be established and maintained, while replication, delivery, and installation are not covered in the SEI-CMM.
- The ISO-9000 requires that the product be identified and traceable during all stages of production, delivery, and installation while the SEI-CMM covers this clause primarily in software-configuration management.

4.7 RELIABILITY ISSUES

4.7.1 Software Reliability

Software reliability is defined as the probability of failure-free operation of a computer program in a specified environment for a specified time.

OR

Reliability of a software product essentially denotes its trustworthiness or dependability.

OR

Reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

The expected curve for software is given in Figure 4.5.

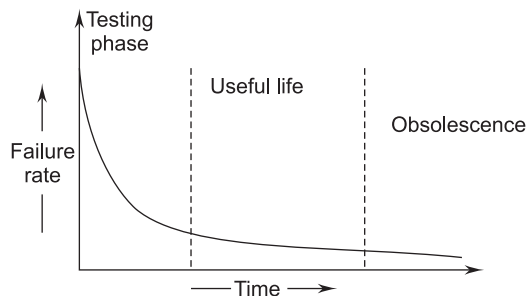


FIGURE 4.5 Software Reliability Curve (Failure Rate Versus Time)

Software may be retired only if it becomes obsolete. Some of the 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

4.7.2 Software-Reliability Specifications

Reliability is a complex concept, which should always be considered at the system level rather than the individual component level. Because the components in a system are interdependent, a failure in one component can be propagated through the system and affect the operation of other components. In a computer-based system, you have to consider three dimensions when specifying the overall system reliability:

1. **Hardware Reliability.** What is the probability of a hardware component failing and how long does it take to repair that component?
2. **Software Reliability.** How likely is it that a software component will produce an incorrect output? Software failures are different from hardware failures in that software does not wear out. It can continue operating correctly after an incorrect result has been produced.
3. **Operator Reliability.** How likely is it that the operator of a system will make an error?

The reliability of a system depends upon a number of factors rather than the sum of all the probabilities (failure probabilities of each factor).

$$P_S = P_A + P_R + \dots + P_N$$

where

P_S = Probability of system failure

P_A = Probability of component A failure

P_B = Probability of component B failure

P_N = Probability of component N failure

As the number of factors increase, the overall probability of system failure increases.

4.7.3 Reliability Terminologies

TABLE 4.3 Reliability Terminologies

Term	Description
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users.
System error	Erroneous system behavior where the behavior of the system does not conform to its specifications.
System fault	An incorrect system state, i.e., a system state that is unexpected by the designers of the system.
Human errors or mistakes	Human behavior that results in the introduction of faults into a system.

4.7.4 Classification of Failures

When a system specifies the reliability then the engineer should identify the failure type and consider whether it should be treated differently in specification. Many large systems are divided into small subsystems and each subsystem has different reliability requirements. It is easy and cheap to assess the reliability requirements of each subsystem separately.

TABLE 4.4 Failure Classifications

Failure Class	Description
Recoverable	The system can recover with or without operator intervention.
Transient	Occurs only with certain inputs.
Unrecoverable	The system cannot recover without operator intervention or the system may need to be restarted.
Corrupting	Failure corrupts system data.
Non-corrupting	Failure does not corrupt system data.
Permanent	Occurs for all input values while invoking a function of the system.

4.8 RELIABILITY METRICS

Reliability metrics are used to quantitatively express the reliability of a software product.

Some reliability metrics, which can be used to quantify the reliability of a software product are:

1. **MTTF (Mean Time to Failure).** Components have an average lifetime and this is reflected in the most widely used hardware reliability metric, Mean Time to Failure (MTTF). The MTTF is the mean time for which a component is expected to be operational.

The MTTF is the average time between observed system failures. An MTTF of 500 means that one failure can be expected every 500 time units. The time units are totally dependent on the system and it can even be specified in the number of transactions, as is the case of database query systems.

MTTF is relevant for systems with long transactions, i.e., where system processing takes a long time. MTTF should be longer than transaction length. For example, it is suitable for computer-aided design systems where a designer will work on a design for several hours as well as for word-processor systems.

2. **MTTR (Mean Time to Repair).** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and then to fix them.

OR

MTTR is the average time to replace a defective component.

Once a hardware component fails then the failure is usually permanent so the Mean Time to Repair (MTTR) reflects the time taken to repair or replace the component.

3. **MTBF (Mean Time Between Failures).** We can combine the MTTF and MTTR metrics to get the MTBF metric:

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

Thus, a MTBF of 300 hours indicates that once a failure occurs, the next failure is expected to occur only after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF.

4. **POFOD (Probability of Failure on Demand).** POFOD is the likelihood that the system will fail when a service request is made. A POFOD of 0.001 means that one out of a thousand service requests may result in failure.

POFOD is an important measure for safety critical systems and should be kept as low as possible. It is relevant for many safety-critical systems with the exception of management components, such as an emergency shutdown system in a chemical plant.

5. **ROCOF (Rate of Occurrences of Failure).** ROCOF is the frequency of occurrence with which unexpected behavior is likely to occur. A ROCOF of 2/100 means that two failures are likely to occur in each 100 operational time units. This metric is sometimes called the failure intensity.

It is relevant for operating systems and transaction-processing systems where the system has to process a large number of similar requests that are relatively frequent; for example, credit-card processing systems, airline-booking systems, etc.

6. **AVAIL (Availability).** Availability is the probability that the system is available for use at a given time. An availability of 0.998 means that in every 1000 time units, the system is likely to be available for 998 of these.

4.8.1 Measurements of Reliability and Availability

Measurement of Reliability

Most hardware-related reliability models are predicted on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., effect of temperature corrosion) are more likely than a design-related failure. There has been debate over the relationship between key concepts in hardware reliability and their applicability to software. Although an irrefutable link has yet to be established.

If we consider a computer-based system, a simple measure of reliability is mean-time-between-failure (MTBF) and it can be expressed as:

$$\text{MTBF} = \text{MTTF} + \text{MTTR},$$

where

MTTF = Mean Time to Failure

MTTR = Mean Time to Repair.

Measurement of Availability

Many researchers argue that MTBF is a far more useful measure than defects/KLOC or defects/FP because an end-user is concerned with failures, not with the total error count. Because each error contained within a program does not have the

same failure rate, the total error count provides little indication of the reliability of a system. In addition to the reliability measure, we must develop a measure of availability. Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as:

$$\text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})} \times 100\%.$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR.

4.9 RELIABILITY GROWTH MODELING

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when a particular level of reliability is likely to be attained. Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. Although several reliability growth models have been proposed as discussed in the following sections.

4.9.1 Jelinski-Moranda Model

The Jelinski-Moranda model is the earliest and probably the best-known reliability model. It proposes a failure intensity function in the form of

$$\lambda(t) = \phi(N - i + 1),$$

where

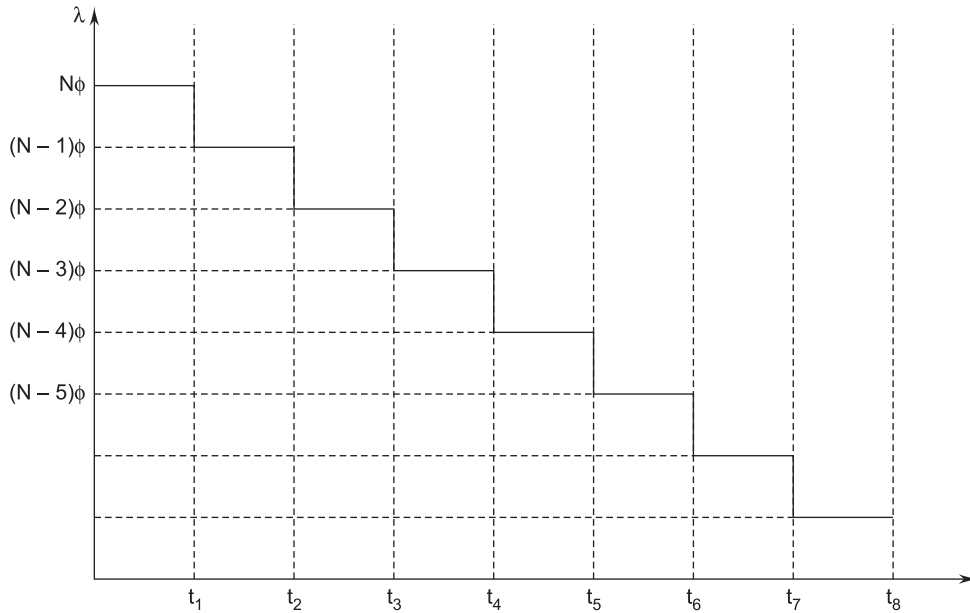
ϕ = Constant of proportionality

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 and failure intensity is shown in Figure 4.6.

FIGURE 4.6 Relation Between T and λ

The time interval t_1, t_2, \dots, t_k may vary in duration depending upon the occurrence of a failure.

Between the $(i-1)^{\text{th}}$ and i^{th} failure, the failure intensity function is $(N-i+1)\phi$.

Example 4.1. *There are 50 errors estimated to be present in a program. We have experienced 30 errors. Use the Jelinski-Moranda model to calculate the failure intensity with a given value of $\phi = 0.03$. What will be the failure intensity after the experience of 40 errors?*

Solution.

$$N = 50 \text{ errors}$$

$$i = 30 \text{ failures}$$

$$\phi = 0.03$$

We know

$$\begin{aligned} \lambda(t) &= \phi(N-i+1) \\ &= 0.03(50-30+1) \\ &= 0.63 \text{ failures/CPU hr.} \end{aligned}$$

After 40 failures

$$\begin{aligned} \lambda(t) &= 0.03(50-40+1) \\ &= 0.33 \text{ failures/CPU hr.} \end{aligned}$$

Hence, there is a continuous decrease in the failure intensity as the number of failures experienced increases.

4.9.2 Little Wood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases. It treats an error's contribution to reliability improvement as an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as testing continues.

There are more complex reliability growth models that give greater accurate approximations to reliability growth. However, these models are beyond the scope of this text.

4.9.3 Step Function Model

The simplest reliability growth model is a step function model, where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in Figure 4.7.

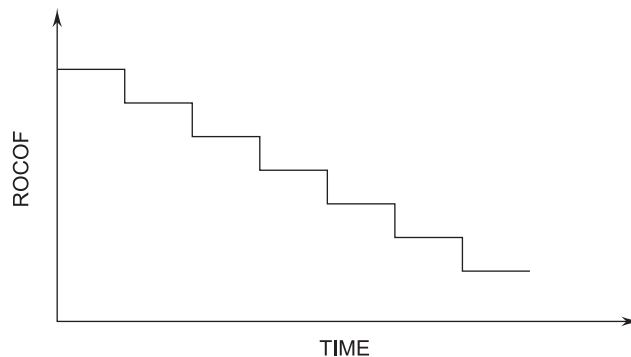


FIGURE 4.7 Step Function Model of Reliability Growth

However, this simple model of reliability, which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since we already know that corrections of different errors contribute differently to reliability growth.

4.10 RELIABILITY ASSESSMENT

In general, many properties of engineering artifacts, such as reliability, are measured and verified in this way. For instance, the reliability of an electrical appliance may be measured in terms of its probability of failure within a given time. This measure is helpful whenever we cannot guarantee absence of failures absolutely. Software reliability is the probability of the failure-free operation of a computer program for a specified time in a specified environment.

The process of measuring the reliability of a system is illustrated in Figure 4.8.

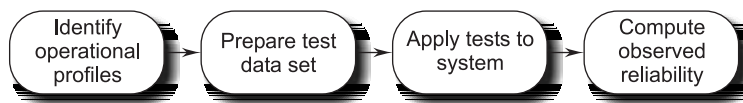


FIGURE 4.8 The Reliability Measurement Process

This process involves four stages:

- Existing systems of the same type are studied to establish an operational profile. An operational profile identifies different classes of system inputs and the probability of these inputs in normal use.
- A set of test data is constructed (sometimes with the help of test-data generators) that reflects the operational profile.
- The system is tested with these data and the number of failures is observed. The times of these failures are also logged; the time units chosen should be appropriate for the reliability metric used.
- After a statistically significant number of failures have been observed, the software reliability can then be computed. You can then work out the appropriate reliability metric value.

This approach to reliability measurement is not easy to apply in practice. The principal difficulties that arise are due to:

- Operational profile uncertainty. The operational profiles may not be an accurate reflection of the real use of the system.
- High costs of test-data generation. Defining a large amount of test data takes a long time if it is not possible to generate this data automatically.
- Statistical uncertainty when high reliability is specified. It is important to generate a statistically significant number of failures to allow accurate reliability measurements.

EXERCISES

1. State some of the software quality factors that are proposed by McCall.
2. Define representative qualities.
3. Describe the various classifications of software quality.
4. How do you define reliability? Discuss various models for reliability allocation.
5. What is software quality assurance?
6. What are the goals of software quality assurance?
7. What are the objectives of software quality assurance?
8. Define an SQA plan.
9. Why is it important for a software-development organization to obtain ISO-9000 certification?
10. Discuss the main requirements of the ISO-9001 and compare it with the SEI-CMM.
11. Discuss how reliability changes over the lifetime of a software product and a hardware product.
12. Define:
 - (i) Software quality
 - (ii) Reliability metrics
 - (iii) Software reliability
13. Compare:
 - (i) ISO and SEI-CMM
 - (ii) Software and hardware reliability
14. Explain, in detail, the SEI-CMM model. Also, differentiate it with ISO.
15. Give the shortcomings of ISO-9000 certification.
16. Explain any two types of reliability growth models.
17. What is software reliability and software availability? Also, discuss how they are measured.
18. What is software quality?
19. Explain the steps an organization will take in order to obtain ISO-9000 certification.
20. Answer the following questions:
 - (i) Can a program be correct and still not be reliable? Explain.
 - (ii) What is ISO-9000 certification?
 - (iii) What are the salient features of ISO-9001 requirements?
 - (iv) What is software reliability? Explain.

Chapter 5

SYSTEM DESIGN

5.1 SYSTEM/SOFTWARE DESIGN

Design is a meaningful representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design.

A set of design concepts has evolved over the years. According to M.A. Jackson, *"The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work and getting it right."* The various design concepts discussed in this chapter provide the necessary framework for "getting it right."

5.1.1 Definition of Software Design

The definitions of software design are as diverse as design methods. Some important software design definitions are outlined below.

According to Coad and Yourdon. *Software Design is the practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details.*

According to Webster. *In a sense, design is representation of an object being created. A design information base that describes aspects of this object, and the design process can be viewed as successive elaboration of representations, such as adding more information or even backtracking and exploring alternatives.*

According to Stevens. *Software Design is the process of inventing and selecting programs that meet the objectives for software systems.*

Input includes an understanding of the following:

- Requirements
- Environmental constraints
- Design criteria

The output of the design effort is composed of the following:

- Architecture design which shows how pieces are interrelated
- Specifications for any new pieces
- Definitions for any new data

5.1.2 Design Objectives/Properties

The various desirable properties or objectives of software design are:

1. **Correctness.** The design of a system is correct if the system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.
2. **Verifiability.** Design should be correct and it should be verified for correctness. Verifiability is concerned with how easily the correctness of the design can be checked. Various verification techniques should be easily applied to design.
3. **Completeness.** Completeness requires that all the different components of the design should be verified, i.e., all the relevant data structures, modules, external interfaces, and module interconnections are specified.
4. **Traceability.** Traceability is an important property that can get design verification. It requires that the entire design element be traceable to the requirements.
5. **Efficiency.** Efficiency of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost

considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system consumes less processor time and memory.

6. **Simplicity.** Simplicity is perhaps the most important quality criteria for software systems. Maintenance of a software system is usually quite expensive. The design of the system is one of the most important factors affecting the maintainability of the system.

5.1.3 Design Principles

The three design principles are as follows:

- Problem partitioning
- Abstraction
- Top-down and Bottom-up design

1. **Problem Partitioning.** When solving a small problem, the entire problem can be tackled at once. For solving larger problems, the basic principle is the time-tested principle of “divide and conquer.” This principle suggests dividing into smaller pieces, so that each piece can be conquered separately.

For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.

However, the different pieces cannot be entirely independent of each other as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning.

Problem partitioning can be divided into two categories:

- (i) Horizontal partitioning
- (ii) Vertical partitioning
 - (i) *Horizontal Partitioning.* Horizontal partitioning defines separate branches of modular hierarchy for each major program function. The simplest

approach to horizontal partitioning defines three partitions: input, data transformation (often called processing), and output. Partitioning their architecture horizontally provides a number of distinct benefits:

- Software that is easier to test.
- Software that is easier to maintain.
- Software that is easier to extend.
- Propagation of fewer side effects.

Conversely, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow.

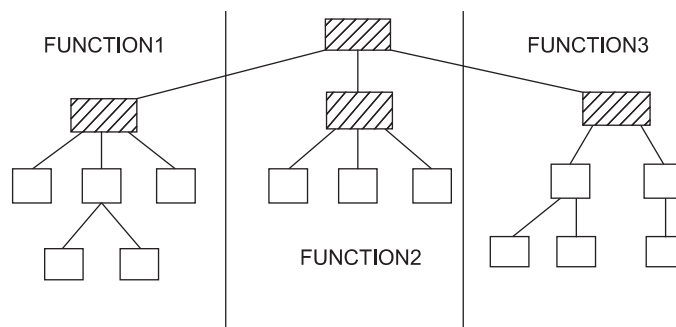


FIGURE 5.1 Horizontal Partitioning

- (ii) *Vertical Partitioning.* Vertical partitioning, often called factoring, suggests that control and work should be distributed from top-down in the program structure. Top-level modules should perform control functions and do actual processing work. Modules that reside low in the structure should be the workers, performing all input, compilation, and output tasks.

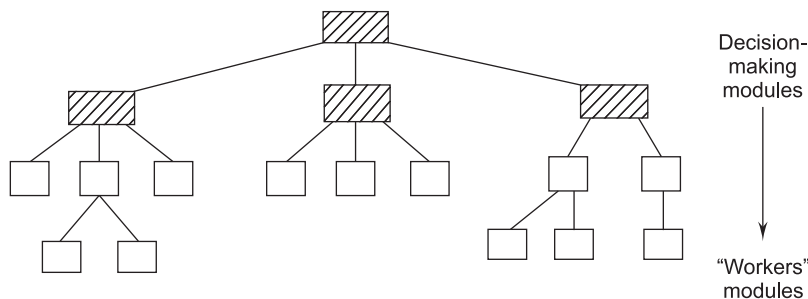


FIGURE 5.2 Vertical Partitioning

2. **Abstraction.** An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior.

Abstraction is an indispensable part of the design process and it is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other, but interact with other components. In order to allow the designer to concentrate on one component at a time, abstraction of other components is used.

Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase.

During the design process, abstractions are used in a reverse manner not in the process of understanding a system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

There are two common abstraction mechanisms for software systems: Functional abstraction and data abstraction. In functional abstraction, a module is specified by the function it performs. For example, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being partitioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function.

The second unit for abstraction is data abstraction. There are certain operations required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects. From outside an object, the internals of the object are hidden; only the operations on the object are visible.

3. **Top-down and Bottom-up Design.** A system consists of components, which have components of their own; indeed a system is a hierarchy of components. The highest-level components correspond to the total system.

To design such hierarchies there are two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in some form of 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. The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design. Most design methodologies are based on the top-down approach.

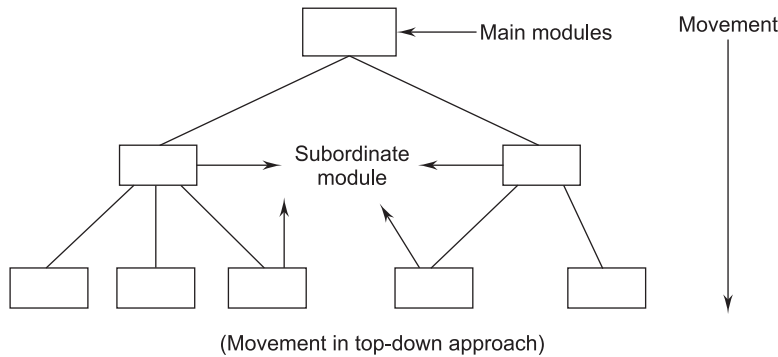


FIGURE 5.3 Top-Down Approach

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components. Bottom-up methods work with layers of abstraction. Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and still a higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. However, if a system is to be built from an existing system, a bottom-up approach is more suitable, as it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable (in the first iteration a top-down approach can be used).

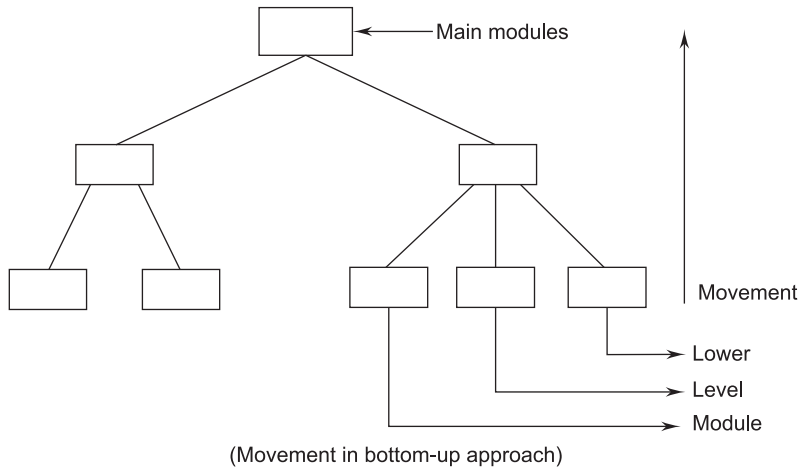


FIGURE 5.4 Bottom-Up Approach

5.2 ARCHITECTURAL DESIGN

Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these subsystems and establishing a framework for subsystem control and communication is called architectural design.

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Architectural design methods have a look into various architectural styles for designing a system. These are:

- Data-centric architecture
- Data-flow architecture
- Object-oriented architecture
- Layered architecture

Data-centric architecture involves the use of a central database operation of inserting and updating it in the form of a table. Data-flow architecture is central around the pipe and filter mechanism. This architecture is applied when input data takes the form of output after passing through various phases of transformations. These transformations can be via manipulations or various computations done

on the data. In object-oriented architecture the software design moves around the clauses and objects of the system. The class encapsulates the data and methods. Layered architecture defines a number of layers and each layer performs tasks. The outer-most layer handles the functionality of the user interface and the inner-most layer mainly handles interaction with the hardware.

5.2.1 Objectives of Architectural Design

The objective of architectural design is to develop a model of software architecture, which gives an overall organization of the program module in the software product. Software architecture encompasses two aspects of structures of the data and hierarchical structures of the software components. Let us see how a single problem can be translated to a collection of solution domains (see Figure 5.5).

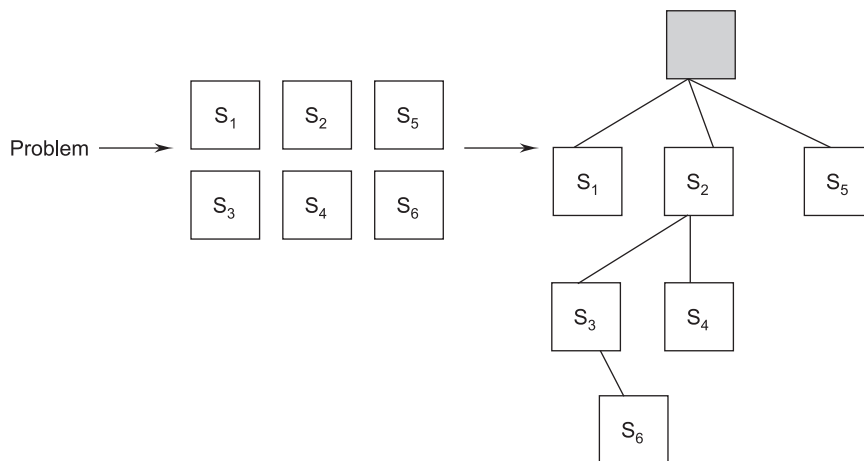


FIGURE 5.5 Problems, Solutions, and Architecture

Architectural design defines the organization of program components. It does not provide the details of each component and its implementation. Figure 5.6 depicts the architecture of a financial accounting system.

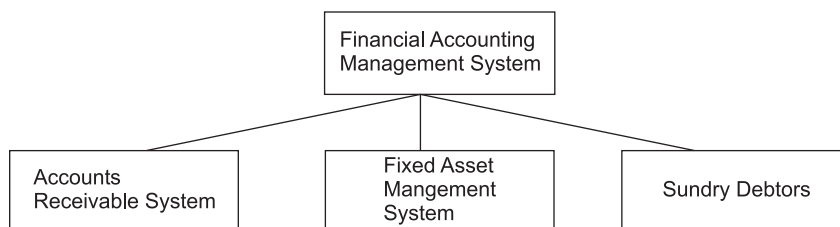


FIGURE 5.6 Architecture of a Financial Accounting System

The objective of architectural design is also to control the relationship between modules. One module may control another module or may be controlled by another module. These characteristics are defined by the fan-in and fan-out of a particular module. The organization of a module can be represented by a tree-like structure.

The number of levels of a component in the structure is called depth and the number of components across the horizontal section is called width. The number of components, which controls the component, is called fan-in, i.e., the number of incoming edges to a component. The number of components that are controlled by the module is called fan-out, i.e., the number of outgoing edges.

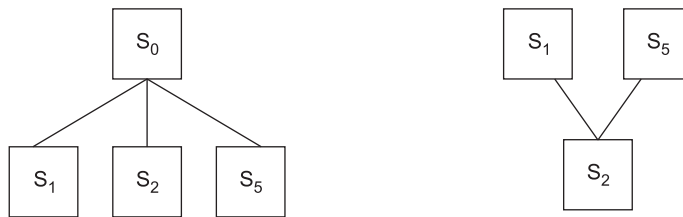


FIGURE 5.7 Fan-in and Fan-out

S_0 controls three components, hence, the fan-out is 3. S_2 is controlled by two components, namely, S_1 and S_5 , hence, the fan-in is 2 (see Figure 5.7).

5.3 LOW-LEVEL DESIGN

5.3.1 Modularization

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.

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 “a module is a work assignment for an individual programmer.” All of these definitions are correct, in the sense that modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, and each control abstraction handles a local aspect of the problem being solved. Modular system consists of well-defined, manageable units with well-defined interfaces among the units. Desirable properties of a modular system include:

- Each function in each abstraction has a single, well-defined purpose.
- Each function manipulates no more than one major data structure.

- Functions share global data selectively. It is easy to identify all routines that share a major data structure.
- Functions that manipulate instances of abstract data types are encapsulated with the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

Modules that may be created during program modularizations are:

- **Process support modules:** In these all the functions and data items that are required to support a particular business process are grouped together.
- **Data abstraction modules:** These are abstract types that are created by associating data with processing components.
- **Functional modules:** In these all the functions that carry out similar or closely related tasks are grouped together.
- **Hardware modules:** In these all the functions, which control particular hardware are grouped together.

Classification of Modules

A module can be classified into three types depending on the activating mechanism.

- An incremental module is activated by an interruption and can be interrupted by another interrupt during the execution prior to completion.
- A sequential module is a module that is referenced by another module and without interruption of any external software.
- Parallel modules are executed in parallel with other modules.

The main purpose of modularity is that it allows the principle of separation of concerns to be applied in two phases: when dealing with the details of each module in isolation (and ignoring the details of other modules) and when dealing with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system. If the two phases are temporally executed in the order mentioned, then we say that the system is designed bottom-up; the converse denotes top-down design.

Advantages of Modular Systems

- Modular systems are easier to understand and explain because their parts are functionally independent.
- Modular systems are easier to document because each part can be documented as an independent unit.
- Programming individual modules is easier because the programmer can focus on just one small, simple problem rather than a large complex problem.

- Testing and debugging individual modules is easier because they can be dealt with in isolation from the rest of the program.
- Bugs are easier to isolate and understand, and they can be fixed without fear of introducing problems outside the module.
- Well-composed modules are more reusable because they are more likely to comprise part of a solution to many problems. Also, a good module should be easy to extract from one program and insert into another.

Modularity is an important property of most engineering processes and products. For example, in the automobile industry, the construction of cars proceeds by assembling building blocks that are designed and built separately. Furthermore, parts are often reused from model to model, perhaps after minor changes. Most industrial processes are essentially modular, made out of work packages that are combined in simple ways (sequentially or overlapping) to achieve the desired result.

5.3.2 Structure Charts

The structure chart is one of the most commonly used methods for system design. Structure charts are used during architectural design to document hierarchical structures, parameters, and interconnections in a system.

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 a black box and appropriate outputs are generated by the black box. This concept reduces 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 Figures 5.8 (a) and (b).

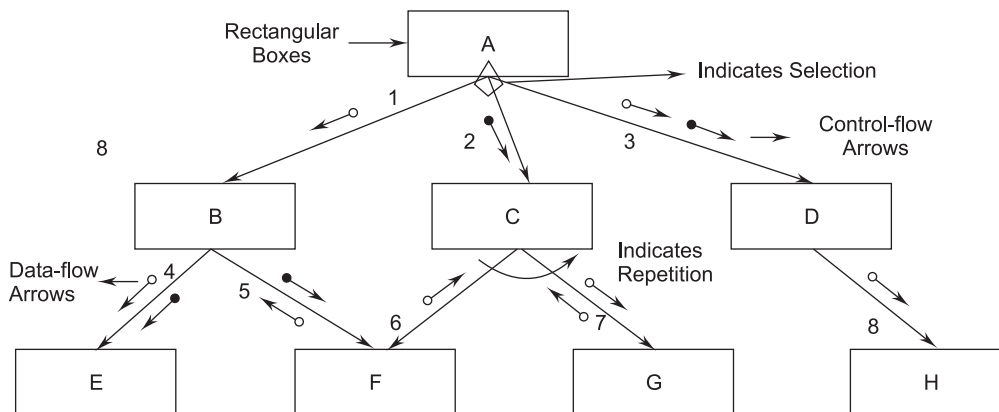


FIGURE 5.8 (a) Hierarchical Format of a Structure Chart

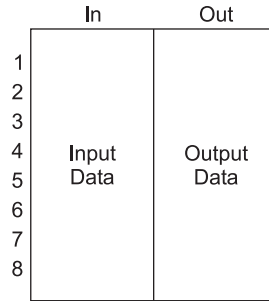


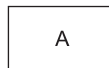
FIGURE 5.8 (b) Format of a Structure Chart

Modules at the top level call the modules at the lower level. The connections 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 a hierarchical numbering scheme. In any structure chart there is one and only one module at the top called the root.

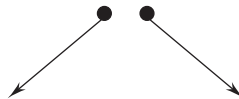
Basic Building Blocks of a Structure Chart

The basic building blocks of a structure chart are the following:

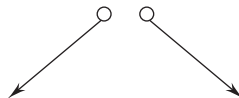
1. **Rectangular Boxes.** A rectangular box represents a module. Usually a rectangular box is annotated with the name of the module it represents.



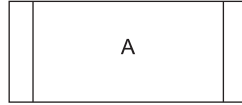
2. **Arrows.** An arrow connecting two modules implies that during program execution, control is passed from one module to the other in the direction of the connecting arrow.



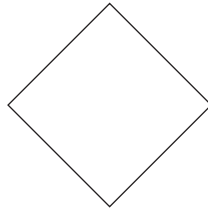
3. **Data-flow Arrows.** Data-flow arrows represent that the named data passes from one module to the other in the direction of the arrow.



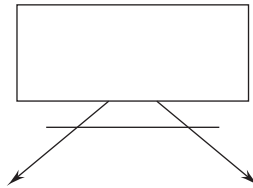
4. **Library Modules.** Library modules are the frequently called modules and are usually represented by a rectangle with double edges. Usually when a module is invoked by many other modules, it is made into a library module.



5. **Selection.** The diamond symbol represents that one module out of several modules connected with the diamond symbol are invoked depending on the condition satisfied, which is written in the diamond symbol.



6. **Repetitions.** A loop around the control-flow arrows denotes that the respective modules are invoked repeatedly.



Example 5.1. A software system called RMS calculating software reads three integral numbers from the user in the range between -1000 and $+1000$ and determines the root mean square (rms) of the three input numbers and then displays it.

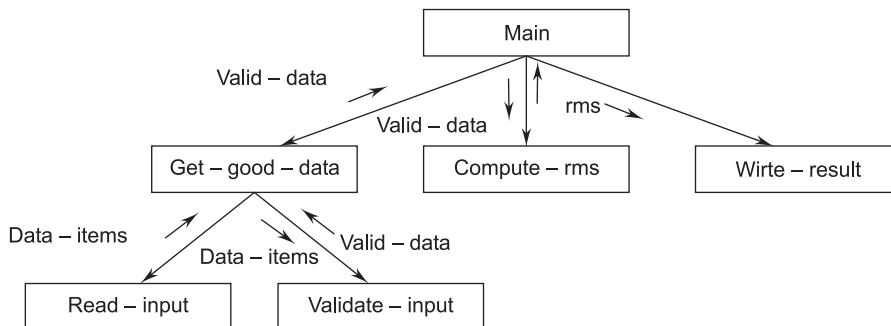


FIGURE 5.9 Structure Chart For Example 5.1

5.3.3 Pseudo-Code

“Pseudo” means imitation or false and “code” refers to the instructions written in a programming language. Pseudo-code notation can be used in both the preliminary and detailed design phases. Using pseudo-code, the designer describes system characteristics using short, concise English language phrases that are structured by keywords, such as If-Then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Pseudo-code is also known as program-design language or structured English. A program-design language should have the following characteristics:

- A fixed syntax of keywords that provide for all structured constructs, data declarations, and modularity characteristics.
- A free syntax of natural language that describes a processing feature.
- A data-declaration facility.
- A subprogram definition and calling techniques.

Advantages of Pseudo-Code

The various advantages of pseudo-code are as follows:

- Converting a pseudo-code to a programming language is much easier compared to converting a flowchart or decision table.
- Compared to a flowchart, it is easier to modify the pseudo-code of program logic whenever program modifications are necessary.
- Writing of pseudo-code involves much less time and effort than the equivalent flowchart.
- Pseudo-code is easier to write than writing a program in a programming language because pseudo-code as a method has only a few rules to follow.

Disadvantages of Pseudo-Code

The various disadvantages of pseudo-code are as follows:

- In the case of pseudo-code, a graphic representation of program logic is not available as with flowcharts.
- There are no standard rules to follow in using pseudo-code. Different programmers use their own style of writing pseudo-code and hence communication problems occur due to lack of standardization.
- For a beginner, it is more difficult to follow the logic or write the pseudo-code as compared to flowcharting.

The use of pseudo-code for detailed design specification is illustrated in Figure 5.9 (a).

```

INITIALIZE tables and counters; OPEN files
READ the first text record
WHILE there are more text records DO
  WHILE there are more words in the text record DO
    EXTRACT the next word
    SEARCH word_table for the extracted word
    IF the extracted word is found THEN
      INCREMENT the extracted word's occurrence count
    ELSE
      INSERT the extracted word into the word_table
    ENDIF
  INCREMENT the words_processed counter
  ENDWHILE at the end of the text record
ENDWHILE when all text records have been processed
PRINT the word_table and the words_processed counter
CLOSE files
TERMINATE the program

```

FIGURE 5.9 (a) An Example of a Pseudo-Code Design Specification

Pseudo-code consists of English-like statements describing an algorithm. It is written using simple phrases and avoids cryptic symbols. It is independent of high-level languages and is a very good means of expressing an algorithm. It is written in a structured manner and indentation is used to increase clarity.

5.3.4 Flowcharts

A flowchart is a convenient technique to represent the flow of control in a program. A flowchart is a pictorial representation of an algorithm that uses symbols to show the operations and decisions to be followed by a computer in solving a problem. The actual instructions are written within symbols/boxes using clear statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation in a sequence.

In fact, flowcharts are the plan to be followed when the program is written. Expert programmers may write programs without drawing the flowcharts. But for a beginner it is recommended that a flowchart should be drawn before writing a program, which in turn will reduce the number of errors and omissions

in the program. Flowcharts also help during testing and modifications in the programs.

Flowchart Symbols

1. **Terminal Symbol.** Terminal symbols are used for two purposes: to define the starting (START or BEGIN) point of the flowchart and to define the ending point (END) of the flowchart.



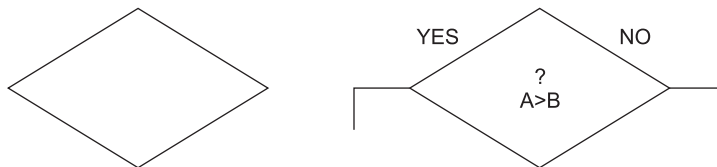
2. **Input/Output Symbol.** Input/output symbols are used to indicate the logical positioning of input/output operations. The input operation is the entry of computer data and the output operation is the displayed output operation.



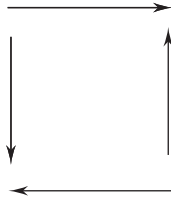
3. **Processing Symbol.** Processing symbols are used to indicate the arithmetic and data-movement instructions. Therefore, all arithmetic processing of adding, subtracting, multiplying, and dividing are represented with a processing symbol box.



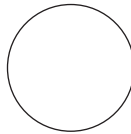
4. **Decision Symbol.** Decision symbols have one entry point and there will be at least two exit points depending upon the decision taken inside the symbol. When a condition is tested, if the condition is true, the path for “yes” is followed. If the condition is false, the path for “no” is followed.



5. **Flow Lines.** Flow lines, which have arrowheads, are used to indicate the flow of program logic in a flowchart. These arrows are used to indicate the direction of the flow of control. This means these statements indicate the next statement to be executed.



6. **Connector Symbol.** If a flowchart is discontinued at some point and continued again in another place, the connector symbol is used. It is a circle with a number written inside it. If a flowchart is discontinued at some point, a circle is drawn pointing away from the chart. Another circle with the same number inside is placed where the flowchart is continued.



7. **Hexagon (Flat).** This is the preparation box. This box contains the loop-setting statement, i.e., some iterative statement.



Flowchart Drawing Rules

Important rules and guidelines used for drawing flowcharts are:

- Only conventional flowchart symbols should be used.
- Arrows can be used to indicate the flow of control in the problem. However, flow lines should not cross each other.
- Processing logic should flow from top to bottom and from left to right.
- Words in the flowchart symbols should be common statements and easy to understand. These should be independent of programming languages.
- Be consistent in using names and variables in the flowchart.
- If the flowchart becomes large and complex then connector symbols should be used to avoid crossing of flow lines.
- Properly labeled connectors should be used to link the portions of the flowchart on different pages.
- Flowcharts should have start and stop points.

Advantages of Flowcharts

The various advantages of flowcharts are as follows:

- **Synthesis.** Flowcharts are used as working models in designing new programs and software systems.
- **Documentation.** Program documentation consists of activities, such as collecting, organizing, storing, and maintaining all related records of a program.
- **Coding.** Flowcharts guide the programmer in writing the actual code in a high-level language, which is supposed to give an error-free program developed expeditiously.
- **Debugging.** The errors in a program are detected only after its execution on a computer. These errors are called bugs and the process of removing these errors is called debugging. In the debugging process, a flowchart acts as an important tool in detecting, locating, and removing bugs from a program.
- **Communication.** A flowchart is a pictorial representation of a program. Therefore, it is an excellent communication technique to explain the logic of a program to other programmers/people.
- **Analysis.** Effective analysis of a logical problem can be easily done with the help of a related flowchart.
- **Testing.** A flowchart is an important tool in the hands of a programmer, which helps him in designing the test data for systematic testing of programs.

Limitations of Flowcharts

The various limitations of flowcharts are as follows:

- The drawing of flowcharts is a very time-consuming process and laborious especially for large, complex problems.
- The redrawing of flowcharts is even more difficult and time consuming. It is very difficult to include any new step in the existing flowchart; redrawing of the flowchart is the only solution.
- There are no standards, which specify the detail that should be included in any flowchart.
- If an algorithm has complex branches and loops, flowcharts become very difficult to draw.
- Sometimes flowcharts are not as detailed as desired.

Example of a Flowchart

As an example, consider an algorithm to find the average of n numbers. The flowchart is shown in Figure 5.10 followed by the algorithm. Here n is the integer

variable denoting the number of values considered for computing the average. Count is another integer variable denoting the number of values that are processed at any instant. The number is an integer variable for storing the values.

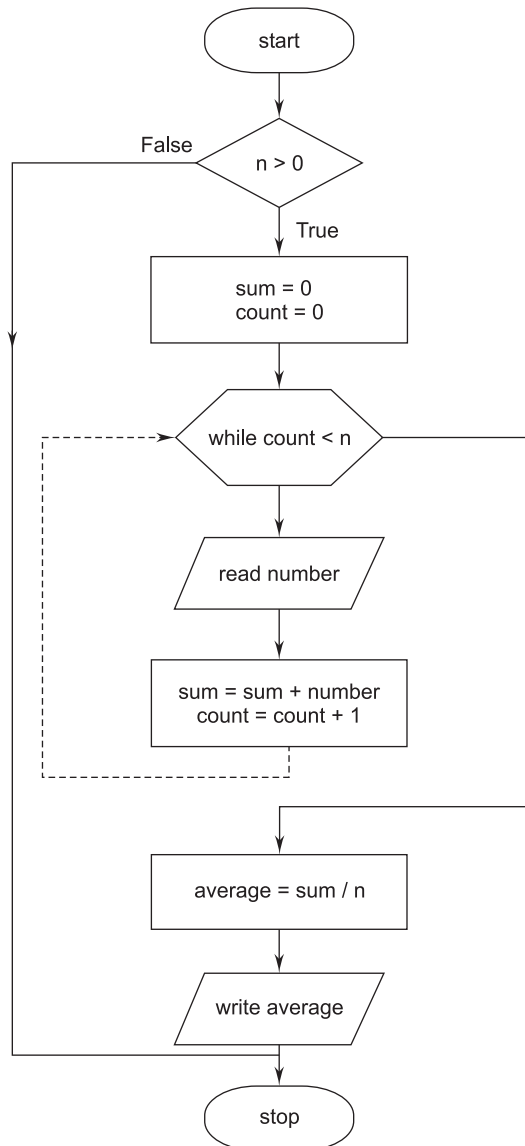


FIGURE 5.10 Flowchart to Find the Average of n Numbers

5.3.5 Difference Between Flowcharts and Structure Charts

A structure chart differs from a flowchart in the following ways:

- It is usually difficult to identify different modules of the software from its flowchart representation.
- Data interchange among different modules is not represented in a flowchart.
- Sequential ordering of tasks inherent in a flowchart is suppressed in a structure chart.
- A structure chart has no decision boxes.

Unlike flowcharts, structure charts show how different modules within a program interact and the data that is passed between them.

5.4 COUPLING AND COHESION

5.4.1 Coupling

The coupling between two modules indicates the degree of interdependence between them. If two modules interchange a large amount of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Highly Coupled: When the modules are highly dependent on each other then they are called highly coupled.

Loosely Coupled: When the modules are dependent on each other but the interconnection among them is weak then they are called loosely coupled.

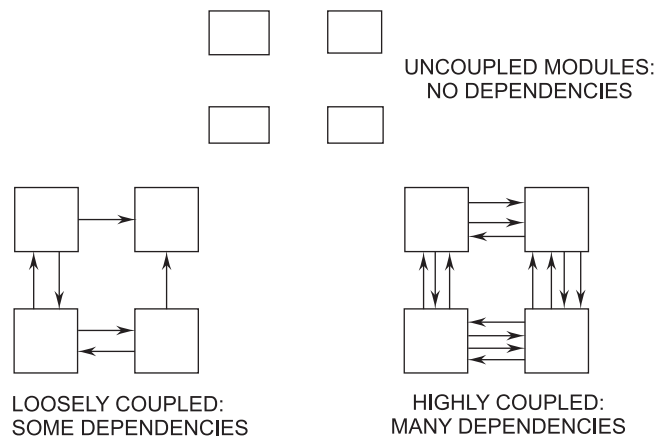


FIGURE 5.11 Coupling

Uncoupled: When the different modules have no interconnection among them then they are called uncoupled modules.

Factors Affecting Coupling Between Modules

The various factors which affect the coupling between modules are depicted in Table 5.1.

TABLE 5.1 Factors Affecting Coupling

	Interface Complexity	Type of Connection	Type of Communication
Low	Simple Obvious	To module by name	Data
High	Complicated Obscure	To internal elements	ControlHybrid

Types of Couplings

Different types of couplings include content, common, external, control, stamp, and data. The strength of a coupling from the lowest coupling (best) to the highest coupling (worst) is given in Figure 5.12.

Data coupling	Best
Stamp coupling	↑
Control coupling	↑
External coupling	↑
Common coupling	↑
Content coupling	(Worst)

FIGURE 5.12 The Types of Module Coupling

1. **Data Coupling.** Two modules are data coupled if they communicate using an elementary data item that is passed as a parameter between the two; for example, an integer, a float, a character, etc. This data item should be problem related and not used for a control purpose.

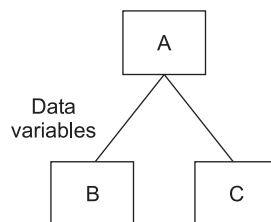


FIGURE 5.13 Data Coupling

When a non-global variable is passed to a module, modules are called data coupled. It is the lowest form of a coupling. For example, passing the variable from one module in C and receiving the variable by value (i.e., call by value).

2. **Stamp Coupling.** Two modules are stamp coupled if they communicate using a composite data item, such as a record, structure, object, etc. When a module passes a non-global data structure or an entire structure to another module, they are said to be stamp coupled. For example, passing a record in PASCAL or a structure variable in C or an object in C++ language to a module.
3. **Control Coupling.** Control coupling exists between two modules if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module that is tested in another module.

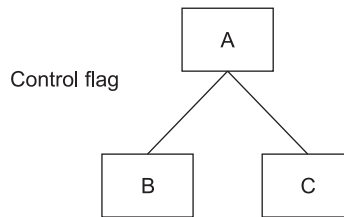


FIGURE 5.14 Control Coupling

The sending module must know a great deal about the inner workings of the receiving module. A variable that controls decisions in subordinate module C is set in super-ordinate module A and then passed to C.

4. **External Coupling.** It occurs when modules are tied to an environment external to software. External coupling is essential but should be limited to a small number of modules with structures.
5. **Common Coupling.** Two modules are common coupled if they share some global data items (e.g., Global variables). Diagnosing problems in structures with considerable common coupling is time-consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common couplings and take special care to guard against them.
6. **Content Coupling.** Content coupling exists between two modules if their code is shared; for example, a branch from one module into another module. It is when one module directly refers to the inner workings of another module. Modules are highly interdependent on each other. It is the highest form of coupling. It is also the least desirable coupling as one component actually modifies another and thereby the modified component is completely dependent on the modifying one.

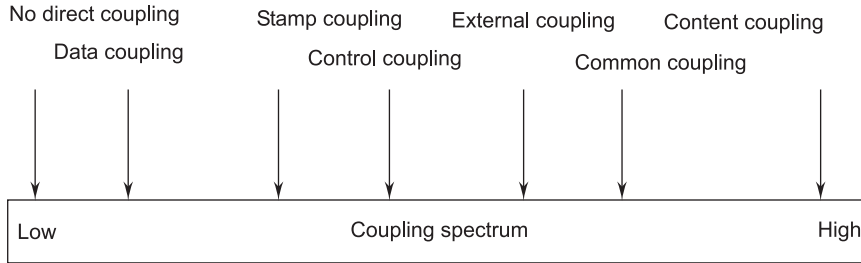


FIGURE 5.15 Couplings

High coupling among modules not only makes a design difficult to understand and maintain, but it also increases development effort as the modules having high coupling cannot be developed independently by different team members. Modules having high coupling are difficult to implement and debug.

5.4.2 Cohesion

Cohesion is a measure of the relative functional strength of a module. The cohesion of a component is a measure of the closeness of the relationships between its components. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.

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 Figure 5.16. Cohesion may be viewed as the glue that keeps the module together. It is a measure of the mutual officio of the components of a module.

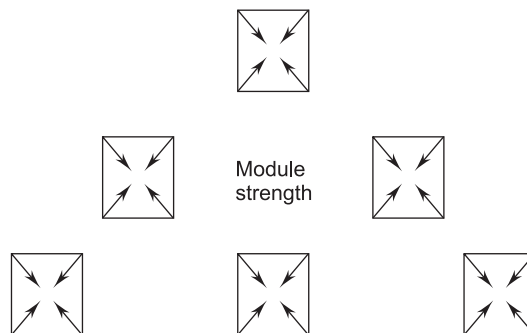


FIGURE 5.16 Cohesion-strength of Relation within Modules

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

Functional Cohesion	Best (high)
Sequential Cohesion	↑
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

FIGURE 5.17 The Types of Module Cohesion

There are seven levels of cohesion in decreasing order of desirability, which are as follows:

1. **Functional Cohesion.** Functional cohesion is said to exist if different elements of a module cooperate to achieve a single function (e.g., managing an employee's payroll). When a module displays functional cohesion, and if we are asked to describe what the module does, we can describe it using a single sentence.

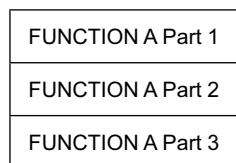


FIGURE 5.18 Functional Cohesion: Sequential with Complete, Related Functions

2. **Sequential Cohesion.** A module is said to possess sequential cohesion if the elements of a module form the parts of a sequence, where the output from one element of the sequence is input to the next.

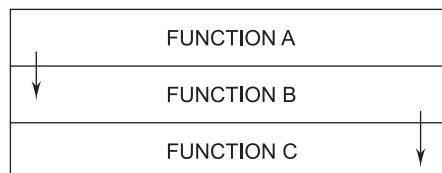


FIGURE 5.19 Sequential Cohesion: Output of One Part is Input to Next

3. **Communicational Cohesion.** A module is said to have communicational cohesion if all the functions of the module refer to or update the same data

structure; for example, the set of functions defined on an array or a stack. All the modules in communicational cohesion are bound tightly because they operate on the same input or output data. For example, the set of functions defined on an array or a stack.

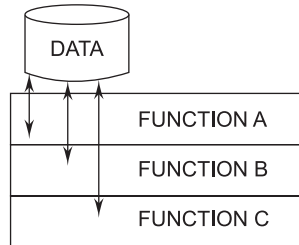


FIGURE 5.20 Communicational Cohesion: Access Same Data

4. **Procedural Cohesion.** A module is said to possess procedural cohesion if the set of functions of the module are all part of a procedure (algorithm) in which a certain sequence of steps has to be carried out for achieving an objective; for example, the algorithm for decoding a message.

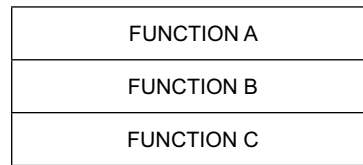


FIGURE 5.21 Procedural Cohesion Related by Order of Function

5. **Temporal Cohesion.** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc., exhibit temporal cohesion.

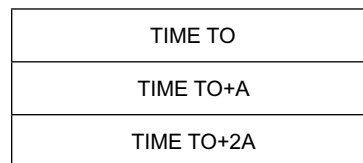


FIGURE 5.22 Temporal Cohesion Related by Time

6. **Logical Cohesion.** A module is said to be logically cohesive if all elements of the module perform similar operations; for example, error handling, data input, data output, etc. An example of logical cohesion is the case where a

set of print functions generating different output reports are arranged into a single module.

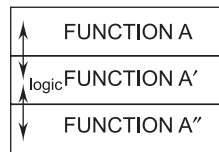


FIGURE 5.23 Logical Cohesion Similar Functions

7. **Coincidental Cohesion.** A module is said to have coincidental cohesion if it performs a set of tasks that relate to each other very loosely. In this case, the module contains a random collection of functions. It means that the functions have been put in the module out of pure coincidence without any thought or design. It is the worst type of cohesion.

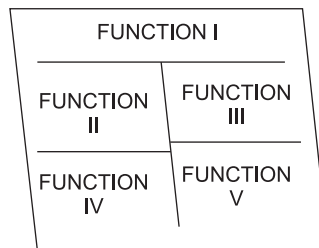


FIGURE 5.24 Coincidental Cohesion Parts Unrelated

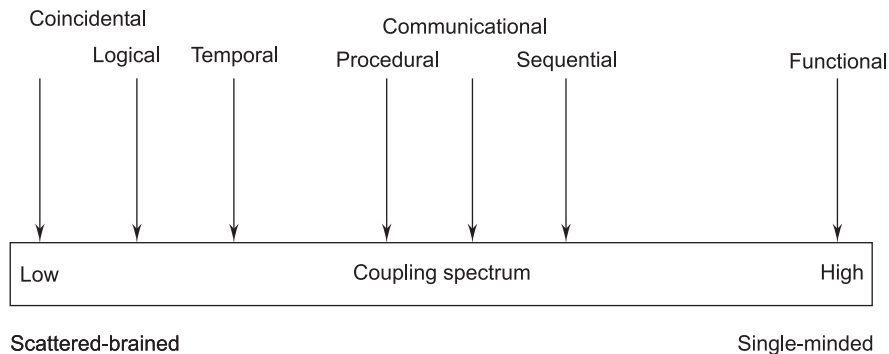


FIGURE 5.25 Cohesion

5.4.3 Relationship Between Coupling and Cohesion

A software engineer must design the modules with the 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 a computer system. Various slots in the motherboard 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 a highly cohesive manner. Figure 5.26 provides a graphical review of cohesion and coupling.

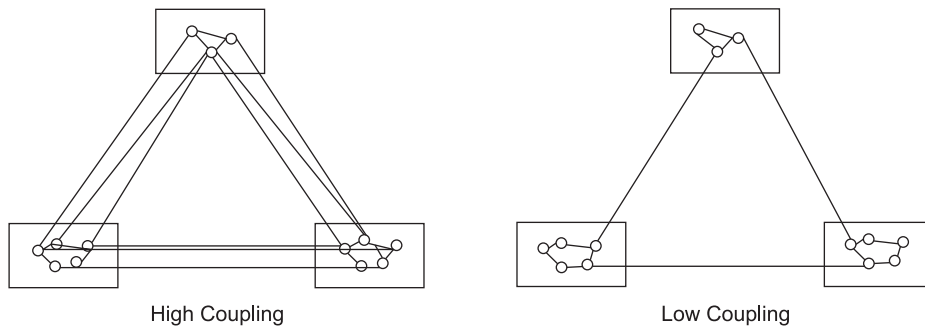


FIGURE 5.26 View of Cohesion and Coupling

Module design with high cohesion and low coupling characterizes a module as a black box when the entire structure of the system is described. Each module can be dealt with separately when the module functionality is described.

5.5 FUNCTIONAL-ORIENTED VERSUS THE OBJECT-ORIENTED APPROACH

Some of the differences between the functional-oriented and the object-oriented approaches, which are very indispensable, are described in Table 5.2.

TABLE 5.2

S. No.	Functional-oriented Approach	Object-oriented Approach
1.	In the functional-oriented design approach, the basic abstractions, which are given to the user, are real-world functions, such as sort, merge, track, display, etc.	In the object-oriented design approach, the basic abstractions are not the real-world functions, but are the data abstraction where the real-world entities are represented, such as picture, machine, radar system, customer, student, employee, etc.

2.	In function-oriented design, functions are grouped together by which a higher-level function is obtained. An example of this technique is SA/SD.	In this design, the functions are grouped together on the basis of the data they operate on, such as in class person, function displays are made member functions to operate on its data members such as the person name, age, etc.
3.	In this approach, the state information is often represented in a centralized shared memory.	In this approach, the state information is not represented in a centralized shared memory but is implemented/distributed among the objects of the system.

5.6 DESIGN SPECIFICATIONS

Design specifications address different aspects of the design model and are completed as the designer refines his representation of the software. First, the overall scope of the design effort is described, which is derived from system specification and the analysis model (software requirements specification).

Then, data design is specified, which includes data structures, any external file structures, internal data structures, and a cross-reference that connects data objects to specific files.

Then architectural design indicates how the program architecture has been derived from the analysis model. Structure charts are used to represent the module hierarchy.

Interface design indicates the design of external and internal program interfaces along with a detailed design of the human/machine interface. A detailed prototype of a GUI may also be represented.

Procedural design specifies components—separately addressable elements of software—such as subroutines, functions, or procedures in the form of English-language processing narratives. This narrative explains the procedural function of a component (module).

Design specification contains a requirements cross-reference. The purpose of this cross-reference is:

- To establish that all requirements are satisfied by the software design.
- To indicate which components are critical to the implementation of specific requirements.

The final section of the design specification contains supplementary data, such as algorithm descriptions, alternative procedures, tabular data, excerpts from

other documents, and other relevant information presented as a special note or a separate appendix.

TABLE 5.3

System objective	Human-machine interface
Major software requirements design	Specification and design
Constraints, limitations	External interface design
Data design	Interfaces to external/systems
Data objects and resultant data structures	Internal design rules
File and database structures	Processing narrative
External file structures	Interface description
Logical structures	Design language description
	Modules used
Access method	Data structures used
Global data	Comments
File and data cross-reference	Requirements cross-reference

5.7 VERIFICATION FOR DESIGN

The output of the system design phase, such as the output of other phases in the development process, should be verified before proceeding with the activities of the next phase. If the design is expressed in some formal notation for which analysis tools are available, then through tools it can be checked for internal consistency (e.g., those modules used by another are defined, the interface of a module is consistent with the way others use it, data usage is consistent with declaration, etc.). If the design is not specified in a formal, executable language, it cannot be processed through tools, and other means for verification have to be used.

There are two fundamental approaches to verification. The first consists of experimenting with the behavior of a product to see whether the product performs as expected (i.e., testing the product). The other consists of analyzing the product—or any design documentation related to it—to deduce its correct operation as a logical consequence of the design decisions. The two categories of verification techniques are also classified as dynamic or static, since the former requires—by definition—executing the system to be verified, while the latter does not. Not surprisingly, the two techniques turn out to be nicely complementary.

5.8 MONITORING AND CONTROL FOR DESIGN

Software project management is crucial to the success of a project. The basic task is to plan the detailed implementation of the development process to ensure that the cost and quality objectives are met. It specifies what is needed to meet the cost, quality, and schedule objectives. For this purpose we need monitoring and control. Monitoring obtains information from the development process and exerts the required control over it. Figure 5.27 shows where monitoring and control is carried out in project management:

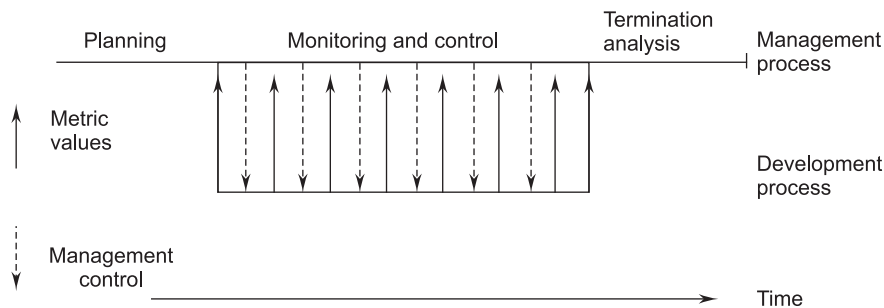


FIGURE 5.27 Phases of Project Management

Monitoring and control systems are an important class of a real-time system. They check sensors and provide information about the system's environment and take actions depending on the sensor reading. Monitoring systems take action when some exceptional sensor value is detected. Control systems continuously control hardware actuators depending on the value of associated sensors.

Consider the following example: A burglar alarm system is to be implemented for a building. This uses several different types of sensors. These include movement detectors in individual rooms, window sensors on ground floor windows, which detect if a window has been broken, and door sensors, which detect a door opening on corridor doors. There are 50 window sensors, 30 door sensors, and 200 movement detectors in the system.

When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The alarm system is normally powered by the main power source but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the main voltage. It interrupts the alarm system when a voltage drop is detected.

EXERCISES

1. What is system design?
2. Explain, in detail, the three design principles in system design.
3. What is abstraction? What are the verification metrics for system design?
4. Define:
 - (i) Problem partitioning
 - (ii) Abstraction
 - (iii) Top-down and bottom-up design
5. Define architectural design.
6. What are the objectives of architectural design?
7. Explain the various design techniques that come under the category of low-level design.
8. Define:
 - (i) Modularization
 - (ii) Structure charts
 - (iii) Pseudo-code
 - (iv) Flowcharts
9. Give any two important differences between the function-oriented and object-oriented design approaches.
10. Discuss the major advantages of the object-oriented design approach over the function-oriented design approach.
11. What is a flowchart? Explain some of its symbols. Also give a suitable example.
12. Give the hierarchical format of a structure chart. Also, give the basic building blocks of a structure chart.
13. Explain the term design specification.
14. Discuss the term verification in reference to system design.
15. Enumerate the term monitoring and control in system design.
16. Discuss some methods of monitoring and control of a software-development process.
17. What is meant by the term coupling in software design? Is it true that in a good design, the modules should have low coupling? Why?
18. Explain the different types of coupling that two modules might exhibit.
19. Explain the different types of cohesion that a module might exhibit.
20. What is coupling and cohesion in reference to software design? How are these concepts useful in arriving at a good design of a system?
21. Is it true that whenever we increase the cohesion of different modules in our design, coupling between these modules automatically decreases? Justify your answer with the help of an appropriate example.
22. What is a flowchart? How is the flow-charting technique useful for software development?
23. Discuss the major advantages of the object-oriented design (OOD) methodology over the data flow-oriented design methodologies.

Chapter 6

SOFTWARE MEASUREMENT AND METRICS

6.1 SOFTWARE METRICS

Software metrics are quantifiable measures that could be used to measure different characteristics of a software system or the software-development process.

Metrics and measurements are necessary aspects of managing a software-development project. For effective monitoring, management needs to get information about the project: how far it has progressed, how much development has taken place, how far behind schedule it is, and the quality of the development so far. Based on this information, decisions can be made about the project. Without proper metrics to quantify the required information, subjective opinion would have to be used, which is often unreliable and goes against the fundamental goals of engineering. Hence, we can say that metrics-based management is also a key component in the software-engineering strategy to achieve its objectives.

6.1.1 Definition

Software metrics can be defined as “*The continuous application of measurement-based techniques to the software-development process and its products to supply meaningful and*

timely management information, together with the use of those techniques to improve that process and its products."

Metrics ensure that the final product is of high quality and the productivity of the project stays high. In this sequence metrics for intermediate products of requirements and design is to predict or get some idea about the metrics of the final product. Several metrics have been designed for coding; namely, size, complexity, style, and reliability.

6.1.2 Categories of Metrics

There are three categories of software metrics, which are as follows:

1. **Product Metrics.** Product metrics describe the characteristics of the product, such as size, complexity, design features, performance, efficiency, reliability, portability, etc.
2. **Process Metrics.** 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
3. **Project Metrics.** 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

6.1.3 Attributes of Effective Software Metrics

Hundreds of metrics have been proposed for computer software, but not all provide practical support to the software engineer. Some demand measurement that is too complex, others are so esoteric that few real-world professionals have any hope of understanding them, and others violate the basic intuitive notions of what high-quality software really is.

Ejiogu defines a set of attributes that should be encompassed by effective software metrics. The derived metric and the measures that lead to it should be:

- *Simple and computable*: It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time.
- *Empirically and intuitively persuasive*: The metrics should satisfy the engineer's intuitive notions about the product attribute under consideration.
- *Consistent and objective*: The metric should always yield results that are unambiguous.
- *Consistent in the use of units and dimensions*: The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units.
- *Programming-language independent*: Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- *An effective mechanism for high-quality feedback*: That is, the metric should lead to a higher-quality end product.

6.2 HALSTEAD'S SOFTWARE SCIENCE

Software science is an approach—based on Halstead's theories—measuring software qualities on the basis of objective code measures. It is based on information theory, which in turn is based on the following measurable quantities, defined for a given program coded in any programming language:

- η_1 , the number of unique, distinct operators appearing in the program;
- η_2 , the number of unique, distinct operands appearing in the program;
- N_1 , the total number of occurrences of operators in the program;
- N_2 , the total number of occurrences of operands in the program.

The terms “operator” and “operand” have intuitive meanings and are to be defined for different programming languages. The distinction is that the operator works on the operand.

Once we have defined η_1 and η_2 precisely, we can define the program vocabulary η and the program length N as

$$\eta = \eta_1 + \eta_2$$

and

$$N = N_1 + N_2.$$

At this point, we can already consider some relations between the above quantities. A first obvious relation is

$$\eta_1 \leq N.$$

Using these definitions, Halstead suggests the equation

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2,$$

where the “hat” has been placed on N to distinguish the quantity obtained, the calculated length, with this equation from the value of the length N obtained by direct observation.

We decided to quote Halstead’s reasoning directly since, in spite of its intuitive attractiveness, it suffers from flaws that make it questionable. The major flaw is that a program is not a “single ordered set of n [distinct!] elements,” but a string, with possible repetitions.

Disregarding this observation, \hat{N} is an estimate of the length of the program and not actual measured value of the length of the program. So, when we compare \hat{N} versus N on the same sample codes or programs our published algorithms shows $(N - \hat{N})/N$ is less than 10%.

Program level ‘ L ’ is a measure of the ‘Level of abstraction’ of the formulation of an algorithm. Another important measure Halstead’s theory introduces is the effort ‘ E ’ defined as

$$\text{Effort} = E = \frac{V}{L}.$$

Halstead argues that E may be interpreted as the number of mental discriminations required to implement a program and also as the effort required to read and understand the program.

In conclusion, Halstead’s theory tries to provide a formal definition for such qualitative, empirical, and subjective software qualities as program complexity, eases of understanding, and level of abstraction, based on the count of some low-level quantities, such as the number of operators and operands appearing in a program. The goal is to be able to predict the level of these qualities a program will have before the start of a project and then measure the level mechanically to assess the quality of the resulting product.

Example 6.1. Table 6.1 illustrates a simple Fortran routine and the associated values of N_1 , N_2 , n_1 , and n_2 . Halstead defines several quantities using these numbers. For example, program length N is defined as $N_1 + N_2$; ($N_1 + N_2 = 50$ in Table 6.1).

Halstead’s estimator of program length is:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2.$$

For example, in Table 6.1

$$\begin{aligned} N &= 10 (3.2) + 7 (2.8) \\ &= 52.9. \end{aligned}$$

Program volume is defined as

$$V = (N_1 + N_2) \log_2 (n_1 + n_2).$$

```

Subroutine sort(X, N)
Dimension X(N)
If (N, I.T. 2) RETURN
  Do 20 I = 2, N
  Do 10 J= 1, 1
  IF (X (1), GE X (J)) GOTO 10
    SAVE = X (I)
    X (I) = X (J)
    X (J) = SAVE
  10 continue
  20 continue
RETURN
END

```

TABLE 6.1 Operator and Operand Count for a Fortran Routine

	Operand	Count	Operator	Count
	1 X	6	1 End of statement	7
	2 I	5	2 Array subscript	6
	3 J	4	3 =	5
	4 N	2	4 Lf()	2
	5 2	2	5 DO	2
	6 save	2	6	2
	$n_2 = 7$ I	1	7 End of program	1
		$22 = N_2$	8. LT.	1
			3. GE.	1
			$n_1 = 10$ goto 10	1
				$28 = N_1$

and language level (the level of language abstraction) is

$$L = (2*n_2)/(n_1*N_2).$$

Program effort is defined as $\frac{V}{L}$;

$$E = (n_1*N_2*(N_1 + N_2)*\log_2 (n_1 + n_2)) / (2*n_2).$$

Program effort is interpreted to be the number of mental discriminations required to implement the program. Alternatively, it can be interpreted as the effort required for reading and understanding a program. Experiments have shown that Halstead's effort metric is well-correlated with the observed effort required to debug and modify small programs. Program effort thus appears to be a measure of interest for software maintenance.

6.3 FUNCTION-POINT BASED MEASURES

6.3.1 Function Points

Function points and feature points are methods of estimating the "amount of functionality" required for a program, and are thus used to estimate project completion time. The basic idea involves counting inputs, outputs, and other features of a description of functionality.

6.3.2 Function-Point Metric

The function-point metric was proposed by Allan Albrecht (1983) when he worked for IBM. Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since functionality cannot be measured directly, it must be derived indirectly using other direct measures. Function points are derived using an empirical relationship based on direct measures of the software's information domain and assessments of software complexity.

Function points are computed by completing Table 6.2.

TABLE 6.2 Function-Point Contribution of an Element

Function type	Simple	Average	Complex
External input	3	4	6
External output	4	5	7
Logical internal file	7	10	15
External interface file	5	7	10
External inquiry	3	4	6

The different function types mentioned in Table 6.2 are discussed as follows:

- External types are input transactions that update internal computer files.
- External output types are transactions where data is output to the user. Typically, these would be printed reports.

- Logical internal file types are the standing files used by the system. The term 'file' does not sit easily with modern information systems. It refers to a group of data that is usually accessed together. It might be made up of one or more record types.
- External interface file types allow for output and input that might pass to and from other computer applications.
- External inquiry types note the U.S. spelling of inquiry and are transactions initiated by the user that provide information but do not update the internal files. The user inputs some information that directs the system to the details required.

6.3.3 Special Features

- The function-point approach is independent of the language, tools, or methodologies used for implementation, i.e., they do not take into consideration programming languages, database-management systems, processing hardware, or any other database technology.
- Function points can be estimated from requirement specifications 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 the estimate.
- 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.

6.3.4 Advantages of Function Points

- Are not restricted to code.
- Are language independent.
- The necessary data is available early in a project and thus only a detailed specification is needed.
- Are more accurate than estimated LOC.
- Can be used to easily estimate the size of a software product directly from the problem specification.

6.3.5 Drawbacks of Function Points

- Subjective counting—different people can come up with different estimates for the same problem.

- Hard to automate and difficult to compute.
- Ignore quality of output.
- Oriented to traditional data-processing applications.

6.3.6 Feature-Point Metrics

A function-point extension called feature points is a superset of the function-point measure that can be applied to systems and engineering software applications. Therefore, its size should be larger compared to simpler functions.

Proponents of function-point and feature-point metrics claim that these metrics are language-independent and can be easily computed from the SRS document during project planning, whereas opponents claim that these metrics are subjective and require a slight of hand. An example of the subjective nature of the function-point metric can be that the way one would group logically related data items could be very subjective. For example, if a data employee details consist of the employee name and his address, one person can consider it a single unit of data while someone else can consider the address as one unit and name as another. Therefore, different engineers can arrive at different function-point measures for the same problem.

Example 6.2. *Compute the function-point value for a project with the following information-domain characteristics.*

Number of user Inputs: 32

Number of User output: 60

Number of User Inquiries: 24

Number of files: 8

Number of external interface: 2

Assume that all complexity adjustment values are average.

Solution.

Measurement Parameter	Count	Weighting Factor (Average)	
Number of user inputs	32	4	= 128
Number of user outputs	60	5	= 300
Number of user inquires	24	4	= 96
Number of files	8	10	= 80
Number of external interfaces	2	7	= 14
Count total			618

$$\begin{aligned}
 \text{Function point (FP)} &= \text{Count Total} \times (0.65 + 0.01 \times S(f_i)) \\
 \text{FP} &= 618 \times (0.65 + 0.01 \times 30) \\
 &= 618 \times (0.65 + 0.3) \\
 &= 618 \times (0.95) \\
 &= 587.10
 \end{aligned}$$

6.4 CYCLOMATIC COMPLEXITY

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path-testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program, and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

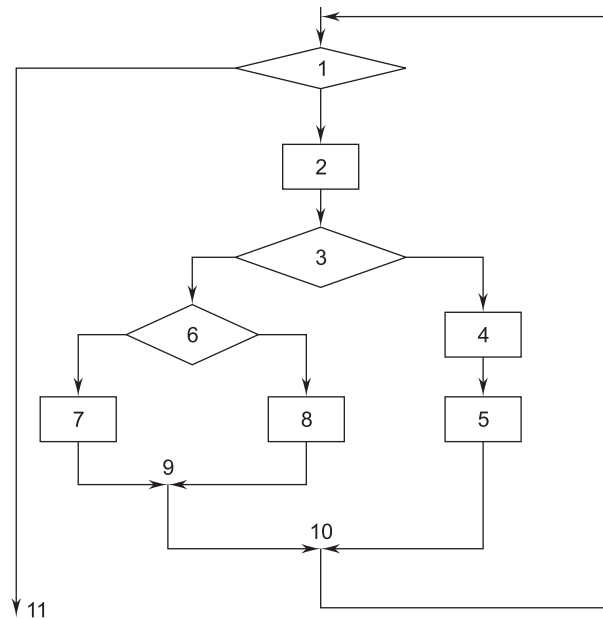


FIGURE 6.1 Flowchart

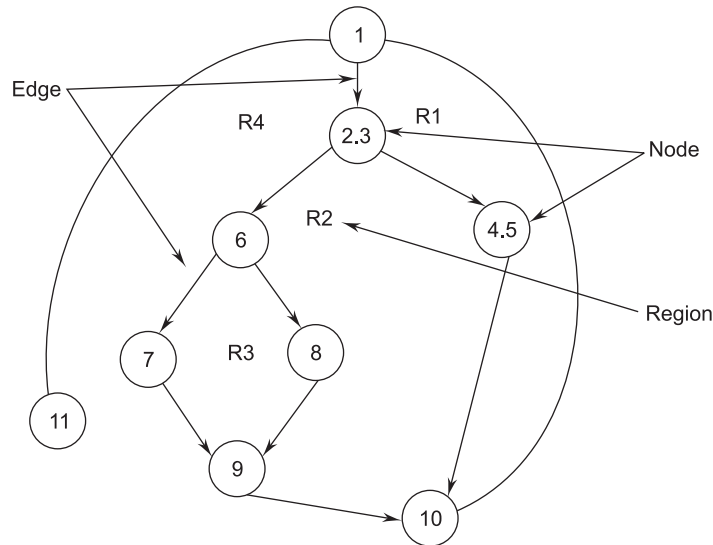


FIGURE 6.2 Flowgraph

An independent path must move along at least one edge that has not been traversed before the path is defined

Example 6.3. A set of independent paths for the flowgraph illustrated in Figure 6.2 is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Cyclomatic complexity has a foundation in graph theory and is computed in one of three ways:

1. The number of regions corresponds to the cyclomatic complexity.
2. Cyclomatic complexity, $V(G)$, for a flowgraph G , is defined as

$$V(G) = E - N + 2,$$

where

E = Number of flowgraph edges

N = Number of flowgraph nodes.

3. Cyclomatic complexity, $V(G)$ for a flowgraph G , is also defined as

$$V(G) = P + 1,$$

where

P = Number of predicate nodes contained in flow graphs G .

The cyclomatic complexity of Figure 6.2 (b) can be computed using each of the algorithms just noted.

- The flowgraph has four regions:

$$V(G) = 11 \text{ edges} - 9 \text{ nodes} + 2 = 4$$

$$V(G) = 3 \text{ predicate nodes} + 1 = 4.$$

Therefore, the cyclomatic complexity of the flowgraph shown is 4. The value $V(G)$ provides us with an upper bound for the number of independent paths that comprise the basis set, and by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

EXERCISES

1. Explain cyclomatic complexity with an example.
2. How can you compute the cyclomatic complexity of a program? How is cyclomatic complexity useful in program testing?
3. What is Halstead's size measure for two modules? Compare this size with the size measured in the LOC.
4. Consider the size measure as the number of bytes needed to store the object code of a program. How useful is this size measure? Is it closer to the LOC or Halstead metric? Explain clearly with the help of an example.
5. How can you correlate the complexity measurement with the size of a module?
6. Define:
 - (a) Product metrics
 - (b) Process metrics
 - (c) Project metrics
7. What are the different attributes of effective software metrics?
8. What is the difference between function-point metrics and feature-point metrics?
9. Explain the different types of software metrics.
10. Explain the special features of function-point metrics.

Chapter 7

SOFTWARE TESTING

7.1 INTRODUCTION TO TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defined for the software process.

A number of software-testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

- To perform effective testing, a software team should conduct effective formal technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developers of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source-code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy must provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems must surface as early as possible. Software testing has been defined as:

- The process of analyzing a software item to detect the differences between existing and required conditions (i.e., bugs) and to evaluate the features of the software items.
- The process of analyzing a program with the intent of finding errors.

OR

Testing is the process of executing a program with the intent of finding errors.

7.2 TESTING PRINCIPLES

There are many principles that guide software testing. Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. The following are the main principles for testing:

1. **All tests should be traceable to customer requirements.** This is in order to uncover any defects that might cause the program or system to fail to meet the client's requirements.
2. **Tests should be planned long before testing begins.** Soon after the requirements model is completed, test planning can begin. Detailed test cases can begin as soon as the design model is designed.
3. **The Pareto principle applies to software testing.** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
4. **Testing should begin "in the small" and progress toward testing "in the large."** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
5. **Exhaustive testing is not possible.** The number of path permutations for even a moderately-sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible,

however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

6. **To be most effective, testing should be conducted by an independent third party.** The software engineer who has created the system is not the best person to conduct all tests for the software. This is shown in Figure 7.1.

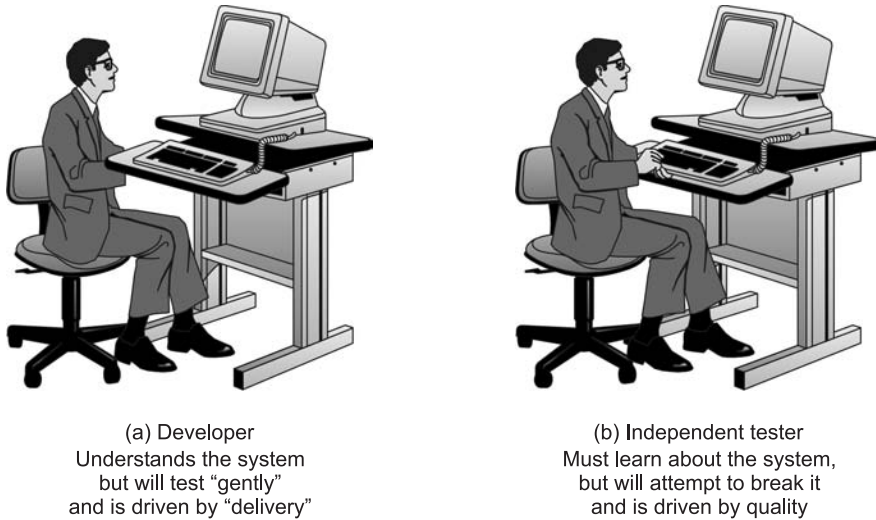


FIGURE 7.1 Who Tests the Software?

7.3 TESTING OBJECTIVES

The testing objective is to test the code, whereby there is a high probability of discovering all errors.

This objective also demonstrates that the software functions are working according to software requirements specification (SRS) with regard to functionality, features, facilities, and performance. It should be noted, however, that testing will detect errors in the written code, but it will not show an error if the code does not address a specific requirement stipulated in the SRS but not coded in the program.

Testing objectives are:

- Testing is a process of executing a program with the intent of finding an error.

- A good test case is one that has a high probability of finding an as-yet-undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

7.4 TEST ORACLES

To test any program, we need to have a description of its expected behavior and a method of determining whether the observed behavior conforms to the expected behavior. For this we need a test oracle.

A test oracle is a mechanism, different from the program itself, which can be used to check the correctness of the output of the program for the test cases. Conceptually, we can consider testing a process in which the test cases are given to the test oracle and the program under testing. The output of the two is then compared to determine if the program behaved correctly for the test cases, as shown in Figure 7.2.

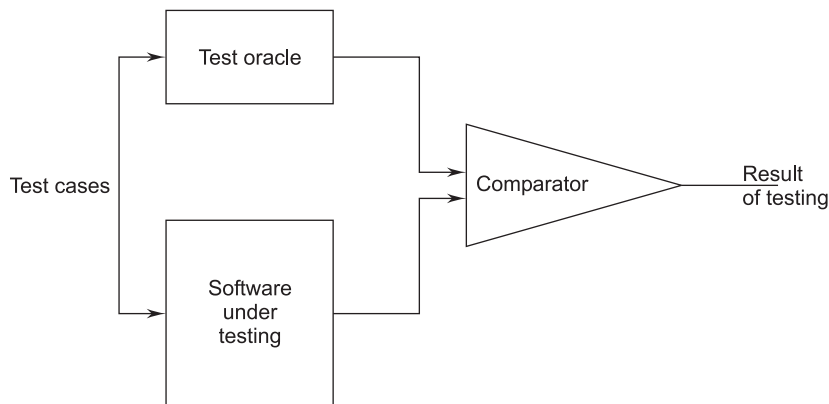


FIGURE 7.2 Test Oracles

Test oracles are human beings, so they may make mistakes when there is a discrepancy between the oracles and the results of a program. First we have to verify the result produced by the oracle before declaring that there is a fault in the program, that's why testing is so cumbersome and expensive.

The human oracles generally use the specifications of the program to decide what the "correct" behavior of the program should be. To help the oracle to determine the correct behavior, it is important that the behavior of the system be unambiguously specified and the specification itself should be error-free.

7.5 LEVELS OF TESTING

There are three levels of testing, i.e., three individual modules in the entire software system.

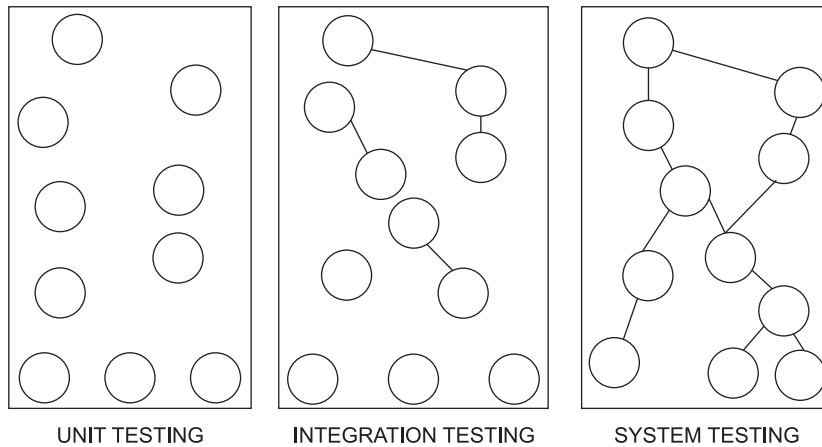


FIGURE 7.3 Levels of Testing

7.5.1 Unit Testing

In unit testing individual components are tested to ensure that they operate correctly. It focuses on verification effort. On the smallest unit of software design, each component is tested independently without other system components.

There are a number of reasons to do unit testing rather than testing the entire product:

- The size of a single module is small enough that we can locate an error fairly easily.
- The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
- Confusing interactions of multiple errors in widely different parts of the software are eliminated.

Unit Test Consideration

The tests that occur as part of unit tests are illustrated schematically in Figure 7.4. The module interface is tested to ensure that information properly flows into and out of the program unit under testing. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an

algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at the boundaries established to limit or restrict processing. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error-handling paths are tested.

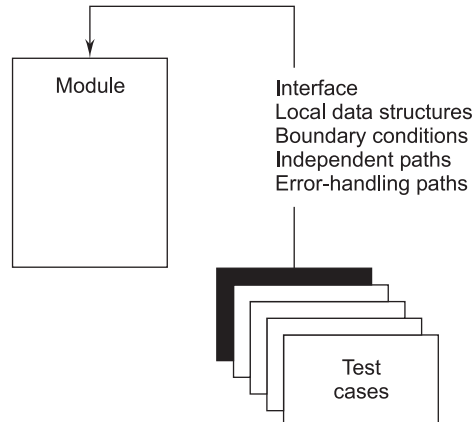


FIGURE 7.4 Unit Tests

Common errors in computation are:

- Incorrect arithmetic precedence
- Mixed code operations
- Incorrect initialization
- Precision inaccuracy
- Incorrect symbolic representation of an expression

Test cases in unit testing should uncover errors, such as:

- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely
- Incorrect comparison of variables
- Improper loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables

Unit-test Procedure

The unit-test environment is illustrated in Figure 7.5.

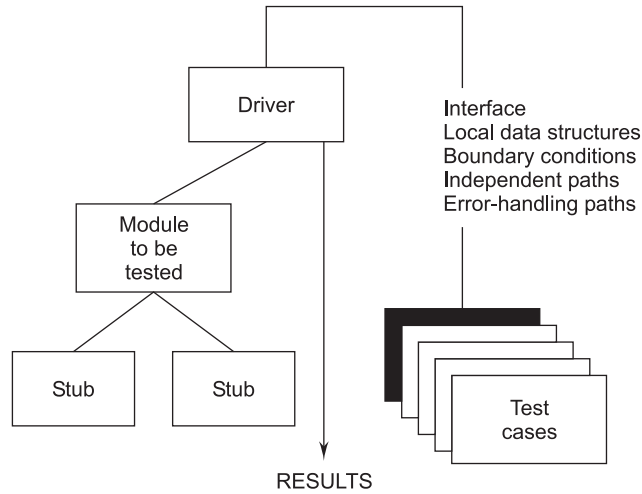


FIGURE 7.5 Unit-test Environment

In most applications a driver is nothing more than a “main program” that accepts test-case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (called by) to the component to be tested. A stub uses the subordinate module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with “simple” overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

7.5.2 Integration Testing

The second level of testing is called integration testing. Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.

In this testing many unit-tested modules are combined into subsystems, which are then tested. The goal here is to see if the modules can be integrated properly.

Objective of Integration Testing

The primary objective of integration testing is to test the module interfaces in order to ensure that there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested.

Approaches to Integration Testing

The various approaches used for integration testing are:

1. Incremental Approach
2. Top-down Integration
3. Bottom-up Integration
4. Regression Testing
5. Smoke Testing
6. Sandwich Integration

These approaches to integration testing are discussed as follows:

1. **Incremental Approach.** The incremental approach means to first combine only two components together and test them. Remove the errors if they are there, otherwise combine another component to it and then test again, and so on until the whole system is developed.

OR

In incremental integration the program is constructed and tested in small increments, where errors are easier to isolate and correct.

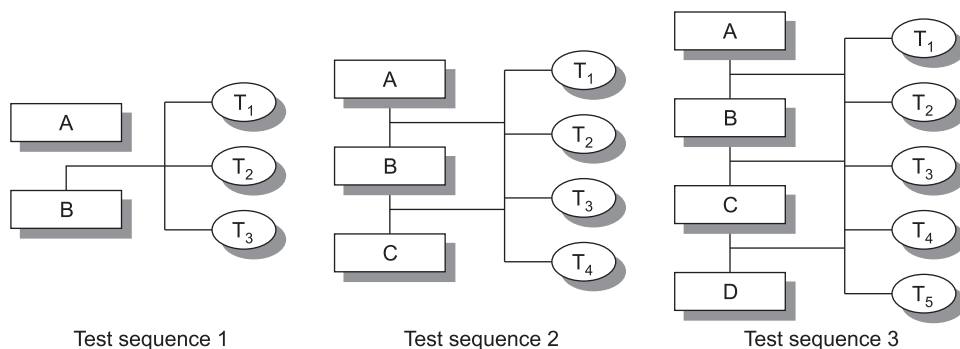


FIGURE 7.6 Incremental Approach

According to Figure 7.6, in test sequence 1 tests T_1 , T_2 , and T_3 are first run on a system composed of module A and module B. If these are corrected or error-free then module C is integrated, i.e., test sequence 2 and then tests T_1 , T_2 , and T_3 are repeated, if a problem arises in these tests, then they interact with the

new module. The source of the problem is localized, thus it simplifies defect location and repair. Finally, module D is integrated, i.e., test sequence 3 is then tested using existing (T_1 to T_5) and new tests (T_6).

2. **Top-Down Integration Testing.** Top-down integration testing is an incremental approach to construction of program structures. Modules are integrated by moving downward through the control hierarchy beginning with the main-control module.

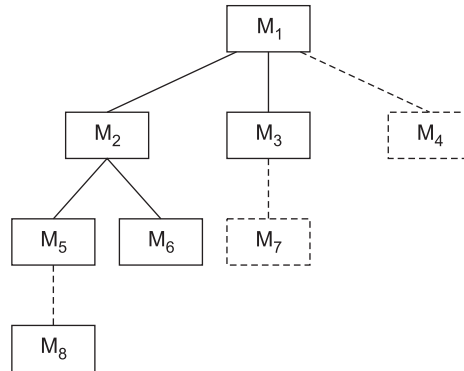


FIGURE 7.7 Top-Down Integration Testing

Considering Figure 7.7:

- (i) *Depth-first integration:* This would integrate all components on a major control path of the structure. For example, M_1 , M_2 , and M_5 would be integrated first. Next, M_8 or M_6 would be integrated. Then, the central and right-hand control paths are built.
- (ii) *Breadth-first integration:* This incorporates all components directly subordinate at each level, moving across the structure horizontally. From Figure 7.7, components M_2 , M_3 , and M_4 would be integrated first. The next control level M_5 , M_6 , and so on follows.

The integration process is performed in a series of five steps:

- The main-control module is used as a test driver and stubs are substituted for all components directly subordinate to the main-control module.
- Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
- Tests are conducted as each component is integrated.

- On completion of each set of tests, another stub is replaced with the real component.
 - Regression testing may be conducted to ensure that new errors have not been introduced.
3. **Bottom-Up Integration Testing.** Bottom-up integration testing, as its name implies, begins construction and testing with the components at the lowest level in the program structure.

A bottom-up integration strategy may be implemented with the following steps:

- Low-level components are combined into clusters (sometimes called builds) that perform specific software subfunctions.
- A driver (a control program for testing) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program stucture.

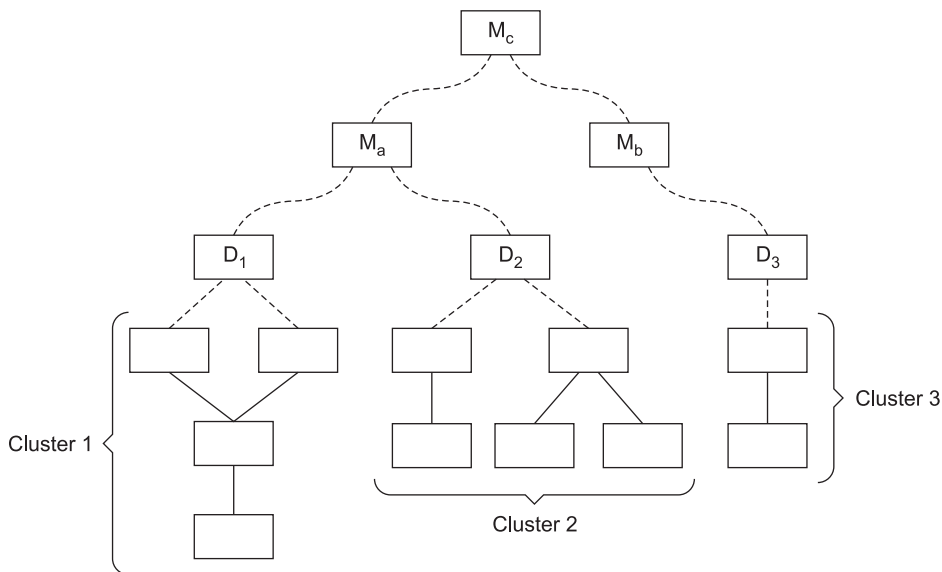


FIGURE 7.8 Bottom-Up Integration Testing

Integration follows the pattern illustrated in Figure 7.8. Components are combined to form clusters 1, 2, 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a . Drivers D_1 and D_2 are removed and the clusters are interfaced directly

to M_a . Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b . Both M_a and M_b will ultimately be integrated with component M_c and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top-down, the number of drivers can be reduced and integration of clusters is greatly simplified.

4. **Regression Testing.** Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

The regression test suite contains three different classes of test cases:

- Additional tests that focus on software functions.
 - A representative sample of tests that will exercise all software functions.
 - Tests that focus on the software components that have been changed.
5. **Smoke Testing.** Smoke testing is an integration testing approach that is commonly used when “shrink-wrapped” software products are developed. Smoke testing is characterized as a rolling integration approach because the software is rebuilt with new components and testing. Smoke testing encompasses the following activities:
 - Software components that have been translated into code are integrated into a “build.” A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
 - A series of tests is designed to expose errors that will keep the build from properly performing its functions.
 - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.

Smoke testing provides a number of benefits when it is applied on complex software engineering projects:

- Integration risk is minimized.
 - Quality of end product is improved.
 - Error diagnosis and correction are simplified.
 - Progress is easier to assess.
6. **Sandwich Integration Testing.** Sandwich integration testing is the combination of both the top-down and bottom-up approach. So, it is also called mixed integration testing. In it, the whole system is divided into three layers, just like a sandwich: the target is in the middle and one layer is above the target and one is below the target.

The top-down approach is used in the layer that is above the target and the bottom-up approach is used in the layer that is below the target. Testing coverage on the middle layer, chosen on the basis of the structure of the component hierarchy and system characteristics, also combines the advantages of both the top-down and bottom-up approach. It also requires drivers. The ability to plan and control the sequence is hard in it. At the beginning the work parallelism is medium. Stubs are also needed. The ability to test particular paths is medium.

7.5.3 System Testing

Subsystems are integrated to make up the entire system. The testing process is concerned with finding errors that result from unanticipated interactions between subsystems and system components. It is also concerned with validating that the system meets its functional and non-functional requirements.

There are essentially three main kinds of system testing:

- Alpha testing
- Beta testing
- Acceptance testing

1. **Alpha Testing.** *Alpha testing refers to the system testing carried out by the test team within the development organization.*

The alpha test is conducted at the developer's site by the customer under the project team's guidance. In this test, users test the software on the development platform and point out errors for correction. However, the alpha test, because a few users on the development platform conduct it, has limited ability to expose errors and correct them. Alpha tests are conducted in a controlled environment. It is a simulation of real-life usage. Once the alpha test is complete, the software product is ready for transition to the customer site for implementation and development.

2. **Beta Testing.** *Beta testing is the system testing performed by a selected group of friendly customers.*

If the system is complex, the software is not taken for implementation directly. It is installed and all users are asked to use the software in testing mode; this is not live usage. This is called the beta test. Beta tests are conducted at the customer site in an environment where the software is exposed to a number of users. The developer may or may not be present while the software is in use. So, beta testing is a real-life software experience without actual implementation. In this test, end users record their observations, mistakes, errors, and so on and report them periodically.

In a beta test, the user may suggest a modification, a major change, or a deviation. The development has to examine the proposed change and put it into the change management system for a smooth change from just developed software to a revised, better software. It is standard practice to put all such changes in subsequent version releases.

3. **Acceptance Testing.** *Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system.*

When customer software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than the software engineers, an acceptance test can range from an informal 'test drive' to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

7.6 WHITE-BOX TESTING/STRUCTURAL TESTING

A complementary approach to functional or black-box testing is called structural or white-box testing. In this approach, test groups must have complete knowledge of the internal structure of the software. We can say structural testing is an approach to testing where the tests are derived from knowledge of the software's structure and implementation. Structural testing is usually applied to relatively small program units, such as subroutines, or the operations associated with an object. As the name implies, the tester can analyze the code and use knowledge about the structure of a component to derive test data. The analysis of the code can be used to find out how many test cases are needed to guarantee that all of the statements in the program are executed at least once during the testing process. It would not be advisable to release software that contains untested statements as the consequence 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.

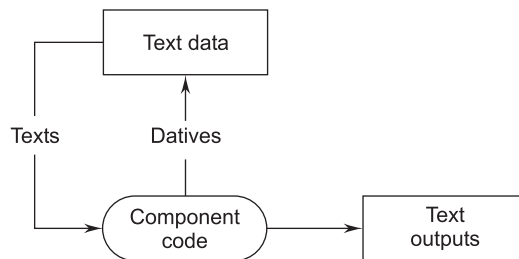


FIGURE 7.9 Structural Testing

White-box testing is also known by other names, such as glass-box testing, structural testing, clear-box testing, open-box testing, logic-driven testing, and path-oriented testing.

In white-box testing, test cases are selected on the basis of examination of the code, rather than the specifications. White-box testing is illustrated in Figure 7.10.

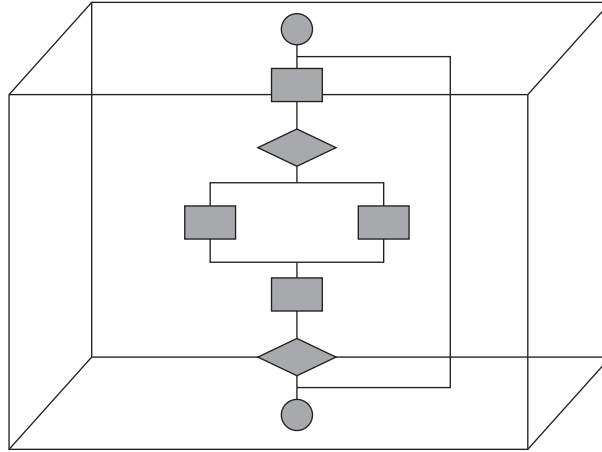


FIGURE 7.10 White-box Testing

Using white-box testing methods the software engineer can test cases that:

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decision on their true and false sides.
- Exercise all loops at their boundaries.
- Exercise internal data structures to ensure their validity.

The nature of software defects are:

- Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- We often believe that a logical path is not to be executed when, in fact, it may be executed on a regular basis.
- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur.

7.6.1 Reasons White-box Testing is Performed

White-box testing is carried out to test whether:

- All paths in a process are correctly operational.
- All logical decisions are executed with true and false conditions.
- All loops are executed with their limit values tested.
- To ascertain whether input data structure specifications are tested and then used for other processing.

7.6.2 Advantages of Structural/White-box Testing

The various advantages of white-box testing include:

- Forces test developer to reason carefully about implementation.
- Approximates the partitioning done by execution equivalence.
- Reveals errors in hidden code.

7.7 FUNCTIONAL/BLACK-BOX TESTING

In functional testing the structure of the program is not considered. Test cases are decided on the basis of the requirements or specifications of the program or module and the internals of the module or the program are not considered for selection of test cases.

Functional testing refers to testing that involves only observation of the output for certain input values, and there is no attempt to analyze the code, which produces the output. The internal structure of the program is ignored. For this reason, functional testing is sometimes referred to as black-box testing (also called behavioral testing) in which the content of a black-box is not known and the function of black box is understood completely in terms of its inputs and outputs.

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. Black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

Other names for black-box testing (BBT) include specifications testing, behavioral testing, data-driven testing, functional testing, and input/output-driven testing.

In black-box testing, the tester only knows the inputs that can be given to the system and what output the system should give. In other words, the basis for deciding test cases in functional testing is the requirements or specifications of the system or module. This form of testing is also called functional or behavioral testing.

Black-box testing is not an alternative to white-box techniques; rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing identifies the following kinds of errors:

- Incorrect or missing functions.
- Interface missing or erroneous.
- Errors in data model.
- Errors in access to external data source.

When these errors are controlled then:

- Function(s) are valid.
- A class of inputs is validated.
- Validity is sensitive to certain input values.
- The software is valid and dependable for a certain volume of data or transactions.
- Rare specific combinations are taken care of.

Black-box testing tries to answer the following questions:

- How is functional validity tested?
- How are system behavior and performance tested?
- How are the boundaries of a data class isolated?
- How will the specific combinations of data affect system operation?
- What data rates and data volume can the system tolerate?
- Is the system particularly sensitive to certain input values?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria:

- Test cases that reduce by a count that is greater than one.
- Test cases that tell us something about the presence or absence of classes of errors.

7.7.1 Categories of Black-box Testing

Functional testing falls into two categories:

1. **Positive Functional Testing:** This testing entails exercising the application's functions with valid input and verifying that the outputs are correct.

Continuing with the word-processing example, a positive test for the printing function might be to print a document containing both text and graphics to a

printer that is online, filled with paper and for which the correct drivers are installed.

2. **Negative Functional Testing:** This testing involves exercising application functionality using a combination of invalid inputs, unexpected operating conditions, and other out-of-bounds scenarios.
 - (i) Continuing the word-processing example, a negative test for the printing function might be to disconnect the printer from the computer while a document is printing.
 - (ii) What probably should happen in this scenario is a plain-English error message appears, informing the user what happened, and instructing him on how to remedy the problem.
 - (iii) What might happen instead is the word-processing software simply hangs up or crashes because the “abnormal” loss of communications with the printer is not handled properly.

7.7.2 Example of Black-box Testing

Figure 7.11 illustrates the model of a system, which is assumed in black-box testing. This approach is equally applicable to systems that are organized as functions or as objects. The tester presents inputs to the component or the system and examines the corresponding outputs. If the outputs are not those predicted then the test has successfully detected a problem with the software.

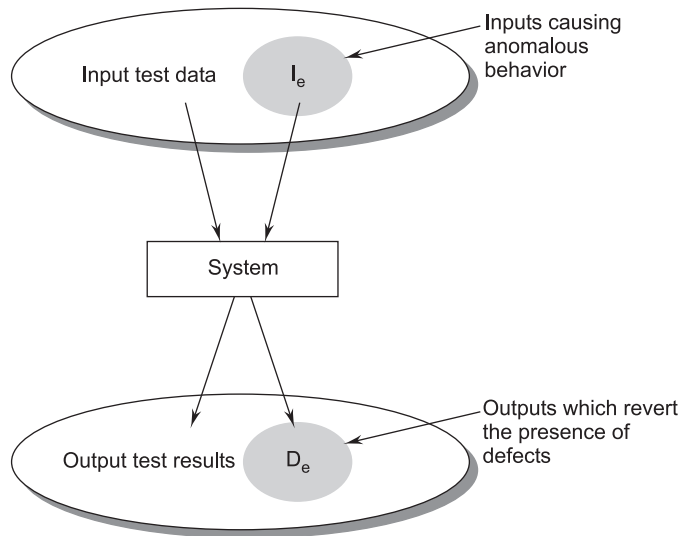


FIGURE 7.11 Black-box Testing

The key problem for the defect tester is to select inputs that have a high probability of being members of the set I_e . In many cases, the selection of these test cases is based on the previous experience of test engineers. They use domain knowledge to identify test cases, which are likely to reveal defects. However, the systematic approach to test data selection discussed in the next section may also be used to supplement this heuristic knowledge.

7.7.3 Advantages of Black-box Testing

The advantages of this type of testing include:

- The test is unbiased because the designer and the tester are independent of each other.
- The tester does not need knowledge of any specific programming languages.
- The test is done from the point-of-view of the user, not the designer.
- Test cases can be designed as soon as the specifications are complete.

7.8 TEST PLAN

A test plan is a document consisting of different test cases designed for different testing objects and different testing attributes. The plan puts the tests in logical and sequential order per the strategy chosen, top-down or bottom-up. The test plan is a matrix of test and test cases listed in order of its execution.

Table 7.1 shows the matrix of test and test cases within the test.

Project Name Project ID

Project Manager Q.A. Manager

TABLE 7.1 Test Plan

Test								
Test ID	Test	1	2	3	...	N	Planned Date	
ID	Tester	Name					Completed	Successful

Test ID, test name, and test cases are designed well before the development phase and have been designed for those who conduct the tests.

A test plan states:

- The items to be tested.
- At what level they will be tested at.
- The sequence they are to be tested in.
- How the test strategy will be applied to the testing of each item and the test environment.

7.9 TEST-CASE DESIGN

A test case is a set of instructions designed to discover a particular type of error or defect in the software system by inducing a failure.

The goal of selected test cases is to ensure that there is no error in the program and if there is it then should be immediately depicted. Ideal test casement should contain all inputs to the program. This is often called exhaustive testing.

There are two criteria for the selection of test cases:

- Specifying a criterion for evaluating a set of test cases.
- Generating a set of test cases that satisfy a given criterion.

Each test case needs proper documentation, preferably in a fixed format. There are many formats; one format is suggested below:

Test case name	Test Case ID
Purpose of test	Testing object (unit, application, module, etc.)
Test attribute	
Tests focus (function, feature, process, interface, validation, verification, etc.)	
Test type (alpha, beta, unit, integration, system)	
Test process	A set of instructions for conducting the test-initial stating condition-inputs-specifications-output expected
Test results	Expected and actual and comparison, error description, post-process state
Action	Correction, authorization, and feedback through retest
Action to initialize the pre-test status	

EXERCISES

1. What is testing? Explain the different types of testing performed during software development.

2. Define the various principles of testing.
3. What are test oracles?
4. What are the different levels of testing? Explain.
5. Suppose a developed software has successfully passed all the three levels of testing, i.e., unit testing, integration testing, and system testing. Can we claim that the software is defect-free? Justify your answer.
6. What is stress testing? Why is stress testing applicable to only certain types of systems?
7. What is unit testing?
8. What is integration testing? Which types of defects are uncovered during integration testing?
9. What is regression testing? When is regression testing done? How is regression testing performed?
10. What is system testing? What are the different kinds of system testing that are usually performed on large software products?
11. What is the difference between black-box testing and white-box testing?
12. Do you agree with the statement: System testing can be considered a pure black-box test? Justify your answer.
13. What are drivers and stub modules in the context of unit testing of a software product?
14. Define sandwich testing.
15. Why is regression testing important? When is it used?
16. What is a test case? What is test-case design?
17. What is the difference between
 - (a) Verification and validation
 - (b) Black-box testing and white-box testing
 - (c) Top-down and bottom-up testing approaches
 - (d) Alpha and beta testing
18. What are test plans and test cases? Illustrate each with an example.
19. Why does software testing need extensive planning? Explain.
20. What is smoke testing?
21. Differentiate between integration testing and system testing.
22. Define structural testing. Give the various reasons structural testing is performed.
23. Explain the two categories of black-box testing. Also state the advantages of black-box testing.

Chapter 8

SOFTWARE-TESTING STRATEGIES

8.1 STATIC-TESTING STRATEGIES

Static testing is the systematic examination of a program structure for the purpose of showing that certain properties are true regardless of the execution path the program may take. Consequently, some static analyses can be used to demonstrate the absence of some faults from a program. Static testing represents actual behavior with a model based upon the program's semantic features and structure. Human comparison often consists of people exercising little discipline in comparing their code against notions of intent that are only loosely and imprecisely defined. But human comparisons may also be quite structured, rigorous, and effective as is the case of inspections and walkthroughs, which are carefully defined and administered processes orchestrating groups of people to compare code and designs to careful specifications of intent. Static testing strategies include:

- Formal technical reviews
- Walkthroughs
- Code inspections
- Compliance with design and coding standards

8.1.1 Formal Technical Reviews

A review can be defined as:

A meeting at which the software element is presented to project personnel, managers, users, customers, or other interested parties for comment or approval.

What is a software review? A software review can be defined as a filter for the software-engineering process. The purpose of any review is to discover errors in the analysis, design, and coding, testing and implementation phases of the software-development cycle. The other purpose of a review is to see whether procedures are applied uniformly and in a manageable manner.

Objectives for Reviews

Review objectives are used:

- To ensure that the software elements conform to their specifications.
- To ensure that the development of the software element is being done as per plans, standards, and guidelines applicable for the project.
- To ensure that the changes to the software elements are properly implemented and affect only those system areas identified by the change specification.

Types of Reviews

Reviews are one of two types: informal technical reviews and formal technical reviews.

- **Informal Technical Review:** An informal meeting and informal desk checking.
- **Formal Technical Review:** A formal software quality assurance activity through various approaches, such as structured walkthroughs, inspections, etc.

What is a Formal Technical Review?

A formal technical review is a software quality assurance activity performed by software-engineering practitioners to improve software product quality. The product is scrutinized for completeness, correctness, consistency, technical feasibility, efficiency, and adherence to established standards and guidelines by the client organization.

The FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended.

Objectives of a Formal Technical Review

The various objectives of a formal technical review are as follows:

- To uncover errors in logic or implementation.
- To ensure that the software has been represented according to predefined standards.
- To ensure that the software under review meets the requirements.
- To make the project more manageable.

For the success of a formal technical review, the following are expected:

- The schedule of the meeting and its agenda reach the members well in advance.
- Members review the material and its distribution.
- The reviewer must review the material in advance.

The Review Meeting

The meeting should consist of two to five people and should be restricted to not more than two hours (preferably). The aim of the review is to review the product/work and the performance of people. When the product is ready, the producer (developer) informs the project leader about the completion of the product and requests for review. The project leader contacts the review leader for the review. The review leader asks the reviewer to perform an independent review of the product/work before the scheduled FTR.

Results of FTR

- Meeting decision
 1. Whether to accept the product/work without any modifications.
 2. Accept the product/work with certain changes.
 3. Reject the product/work due to error.
- Review summary report
 1. What was reviewed?
 2. Who reviewed it?
 3. Findings of the review.
 4. Conclusion.

8.1.2 Code Walk-throughs

A code walk-through is an informal analysis of code as a cooperative, organized activity by several participants. The analysis is based mainly on the game of

“playing the computer.” That is, participants select some test cases (the selection could have been done previously by a single participant) and simulate execution of the code by hand. This is the reason for the name walk-through: participants “walk through the code” or through any design notation.

In general, the following prescriptions are recommended:

- Everyone’s work should be reviewed on a scheduled basis.
- The number of people involved in the review should be small (three to five).
- The participants should receive written documentation from the designer a few days before the meeting.
- The meeting should last a predefined amount of time (a few hours).
- Discussion should be focused on the discovery of errors, not on fixing them, nor on proposing alternative design decisions.
- Key people in the meeting should be the designer, who presents and explains the rationale of the work, a moderator for the discussion, and a secretary, who is responsible for writing a report to be given to the designer at the end of the meeting.
- In order to foster cooperation and avoid the feeling that the designers are being evaluated, managers should not participate in the meeting.

8.1.3 Code Inspections

A code inspection, originally introduced by Fagan (1976) at IBM, is similar to a walk-through but is more formal. In Fagan’s experiment, three separate inspections were performed: one following design, but prior to implementation; one following implementation, but prior to unit testing; and one following unit testing. The inspection following unit testing was not considered to be cost effective in discovering errors; therefore, it is not recommended.

The organization aspects of code inspection are similar to those of code walk-through (i.e., the number of participants, duration of the meeting, psychological attitudes of the participants, etc., should be about the same), but there is a difference in goals.

In code inspection, the analysis is aimed explicitly at the discovery of commonly made errors. In such a case, it is useful to state beforehand the type of errors for which we are searching. For instance, consider the classical error of writing a procedure that modifies a formal parameter and calling the procedure with a constant value as the actual parameter.

The following is a list of some classical programming errors, which can be checked for during code inspection:

- Use of uninitialized variables
- Jumps into loops
- Non-terminating loops
- Incompatible assignments
- Array indices out of bounds
- Improper storage allocation and deallocation
- Mismatches between actual and formal parameters in procedure calls
- Use of incorrect logical operators or incorrect precedence among operators
- Improper modification of loop variables
- Comparison of equality of floating-point values, etc.

Checklist for Code Inspections

Inspections or reviews are more formal and conducted with the help of some kind of checklist. The steps in the inspections or reviews are:

- Is the number of actual parameters and formal parameters in agreement?
- Do the type attributes of actual and formal parameters match?
- Do the dimensional units of actual and formal parameters match?
- Are the number of attributes and ordering of arguments to built-in functions correct?
- Are constants passed as modifiable arguments?
- Are global variable definitions and usage consistent among modules?
- Application of a checklist specially prepared for the development plan, SRS, design and architecture
- Nothing observation: ok, not ok, with comments on mistake or inadequacy
- Repair-rework
- Checklists prepared to countercheck whether the subject entity is correct, consistent, and complete in meeting the objectives

8.1.4 Differences Between Walk-throughs and Inspections/Reviews

- The basic difference between the two is that a walk-through is less formal and has only a few steps, whereas inspections and reviews are more formal and logically sequential with many steps.
- Both processes are undertaken before actual development, and hence they are conducted on documents, such as a development plan, SOW, RDD and SRS,

design document, and broad WBS to examine their authenticity, completeness, correctness, and accuracy.

- Both are costly but the cost incurred is comparatively much lower than the cost of repair at a much later stage in the development cycle.
- Another difference between a walk-through and an inspection is that the former is less formal and quick; whereas inspection is more formal, takes more time, and is far more systematic.

8.2 DEBUGGING

8.2.1 Introduction/Definition

Debugging means identifying, locating, and correcting the bugs usually by running the program. It is an extensively used term in programming. These bugs are usually logical errors.

During the compilation phase the source files are accessed and if errors are found, then that file is edited and the corrections are posted in the file. After the errors have been detected and the corrections have been included in the source file, the file is recompiled. This detection of errors and removal of those errors is called debugging. The file is compiled again, so changes done last time get included in the object file also by itself. This process of compilation, debugging, and correction posting in the source file continues until all syntactical errors are removed completely. If a program is very large and complex, the more the program has to be corrected and compiled.

Successful compilation of the program means that now the program is following all the rules of the language and is ready to execute. All of the syntax errors of the program are indicated by the compiler at this stage.

8.2.2 Debugging Tactics/Categories

The various categories for debugging are:

- Brute-force debugging
- Backtracking
- Cause elimination
- Program slicing
- Fault-tree analysis

The various categories for debugging mentioned above are discussed as follows:

1. **Brute-force Debugging.** The programmer appends the print or write statement which, when executed, displays the value of a variable. The programmer may trace the value printed and locate the statement containing the error. Earlier when the time for execution was quite high, programmers had to use the core dumps. The core dumps are referred to as the static image of the memory and this may be scanned to identify the bug.
2. **Backtracking.** In this technique, the programmer backtracks from the place or statement which gives the error symptoms for the first time. From this place, all the statements are checked for possible cause of errors. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.
3. **Cause Elimination.** Cause elimination is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.
A list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, the data are refined in an attempt to isolate the bug.
4. **Program Slicing.** This technique is similar to backtracking. However, the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.
5. **Fault-tree Analysis.** Fault-tree analysis, a method originally developed for the U.S. Minuteman missile program, helps us to decompose the design and look for situations that might lead to failure. In this sense, the name is misleading; we are really analyzing failures, not faults, and looking for potential causes of those failures. We build fault trees that display the logical path from effect to cause. These trees are then used to support fault correction or tolerance, depending on the design strategy we have chosen.

8.2.3 Debugging Process

Debugging is not testing but always occurs as a consequence of testing. Referring to Figure 8.1, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the lack of corresponding data is a symptom of an underlying cause as still hidden. Debugging attempts to match symptom with cause, thereby leading to error correction.

Debugging will always have one of two outcomes:

- The cause will be found and corrected and removed or
- The cause will not be found.

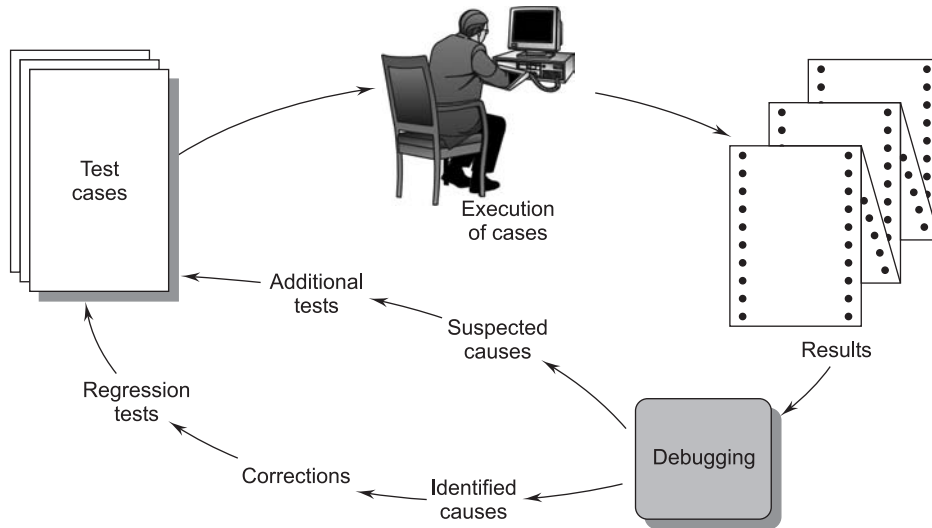


FIGURE 8.1 The Debugging Process

8.2.4 Program Debugging

People think that program testing and debugging are the same thing. Though closely related, they are two distinct processes. Testing establishes the presence of errors in the program. Debugging is the locating of those errors and correcting them. Debugging depends on the output of testing which tells the programmer about the presence or absence of errors.

There are various debugging stages, as shown in Figure 8.2. The incorrect parts of the code are located and the program is modified to meet its requirements. After repairing, the program is tested again to ensure that the errors have been corrected. Debugging can be viewed as a problem-solving process.

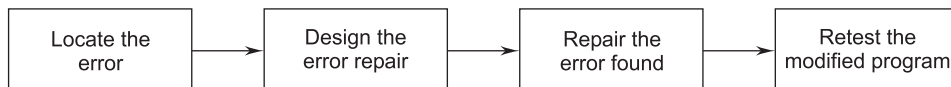


FIGURE 8.2 Debugging Stages

There is no standard method to teach how to debug a program. The debugger must be a skilled person who can easily understand the errors by viewing the output. The debugger must have knowledge of common errors, which occur very often in a program.

After errors have been discovered, then correct the error. If the error is a coding error, then that error can be corrected easily. But, if the error is some design

mistake, then it may require effort and time. Program listings and the hard copy of the output can be an aid in debugging.

8.2.5 Debugging Guidelines

Some general guidelines for effective debugging include:

- Many a times, debugging requires a thorough understanding of the program design.
- Debugging may sometimes even require a full redesign of the system.
- One must be aware of the possibility that any error correction may introduce new errors. Therefore, after every round of error-fixing, regression testing must be carried out.

8.2.6 Characteristics of Bugs

Some characteristics of bugs are as follows:

- The symptom and the cause may be geographically remote.
- The symptom may disappear when another error is corrected.
- The symptom may actually be caused by non-errors.
- The symptom may be caused by a human error.
- The symptom may be a result of timing problems.
- It may be difficult to accurately reproduce input conditions.
- The symptom may be intermittent.
- The symptom may be due to causes that are distributed across a number of tasks running on different processors.

8.3 ERROR, FAULT, AND FAILURE

8.3.1 Errors

An error is a discrepancy between the actual value of the output given by the software and the specified correct value of the output for that given input. That is, error refers to the difference between the actual output of the software and the correct output. An error is also used to refer to the wrong decision in a given case as compared to what is expected to be the right one. Error also refers to human actions that result in software containing a defect or fault.

Types of Errors

Errors can be classified into two categories:

1. **Syntax Error.** A syntax error is a program statement that violates one or more rules of the language in which it is written.
2. **Logic Error.** A logic error deals with incorrect data fields, out-of-range terms, and invalid combinations.

8.3.2 Faults

A fault is a condition that causes a system to fail in performing its required function.

A fault is the basic reason for software malfunction. It is also commonly called a bug. Even though correct input is given to the system, when it fails then we say the system has a fault or a bug, and needs repair.

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.

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, are environmental influence. We can often find faults without execution. They may be found through inspection, compiler diagnostics, design or code reviews, or code reading.

8.3.3 Failure

Failure is the inability of the software to perform a required function to its specification.

In other words, when software goes ahead in processing without showing error or fault even though certain input and process specification are violated, then it is called a software failure.

A software failure occurs when the behavior of software is different from the required behavior.

A failure is produced only when there is a fault in the system. In other words, faults have the potential to cause failures and their presence is a necessary but not a sufficient condition for failure to occur.

EXERCISES

1. Explain, in brief, the various static-testing strategies.
2. Give a comparative study of inspection, reviews, walk-throughs, and checklists.
3. Define various testing strategies in detail.
4. What is a code walk-through? List the important types of errors checked during a code walk-through.
5. How can design attributes facilitate debugging?
6. What are the various debugging approaches? Discuss them with the help of examples.
7. Define the term “debugging.” Explain the various debugging techniques available.
8. Why is it advantageous to detect as many errors as possible during code review than during testing?
9. Define a review. Also, explain the different types of reviews.
10. What is a Formal Technical Review (FTR)? What are the objectives of a FTR?
11. What is the role of a formal technical review as a quality-assurance activity? Discuss the details of the review meeting, reporting, and record keeping.
12. Enumerate the various steps involved in a inspection/review.
13. What is the difference between a code walk-through and a code inspection/review?
14. What are the guidelines used for effective debugging?
15. Describe the debugging process with the help of a suitable diagram.
16. What is program debugging?
17. Enumerate some of the characteristics shown by bugs.
18. What do you understand by the terms error, fault, and failure?
19. What is the difference between a syntax error and logical error?

SOFTWARE MAINTENANCE AND PROJECT MANAGEMENT

9.1 SOFTWARE AS AN EVOLUTION ENTITY

Lehman and Belady have studied the characteristics of the evolution of several software products [1980]. They have expressed their observations in the form of laws. Their important laws are given below.

1. **Lehman's first law:** A software product must change continually or become progressively less useful.
2. **Lehman's second law:** The structure of a program tends to degrade as more and more maintenance is carried out on it.
3. **Lehman's third law:** Over a program's lifetime, its rate of development is approximately constant.

9.2 SOFTWARE-CONFIGURATION MANAGEMENT ACTIVITIES

Configuration management is carried out through three principal activities:

1. Configuration identification,
2. Configuration control, and
3. Configuration accounting.

Briefly, these three activities can be described as:

- **Configuration Identification:** Which parts of the system must be kept track of?
- **Configuration Control:** Ensures that changes to a component happen smoothly.
- **Configuration Accounting:** Keeps track of what has been changed, when, and why.

1. **Configuration Identification.** The project manager normally classifies the objects associated with a software's development into three main categories: controlled, pre-controlled, and uncontrolled. Controlled objects are those that are already put under configuration control. You must follow some formal procedures to change them. Pre-controlled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not and will not be subject to configuration control. Controllable objects include both controlled and pre-controlled objects. Typical controllable objects include:

- Requirement specifications documents
- Designing documents
- Tools used to build the system, such as compilers, linkers, lexical analyzers, parsers, etc.
- Source code for each module
- Test cases
- Problem reports

The configuration-management plan written during the project-planning phase lists all controlled objects.

2. **Configuration Control.** Configuration control is the process of managing changes to controlled objects. The configuration control system prevents unauthorized changes to any controlled object. In order to change a controlled object, such as a module, a developer can get a private copy of the module from a reserve operation (see Figure 9.1). Configuration-management tools allow only one person to reserve a module at any time. Once an object is reserved it does not allow anyone else to reserve this module until the reserved module is restored. Thus, by preventing more

than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.

The CCB is a group of people responsible for configuration management. The CCB evaluates the request based on its effect on the project, and the benefit due to change.

An important reason for configuration control is people need a stable environment to develop a software product.

Suppose you are trying to integrate module A, with modules B and C. You cannot make progress if the developer of module C keeps changing it; this is especially frustrating if a change to module C forces you to recompile A.

As soon as a document under configuration control is updated, the updated version is frozen and is called a baseline, as shown in Figure 9.1.

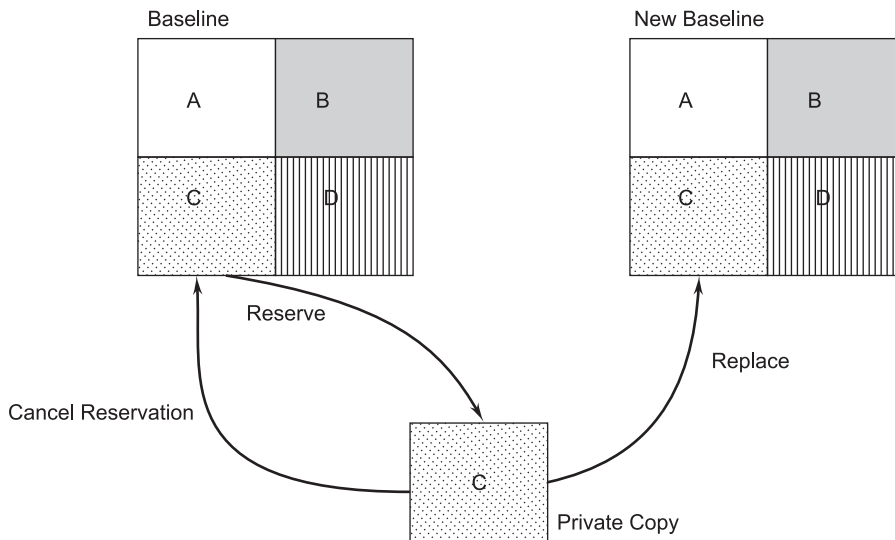


FIGURE 9.1 Configuration Control

3. **Configuration Accounting.** Configuration accounting can be explained with two concepts:

- (i) *Status Accounting.* Once the changes in the baselines occur, some mechanisms must be used to record how the system evolves and what its current state is. This task is accomplished by status accounting. Its basic function is to record the activities related to the other SCM functions.

Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions:

- (a) What happened? (b) Who did it?
- (c) When did it happen? (d) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in Figure 9.1. Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. The key elements under status accounting are shown in Figure 9.2.

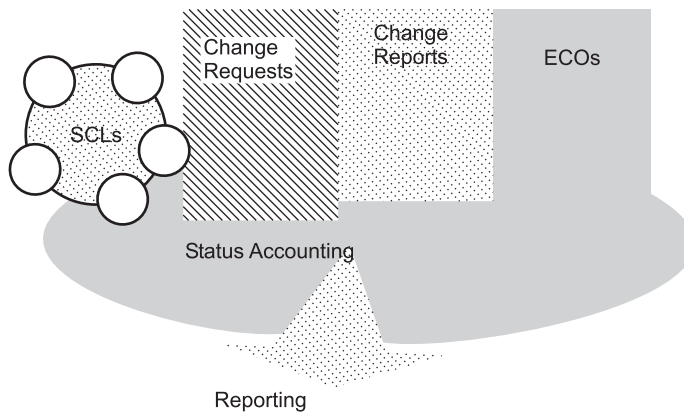


FIGURE 9.2 Status Accounting

- (ii) *Configuration Audit.* A software-configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:
 - (a) Has the change specified in the ECO been made? Have any additional modifications been incorporated?
 - (b) Has a formal technical review been conducted to assess technical correctness?
 - (c) Has the software process been followed and have software-engineering standards been properly applied?
 - (d) Has the change been “highlighted” in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?

- (e) Have SCM procedures for noting the change, recording it, and reporting it been followed?
- (f) Have all related SCIs been properly updated?

The SCM audit revolves around software-configuration items, change requests, and the software quality-assurance plan.

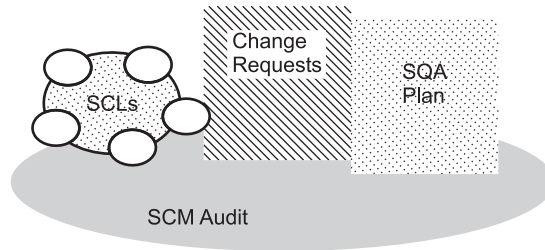


FIGURE 9.3 Configuration Accounting

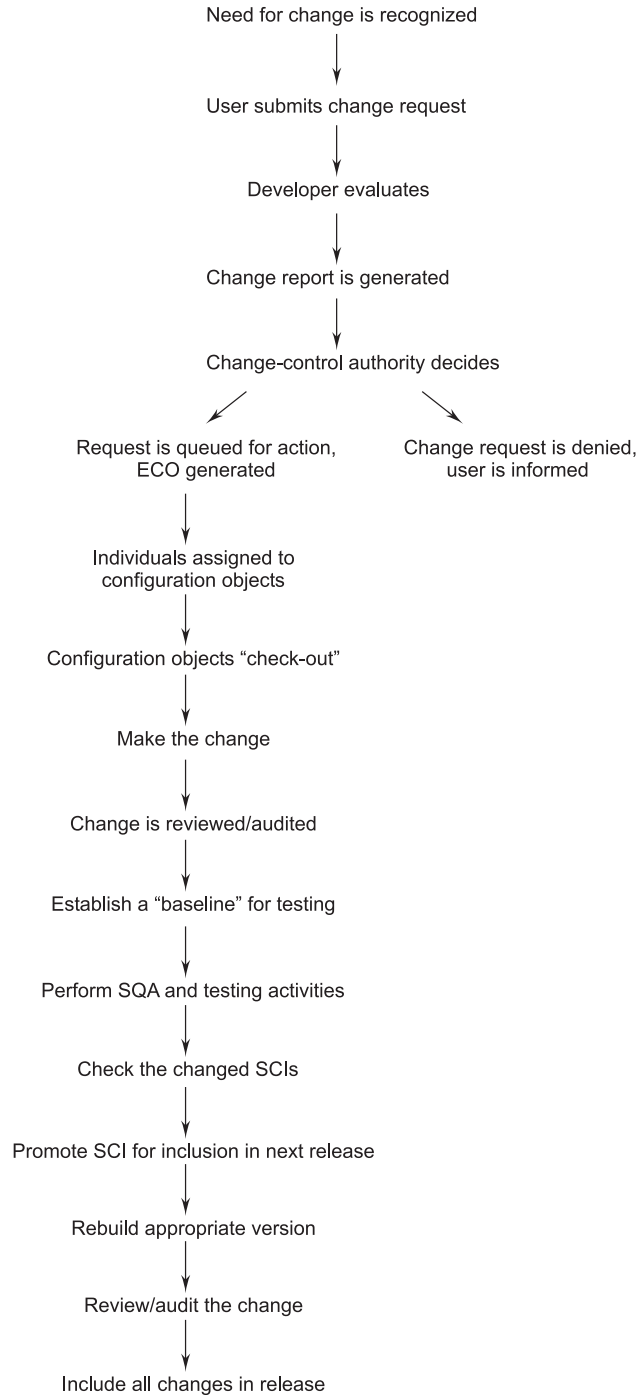
9.3 CHANGE-CONTROL PROCESS

At any moment, the configuration-management team must know the state of any component or document in the system. Consequently, configuration management should emphasize communication among those whose actions affect the system. Cashman and Holt (1980) suggest that we should always know the answers to the following questions:

- Synchronization: When was the change made?
- Identification: Who made the change?
- Naming: What components of the system were changed?
- Authentication: Was the change made correctly?
- Authorization: Who authorized that the change be made?
- Routing: Who was notified of the change?
- Cancellation: Who can cancel the request for a change?
- Delegation: Who is responsible for the change?
- Valuation: What is the priority of the change?

Notice that these questions are management questions, not technical ones. We must use procedures to manage change carefully.

Change control combines human procedures and automated tools to provide a mechanism for the control of change. The change-control process is illustrated schematically in Figure 9.4.

**FIGURE 9.4** Change-Control Process

9.4 SOFTWARE-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 report includes:

- 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.

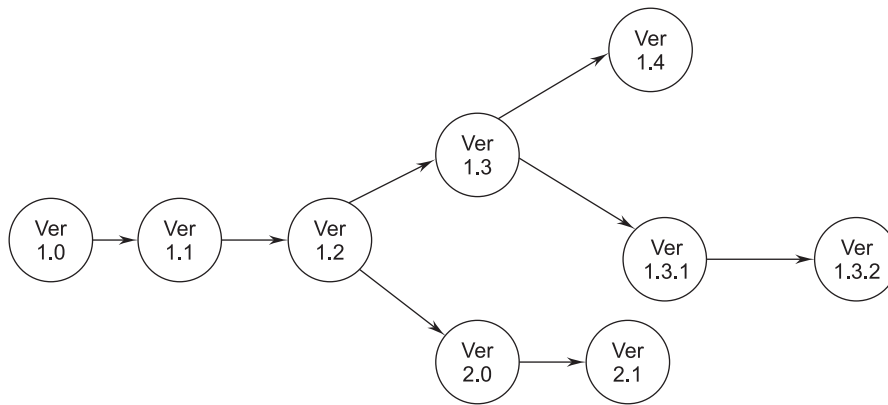


FIGURE 9.5 An Evolutionary Graph for a Different Versions of an Item

The above evolutionary graph (Figure 9.5) depicts the evolution of a configuration item during the development life-cycle. The initial version of the item is given version number Ver 1.0. Subsequent changes to the item which could be mostly fixing bugs or adding minor functionality is given as Ver 1.1 and Ver 1.2. After that, a major modification to Ver 1.2 is given a number Ver 2.0, and at the same time, a parallel version of the same item without the major modification is maintained and given a version number 1.3.

Commercial tools are available for version control, which perform one or more of the following tasks:

- Source-code control
- Revision control
- Concurrent-version control

There are many commercial tools, such as Rational Clear Case, Microsoft Visual Source Safe, and a number of other tools to help version control.

9.5 SOFTWARE-CONFIGURATION MANAGEMENT

Software-configuration management deals with effectively tracking and controlling the configuration of a software product during its life-cycle.

Before we discuss configuration management, we must be clear about what exactly is meant by a version and a revision of a software product. A new version of software is created when there is significant change in functionality, technology, or the hardware it runs on, etc. On the other hand, a new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc.

The configuration-management team is responsible for assuring that each version or release is correct and stable before it is released for use, and that changes are made accurately and promptly. Accuracy is critical, because we want to avoid generating new faults while correcting existing ones. Similarly, promptness is important, because fault detection and correction are proceeding at the same time that the test team searches for additional faults. Thus, those who are trying to repair system faults should work with components and documentation that reflect the current state of the system.

Thus, configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes.

OR

Software-configuration management (SCM) is a set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Configuration-management procedures define how to record and process proposed system changes, how to relate these to system components, and the methods used to identify different versions of the system. Configuration-management tools are used to store versions of system components, build systems from these components, and track the release of system versions to customers.

The IEEE defines SCM as *the process of identifying and defining the items in the system, controlling the change of these items throughout their life-cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items.*

Figure 9.6 shows a simple configuration-management procedure based on the rebuilding and deployment process.

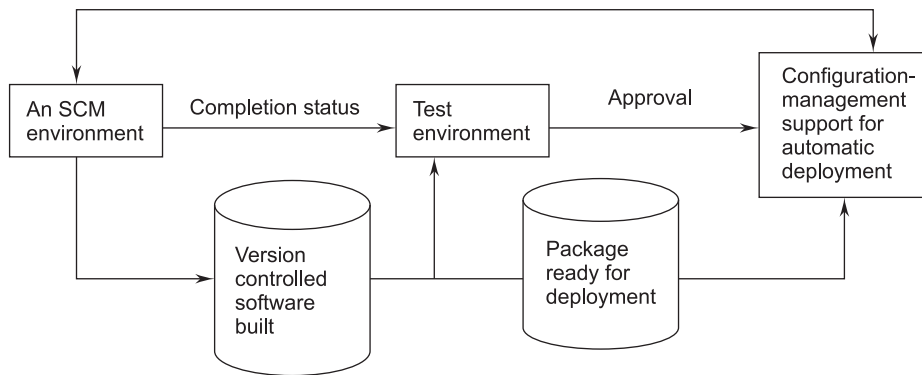


FIGURE 9.6 Simple Configuration-Management Environment

9.5.1 Versions and Releases

A configuration for a particular system is sometimes called a version. Thus, the initial delivery of a software package may consist of several versions, one for each platform or situation in which the software will be used. For example, aircraft software may be built so that version 1 runs on Navy planes, version 2 runs on Air Force planes, and version 3 runs on commercial airliners.

A new release of the software is an improved system intended to replace the old one. Often, software systems are described as version n , release m , or as version $n.m$, where the number reflects the system's position as it grows and matures. Version n is sometimes intended to replace version $n-1$, and release m supersedes $m-1$.

Version and Release Management

Version and release management are the processes of identifying and keeping track of different versions and releases of a system. New system versions should always be created by the CM team rather than the system developers, even when they are not intended for external release. This makes it easier to maintain consistency in the configuration database as only the CM team can change version information.

A system version is an instance of a system that differs, in some way, from other instances. New versions of the system may have different functionality, performance, or may repair system faults. Some versions may be functionally equivalent but designed for different hardware or software configurations. If there are only small differences between versions, one of these is sometimes called a variant of the other.

A system release is a version that is distributed to customers. Each system release should either include new functionality or be intended for a different hardware platform. There are always many more versions of a system than releases as versions are created within an organization for internal development or testing that are never released to customers.

9.6 NEED FOR MAINTENANCE

Software maintenance is the activity associated with keeping an operational computer system continuously in tune with the requirements of users and data-processing operations. The software maintenance process is expensive and risky and is very challenging. There is a need for software maintenance due to the following reasons:

- Changes in user requirements with time
- Program/System problems
- Changing hardware/Software environment
- To improve system efficiency and throughout
- To modify the components
- To test the resulting product to verify the correctness of changes
- To eliminate any unwanted side effects resulting from modifications
- To augment or fine-tune the software
- To optimize the code to run faster
- To review standards and efficiency
- To make the code easier to understand and work with
- To eliminate any deviations from specifications

9.6.1 Maintenance To-Do List

- Correct errors
- Correct requirements and design flaws
- Improve the design
- Make enhancements
- Interface with other systems
- Convert for use with other hardware
- Migrate legacy systems
- Retire systems

- Major aspects
- Maintain control over the system's day-to-day functions
- Maintain control over system modification
- Perfect existing acceptable functions
- Prevent system performance from degrading to unacceptable levels

9.7 CATEGORIES OF MAINTENANCE

Maintenance may be **classified** into the four categories as follows:

- **Corrective** - reactive modifications to correct discovered problems.
 - **Adaptive** - modifications to keep it usable in a changed or changing environment.
 - **Perfective** - improve performance or maintainability.
 - **Preventive** - modifications to detect and correct latent faults.
- (i) **Corrective Maintenance.** Corrective maintenance means repairing processing or performance failures or making changes because of previously uncorrected problems.
- (ii) **Adaptive Maintenance.** Adaptive maintenance means changing the program functions. This is done to adapt to external environment changes. For example, the current system was designed so that it calculates taxes on profits after deducting the dividend on equity shares. The government has issued orders now to include the dividend in the company profit for tax calculation. This function needs to be changed to adapt to the new system.
- (iii) **Perfective Maintenance.** Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's additional or changing needs. For example, earlier data was sent from stores to headquarters on magnetic media but after stores were electronically linked via leased lines, the software was enhanced to send data via leased lines.

As maintenance is very costly and very essential, efforts have been done to reduce its costs. One way to reduce the costs is through a maintenance management and software-modification audit. Software modification consists of program rewriting and system level-upgradation.

- (iv) **Preventive Maintenance.** Preventive maintenance is the process by which we prevent our system from being obsolete. Preventive maintenance involves the concept of re-engineering and reverse engineering in which an old system with an old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

9.8 MAINTENANCE COSTS

In the 1970s, most of a software system's budget was spent on development. The ratio of development money to maintenance money was reversed in the 1980s, and various estimates place maintenance at 40 to 60% of the full life-cycle cost of a system (i.e., from development through maintenance to eventual retirement or replacement). However, this cost may vary widely from one application domain to another.

It is advisable to invest more effort in early phases of the software life-cycle to reduce maintenance costs. The defect repair ratio increases heavily from the analysis phase to the implementation phase as shown in Table 9.1.

TABLE 9.1 Defect Repair Ratio

Phase	Ratio
Analysis	1
Design	10
Implementation	100

Therefore, more effort during development will certainly reduce the cost of maintenance.

9.8.1 Factors Affecting Effort

There are many other factors that contribute to the effort needed to maintain a system. These factors include the following:

- Application type
- System novelty
- Turnover and maintenance staff availability
- System life-span
- Dependence on a changing environment
- Hardware characteristics
- Design quality
- Code quality
- Documentation quality
- Testing quality

9.8.2 Modeling Maintenance Effort

As with development, we want to estimate the effort required to maintain a software system.

1. **Belady and Lehman Model.** Belady and Lehman (1972) were among the first researchers to try to capture maintenance effort in a predictive model. This model indicates that effort and costs can increase exponentially if a poor software development approach is used, and the person or group that used the approach is no longer available to perform maintenance.

Belady and Lehman capture these effects in an equation:

$$M = P + Ke^{(c-d)}.$$

M is the total maintenance effort expended for a system, and p represents wholly productive efforts: analysis, evaluation, design, coding, and testing. c is the complexity caused by the lack of structured design and documentation; it is reduced by d , the degree to which the maintenance team is familiar with the software. Finally, K is a constant determined by comparing this model with the effort relationships on actual projects; it is called an empirical constant because its value depends on the environment.

The Belady-Lehman equation expresses a very important relationship among the factors determining maintenance effort. 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, the function and purpose of the software, then the value of ' d ' will be low.

The result is that the cost for maintenance increases exponentially. Thus, to economize on maintenance, the best approach is to build the system using good software-engineering practices and to give the maintainers time to become familiar with the software.

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) *The maintenance team has good level of understanding of the project ($d = 0.9$).*
- (ii) *The maintenance team has poor understanding of project ($d = 0.1$).*

Solution.

$$\text{Development effort (P)} = 500 \text{ PM}$$

$$K = 0.3$$

$$C = 8.$$

- (i) The maintenance team has good level of understanding ($d = 0.9$)

$$\begin{aligned} M &= P + Ke^{(c-d)} \\ &= 500 + 0.3e^{(8-0.9)} \\ &= 500 + 363.59 = 863.59 \text{ PM.} \end{aligned}$$

- (ii) The maintenance team has poor level of understanding ($d = 0.1$)

$$\begin{aligned} M &= P + Ke^{(c-d)} \\ &= 500 + 0.3e^{(8-0.1)} \\ &= 500 + 809.18 = 1309.18 \text{ PM.} \end{aligned}$$

Hence, it is clear that effort increases exponentially if poor software-engineering approaches are used and understandability of the project is poor.

2. **Boehm Model.** Boehm 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.

$$ACT = \frac{KLOC_{\text{added}} + KLOC_{\text{deleted}}}{KLOC_{\text{total}}},$$

where $KLOC_{\text{added}}$ is the total kilo lines of source code added during maintenance. $KLOC_{\text{deleted}}$ is the total $KLOC_{\text{deleted}}$ during maintenance. Thus, the code that is changed should be counted in both the code added and the code deleted.

The annual change traffic is multiplied with the total development cost to arrive at the maintenance cost.

$$\text{Maintenance cost} = ACT \times \text{development cost}$$

Most maintenance-cost estimation models, however, give only approximate results because they do not take into account several factors, such as the experience level of engineers and the familiarity of engineers with the product, hardware requirements, software complexity, etc.

Example 9.2. *The Annual change traffic for a software system is 15% per year. The development effort is 600 PMs. Compute an estimate for the Annual maintenance effort (AME). If the lifetime of the project is 10 years, what is the total effort of the project?*

Solution.

The development effort = 600 PM.

ACT = 15%.

Total duration for which effort is to be calculated = 10 years.

The maintenance effort is a fraction of the development effort and is assumed to be constant.

$$\begin{aligned} \text{AME} &= \text{ACT} \times \text{SDE} \\ &= 0.15 \times 600 = 90 \text{ PM} \end{aligned}$$

The maintenance effort for 10 years = $10 \times 90 = 900 \text{ PM}$

Total effort = $600 + 900 = 1500 \text{ PM}$.

9.9 SOFTWARE-PROJECT ESTIMATION

Software-project estimation is the process of estimating various resources required for the completion of a project. Effective software-project estimation is an important activity in any software-development project. Underestimating software projects and understaffing it often leads to low-quality deliverables, and the project misses the target deadline leading to customer dissatisfaction and loss of credibility to the company. On the other hand, overstaffing a project without proper control will increase the cost of the project and reduce the competitiveness of the company.

Software-project estimation mainly encompasses the following steps:

1. **Estimating the Size of the Project.** There are many procedures available for estimating the size of a project, which are based on quantitative approaches, such as estimating lines of code or estimating the functionality requirements of the project called function points.
2. **Estimating Efforts Based on Person-months or Person-hours.** Person-month is an estimate of the personal resources required for the project.
3. **Estimating Schedule in Calendar Days/Month/Year Based on Total Person-months Required and Manpower Allocated to the Project.** The duration in calendar month = Total person-months/Total manpower allocated.
4. **Estimating Total Cost of the Project Depending on the Above and Other Resources.** In a commercial and competitive environment, software-project estimation is crucial for managerial decision-making. Table 9.2 gives the relationship between various management functions and software metrics/indicators. Project estimation and tracking help to

plan and predict future projects and provide baseline support for project management and supports decision-making.

TABLE 9.2 Software-Project Estimation

Activity	Tasks Involved
Planning	Cost estimation, planning for training of manpower, project scheduling, and budgeting the project.
Controlling	Size metrics and schedule metrics help the manager to keep control of the project during execution.
Monitoring/improving	Metrics are used to monitor progress of the project and wherever possible sufficient resources are allocated to improve it.

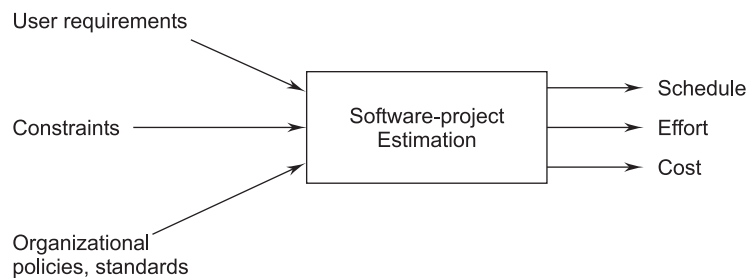


FIGURE 9.7 Software-project Estimation

9.9.1 Estimating Size

Estimating the size of the software to be developed is the very first step to make an effective estimation of the project. Customer requirements and system specifications form a baseline for estimating the size of a software. At a later stage of the project, system design documents can provide additional details for estimating the overall size of the software.

- The ways to estimate project size can be through past data from an earlier developed system. This is called estimation by analogy.
- The other way of estimation is through product feature/functionality. The system is divided into several subsystems depending on functionality, and the size of each subsystem is calculated.

9.9.2 Estimating Effort

Once the size of software is estimated, the next step is to estimate the effort based on the size. The estimation of effort can be made from the organizational specifics of the software-development life-cycle. The development of any application software system is more than just the coding of the system. Depending on deliverable requirements, the estimation of effort for a project will vary.

Efforts are estimated in the number of person-months.

- The best way to estimate effort is based on the organization's own historical data of development processes. Organizations follow a similar development life-cycle when developing various applications.
- If the project is of a different nature, which requires the organization to adopt a different strategy for development, then different models based on algorithmic approaches can be devised to estimate the required effort.

9.9.3 Estimating Schedule

The next step in the estimation process is estimating the project schedule from the effort estimated. The schedule for a project will generally depend on human resources involved in a process. Efforts in person-months are translated to calendar months.

Schedule estimation in calendar months can be calculated using the following model [McConnell]:

$$\text{Schedule in calendar months} = 3.0 * (\text{person-months})^{1/3}$$

The parameter 3.0 is variable, used depending on the situation that works best for the organization.

9.9.4 Estimating Cost

Cost estimation is the next step for projects. The cost of a project is derived not only from the estimates of effort and size but from other parameters, such as hardware, travel expenses, telecommunication costs, training costs, etc. Figure 9.8 depicts the cost-estimation process.

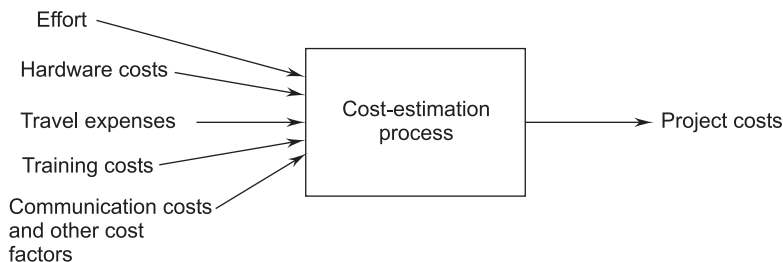


FIGURE 9.8 Cost-estimation Process

Figure 9.9 depicts the project-estimation process.

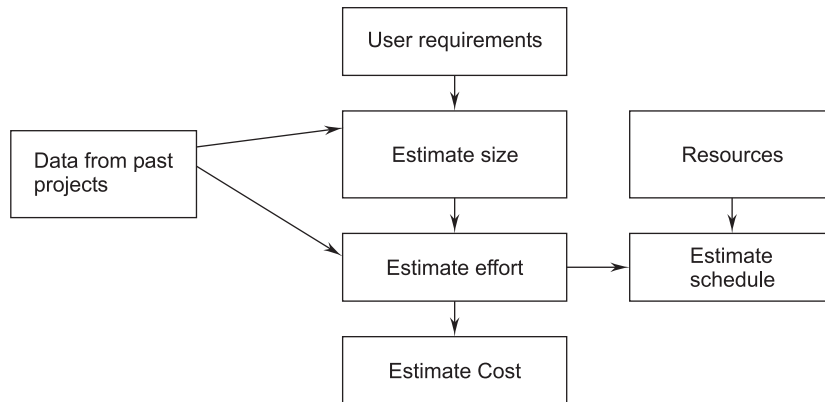


FIGURE 9.9 Project-estimation Process

Once the estimation is complete, we may be interested to know how accurate the estimates are. The answer to this is “we do not know until the project is complete.” There is always some uncertainty associated with all estimation techniques. The accuracy of project estimation will depend on the following:

- Accuracy of historical data used to project the estimation.
- Accuracy of input data to various estimates.
- Maturity of an organization’s software-development process.

The following are some of the reasons cost estimation can be difficult:

- Software-cost estimation requires a significant amount of effort. Sufficient time is usually not allocated for planning.
- Software-cost estimation is often done hurriedly, without an appreciation for the actual effort required and is far from realistic.
- Lack of experience for developing estimates, especially for large projects.
- An estimator uses the extrapolation technique to estimate, ignoring the non-linear aspects of the software-development process.

9.9.5 Reasons for Poor/Inaccurate Estimation

The following are some of the reasons for poor and inaccurate estimation:

- Requirements are imprecise. Also, requirements change frequently.
- The project is new and is different from past projects handled.
- Non-availability of enough information about past projects.
- Estimates are forced to be based on available resources.

- Cost and time tradeoffs.

If we elongate the project, we can reduce overall costs. Usually, customers and management do not like long project durations. There is always the shortest possible duration for a project, but it comes at a cost.

The following are some of the problems associated with estimates:

- Estimating size is often skipped and a schedule is estimated, which is of more relevance to management.
- Estimating size is perhaps the most difficult step, which has a bearing on all other estimates.
- Let us not forget that even good estimates are only projections and subject to various risks.
- Organizations often give less importance to collection and analysis of historical data of past development projects. Historical data is the best input to estimate a new project.
- Project managers often underestimate the schedule because management and customers often hesitate to accept a prudent realistic schedule.

9.9.6 Project-estimation Guidelines

Some guidelines for project estimation are as follows:

- Preserve and document data pertaining to past projects.
- Allow sufficient time for project estimation especially for bigger projects.
- Prepare realistic developer-based estimates. Associate people who will work on the project to reach a realistic and more accurate estimate.
- Use software-estimation tools.
- Re-estimate the project during the life-cycle of the development process.
- Analyze past mistakes in the estimation of projects.

9.10 CONSTRUCTIVE COST MODEL (COCOMO)

COCOMO stands for Constructive Cost Model. It was introduced by Barry Boehm in 1981. It is perhaps the best known and most thoroughly documented of all software-cost estimation models. It provides the following three levels of models:

- **Basic COCOMO:** A single-value model that computes software-development costs as a function of an estimate of LOC.
- **Intermediate COCOMO:** This model computes development costs and effort as a function of program size (LOC) and a set of cost drivers.

- **Complete COCOMO:** This model computes development effort and costs which incorporates all characteristics of intermediate levels with assessment of cost implications in each step of development (analysis, design, testing, etc.).

This model may be applied to three classes of software projects as given below:

- **Organic:** Small-size project. A simple software project where the development team has good knowledge of the application.
- **Semi-detached:** An intermediate-size project, and the project is based on rigid and semi-rigid requirements.
- **Embedded:** The project is developed under hardware, software, and operational constraints. Examples are embedded software and flight-control software.

9.10.1 Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

$$\begin{aligned}\text{Effort} &= a_1 \times (\text{KLOC})^{a_2} && \text{PM} \\ T_{\text{dev}} &= b_1 \times (\text{Effort})^{b_2} && \text{Months}\end{aligned}$$

where KLOC is the estimated size of the software product expressed in Kilo Lines and Code a_1, a_2, b_1, b_2 are constants of the software product.

T_{dev} is the estimated time to develop the software, expressed in months.

Effort is the total effort required to develop the software product, expressed in person-months (PM).

The person-month curve is shown in Figure 9.10.

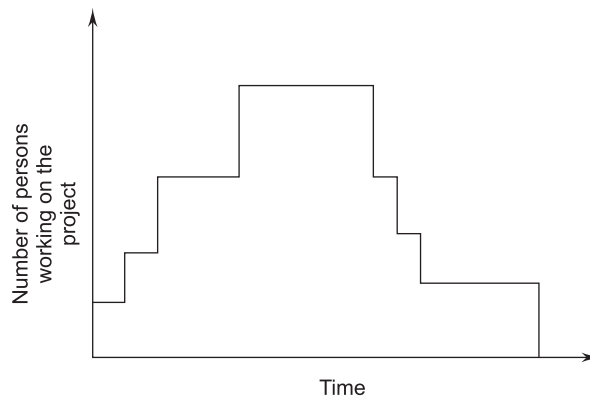


FIGURE 9.10 Person-month Curve

Estimation of Development Effort

Organic: $\text{Effort} = 2.4 (\text{KLOC})^{1.05} \text{ PM}$

Semi-detached: $\text{Effort} = 3.0 (\text{KLOC})^{1.12} \text{ PM}$

Embedded: $\text{Effort} = 3.6 (\text{KLOC})^{1.20} \text{ PM}$

Figure 9.11 (a) shows a plot of the estimated effort versus the size for various product sizes.

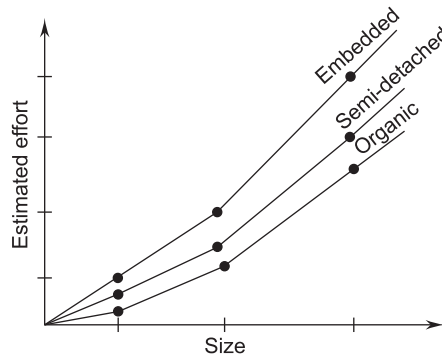


FIGURE 9.11 (a) Effort Versus Size

From Figure 9.11 (a), we can observe that the effort is almost linearly proportional to the size of the software product.

Estimation of Development Time

Organic: $T_{\text{dev}} = 2.5 (\text{Effort})^{0.38} \text{ Months}$

Semi-detached: $T_{\text{dev}} = 2.5 (\text{Effort})^{0.35} \text{ Months}$

Embedded: $T_{\text{dev}} = 2.5 (\text{Effort})^{0.32} \text{ Months}$

Figure 9.11 (b) shows a plot of development time versus product size.

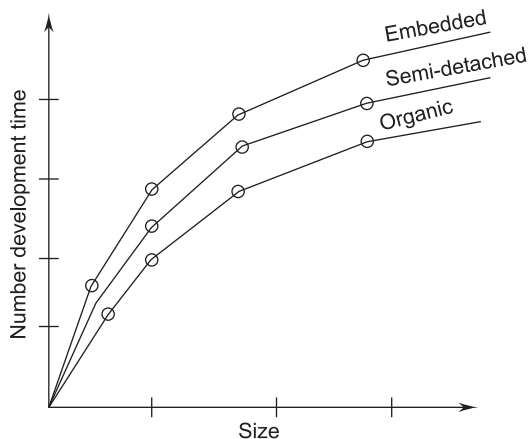


FIGURE 9.11 (b) Development Time Versus Product Size

From Figure 9.11(b), we can observe that the development time is a sub-linear function of the size of the product.

9.10.2 Intermediate COCOMO Model

The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained through the basic COCOMO expression by using a set of 15 cost drivers (multipliers) based on various attributes of software development.

TABLE 9.3 COCOMO Intermediate Cost Drivers

Driver Type	Code	Cost Driver
Product attributes	RELY	Required software reliability
	DATA	Database size
	CPLX	Product complexity
Computer attributes	TIME	Execution time constraints
	STOR	Main storage constraints
	VIRT	Virtual machine volatility—degree to which the operating system changes
	TURN	Computer turnaround time
Personnel attributes	ACAP	Analyst capability
	AEXP	Application experience
	PCAP	Programmer capability
	VEXP	Virtual machine (i.e., operation system) experience
	LEXP	Programming-language experience
Project attributes	MODP	Use of modern programming practices
	TOOL	Use of software tools
	SCED	Required development schedule

9.10.3 Complete COCOMO Model

The shortcomings of both basic and intermediate COCOMO models are that they:

- Consider a software product a single homogenous entity. However, most large systems are made up of several smaller subsystems. Some subsystems may be considered organic, some embedded, etc. For some, the reliability requirements may be high, and so on.

- Consider the cost of each subsystem separately.
- Consider the costs of the subsystems but add them separately to obtain total cost.
- Reduce the margin of error in the final estimate.

A large amount of work has been done by Boehm to capture all significant aspects of software development. It offers a means for processing all the project characteristics to construct a software estimate. The complete model introduces two more capabilities:

1. **Phase-Sensitive Effort Multipliers.** Some phases (design, programming, and integration/test) are more affected than others by factors defined by the cost drivers. The complete 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 levels; that is, the level at which it is most susceptible to variation.

9.11 SOFTWARE-RISK ANALYSIS AND MANAGEMENT

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

Risk is defined as an exposure to the chance of injury or loss. That is, risk implies that there is a possibility that something negative may happen. In the context of software projects, negative implies that there is an adverse effect on cost, quality, or schedule.

9.11.1 Risk Management

Risk management is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimal.

Risk management is a scientific process based on the application of game theory, decision theory, probability theory, and utility theory. The Software Engineering Institute (SEI) classifies the risk hierarchy as shown in Figure 9.12.

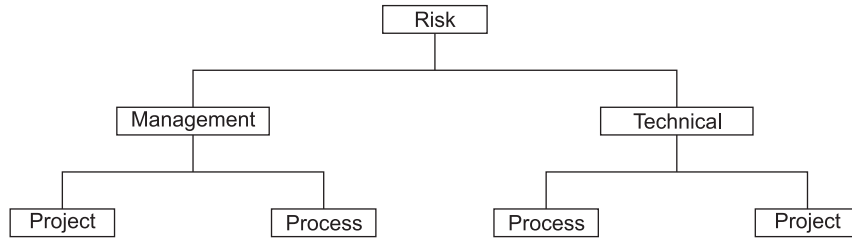


FIGURE 9.12 Risk Hierarchy

Risk scenarios may emerge out of management challenges and technical challenges in achieving specific goals and objectives.

Risk management must be performed regularly throughout the achievement life-cycle. Risks are dynamic, as they change over time. Risk management should not be regarded as an activity integral to the main process of achieving specific goals and objectives. Risk and its management should not be treated as an activity outside the main process of achievement. Risk is managed best when risk management is implemented as a mainstream function in the software-development and goal-achievement processes.

9.11.2 Management of Risks

Risk management plays an important role in ensuring that the software product is error-free. Firstly, risk management takes care that the risk is avoided, and if it is not avoidable, then the risk is detected, controlled, and finally recovered.

Risk-management Categories

A priority is given to risk and the highest priority risk is handled first. Various factors of the risk include who are the involved team members, what hardware and software items are needed, where, when, and why. The risk manager does scheduling of risks. Risk management can be further categorized as follows:

1. Risk Avoidance
 - Risk anticipation
 - Risk tools
2. Risk Detection
 - Risk analysis
 - Risk category
 - Risk prioritization

3. Risk Control

- Risk pending
- Risk resolution
- Risk not solvable

4. Risk Recovery

- Full
- Partial
- Extra/alternate feature

Figure 9.13 depicts a risk-management tool.

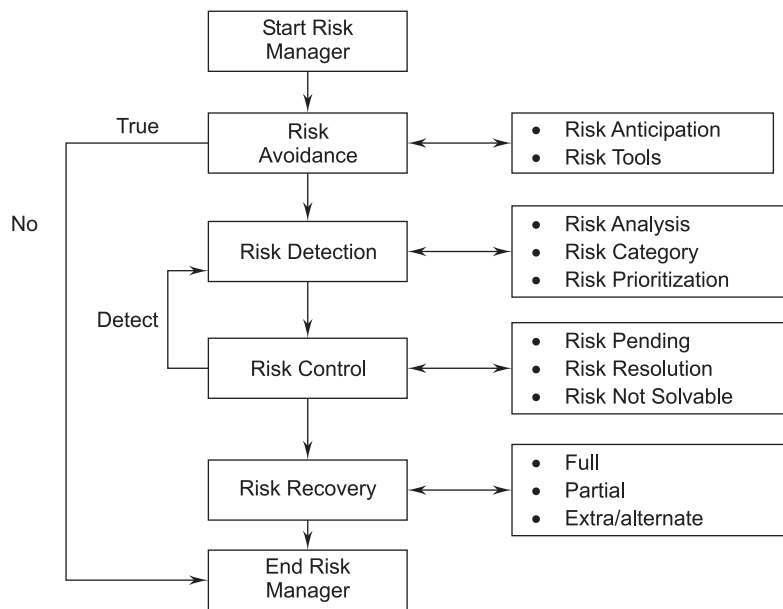


FIGURE 9.13 Risk-management Tool

From Figure 9.13, it is clear that the first phase is to avoid risk by anticipating and using tools from previous project histories. In the case where there is no risk, the risk manager stops. In the case of risk, detection is done using various risk analysis techniques and further prioritizing risks. In the next phase, risk is controlled by pending risks, resolving risks, and in the worst case (if the risk is not solved), lowering the priority. Lastly, risk recovery is done fully, partially, or an alternate solution is found.

1. Risk Avoidance

- *Risk Anticipation:* Various risk anticipation rules are listed according to standards from previous projects, experience, and also as mentioned by the project manager.
- *Risk Tools:* Risk tools are used to test whether the software is risk-free. The tools have a built-in database of available risk areas and can be updated depending upon the type of project.

2. Risk Detection

The risk-detection algorithm detects a risk and it can be categorically stated as the following:

- *Risk Analysis:* In this phase, the risk is analyzed with various hardware and software parameters as probabilistic occurrence (pr), weight factor (wf) (hardware resources, lines of code, people), and risk exposure (pr * wf). Table 9.4 depicts a risk-analysis table.

TABLE 9.4 Risk-analysis Table

S.No.	Risk Name	Probability of Occurrence (pr)	Weight Factor (wf)	Risk Exposure (pr*wf)
1.	Stack overflow	5	15	75
2.	No password forgot option	7	20	140
.....

The maximum value of risk exposure indicates that the problem has to be solved as soon as possible and be given high priority. A risk-analysis table is maintained as shown in Table 9.4.

- *Risk Category:* Risk identification can come from various factors, such as persons involved in the team, management issues, customer specification and feedback, environment, commercial, technology, etc. Once the proper category is identified, priority is given depending upon the urgency of the product.
- *Risk Prioritization:* Depending upon the entries of the risk-analysis table, the maximum risk exposure is given high priority and has to be solved first.

3. **Risk Control.** Once the prioritization is done, the next step is to control various risks as follows:

- *Risk Pending:* According to the priority, low-priority risks are pushed to the end of the queue with a view of various resources (hardware, manpower, software) and if it takes more time their priority is made higher.
- *Risk Resolution:* The risk manager decides how to solve the risk.
- *Risk Elimination:* This action leads to serious errors in the software.
- *Risk Transfer:* If the risk is transferred to some part of the module, then the risk-analysis table entries get modified. And again, the risk manager will control high-priority risks.
- *Disclosures:* Announce the smaller risks to the customer or display message boxes as warnings so that the user take proper steps during data entry, etc.
- *Risk not Solvable:* If a risk takes more time and more resources, then it is dealt with in its totality on the business side of the organization and thereby the customer is notified, and the team member proposes an alternate solution. There is a slight variation in the customer specifications after consultation.

4. **Risk Recovery**

- *Full:* The risk-analysis table is scanned and if the risk is fully solved, then the corresponding entry is deleted from the table.
- *Partial:* The risk-analysis table is scanned and due to partially solved risks, the entries in the table are updated and thereby priorities are also updated.
- *Extra/alternate features:* Sometimes it is difficult to remove risks, and in that case, we can add a few extra features, that solve the problem. Therefore, some coding is done to resolve the risk. This is later documented or the customer is notified.

9.11.3 Sources of Risks

There are two major sources of risk, which are as follows:

1. **Generic Risks.** Generic risks are the risks common to all software projects. For example, requirement misunderstanding, allowing insufficient time for testing, losing key personnel, etc.
2. **Project-Specific Risks.** A vendor may be promising to deliver particular software by a particular date, but is unable to do it.

9.11.4 Types of Risks

There are three types of risks, which are discussed as follows:

1. **Product Risks.** Product risks are risks that affect the quality or performance of the software being developed. This originates from conditions, such as unstable requirement specifications, not being able to meet the design specifications affecting software performance, and uncertain test specifications. In view of the software product risks, there is a risk of losing the business and facing strained customer relations. For example, CASE tools under performance.
2. **Business Risks.** Business risks are risks that affect the organization developing or procuring the software. For example, technology changes and product competition. The top five business risks are:
 - Building an excellent product or system that no one really wants (market risks).
 - Building a product that no longer fits into the overall business strategy for the company (strategic risk).
 - Building a product that the sales force doesn't understand how to sell.
 - Losing the support of senior management due to a change in focus or a change in people (management risk).
 - Losing budgetary or personnel commitment (budget risks).
- 3 **Project Risks.** Project risks are risks that affect the project schedule or resources. These risks occur due to conditions and constraints about resources, relationships with vendors and contractors, unreliable vendors, and lack of organizational support. Funding is the significant project risk management has to face. It occurs due to initial budgeting constraints and unreliable customer payments. For example, staff turnover, management change, hardware uninvertibility.

EXERCISES

1. What is software maintenance? Describe various categories of maintenance. Which category consumes maximum effort and why?
2. Some people feel that "maintenance is manageable." What is your opinion about this issue?
3. Explain Boehm's maintenance model with the help of a diagram.
4. Describe various maintenance-cost estimation models.
5. Define the Belady and Lehman model for the calculation of maintenance effort.

6. Annual change traffic (ACT) in a software system is 40% per year. The initial development cost was Rs 25 lacs. The total life-time for the software is 12 years. What is the total cost of the software system?
7. Why is maintenance needed?
8. What are the different types of maintenance that a software product might need? Why is such maintenance required?
9. What is software configuration? What is meant by software-configuration management? How can you manage software configuration (only give the names of the principle activities involved)?
10. What is the difference between a revision and a version of a software product?
11. Define the terms change control and version control. Why are these necessary?
12. Explain how change and version control are achieved using a configuration-management tool.
13. Define:
 - (a) Status accounting
 - (b) Configuration audit
14. What is software-configuration management? Also discuss, in brief, the various software-configuration management activities.
15. What is configuration identification?
16. What are configuration-management activities? Draw the schematic diagram of a change-control process.
17. List three common types of risks that a typical software project might suffer from.
18. Define "software version control."
19. Using a schematic diagram show the order in which the following are estimated using COCOMO estimation techniques: cost, effort, duration, size.
20. What are the various reasons for poor/inacurate estimation?
21. Discuss the various types of COCOMO models. Explain the phase-wise distribution of effort.
22. Describe the basic COCOMO model in detail.
23. Why does cost estimation play an important role in the software-development process?
24. What is risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?
25. What are the two major sources of risk? Explain the three categories of risks that a software project might suffer from.

Chapter 10

COMPUTER-AIDED SOFTWARE ENGINEERING

10.1 CASE AND ITS SCOPE

CASE stands for Computer-Aided Software Engineering.

CASE is a tool that helps a software engineer maintain and develop software. The workshop for software engineering is called an Integrated Project Support Environment (IPSE) and the tool-set that fills the workshop is called CASE.

CASE is an automated support tool for software engineers in any software-engineering process.

Software engineering mainly includes the following processes:

- Translation of user needs into software requirements
- Transaction of software requirements into design specifications
- Implementation of design into code
- Testing of the code
- Documentation

CASE technology provides software-process support by automating some process activities and by providing information about the software being developed. Examples of activities, which can be automated using CASE, include:

- The development of graphical system models as part of the requirements specification or the software design.
- Understanding a design using a data dictionary, which holds information about the entities and relations in a design.
- The generation of user interfaces from a graphical interface description, which is created interactively by the user.
- Program debugging through the provision of information about an executing program.
- The automated translation of programs from an old version of a programming language, such as COBOL, to a more recent version.

10.2 LEVELS OF CASE

There are three different levels of CASE technology:

1. **Production Process Support Technology.** This includes support for process activities, such as specification, design, implementation, testing, and so on.
2. **Process Management and Technology.** This includes tools to support process modeling and process management. These tools are used for specific support activities.
3. **Meta-CASE Technology.** Meta-CASE tools are generators, which are used to create production process-management support tools.

10.3 ARCHITECTURE OF CASE ENVIRONMENT

The architecture of the CASE environment is illustrated in Figure 10.1.

The important components of a modern CASE environment are the user interface, the Tools Management System (tools-set), the Object Management System (OMS), and the repository. These various components are discussed as follows:

1. **User Interface.** It provides a consistent framework for accessing different tools; thus making it easier for the user to interact with different tools and reduces learning time of how the different tools are used.

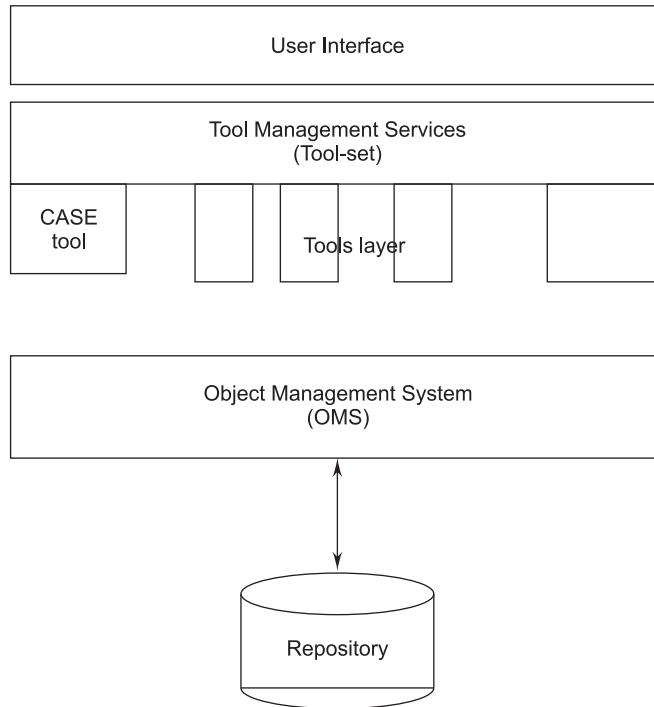


FIGURE 10.1 Architecture of CASE Environment

2. **Tools-Management Services (Tools-set).** The tools-set section holds the different types of improved quality tools. The tools layer incorporates a set of tools-management services with the CASE tool themselves. The Tools Management Service (TMS) controls the behavior of tools within the environment. If multitasking is used during the execution of one or more tools, TMS performs multitask synchronization and communication, coordinates the flow of information from the repository and object-management system into the tools, accomplishes security and auditing functions, and collects metrics on tool usage.
3. **Object-Management System (OMS).** The object-management system maps these (specification design, text data, project plan, etc.) logical entities into the underlying storage-management system, i.e., the repository.

Working in conjunction with the CASE repository, the OML provides integration services a set of standard modules that couple tools with the repository. In addition, the OML provides configuration management services by enabling the identification of all configuration objects performing version control, and providing support for change control, audits, and status accounting.

4. **Repository.** The repository is the CASE database and the access-control functions that enable the OMS to interact with the database. The word CASE repository is referred to in different ways, such as project database, IPSE database, data dictionary, CASE database, and so on.

10.4 BUILDING BLOCKS FOR CASE

The building blocks for CASE are illustrated in Figur 10.2.

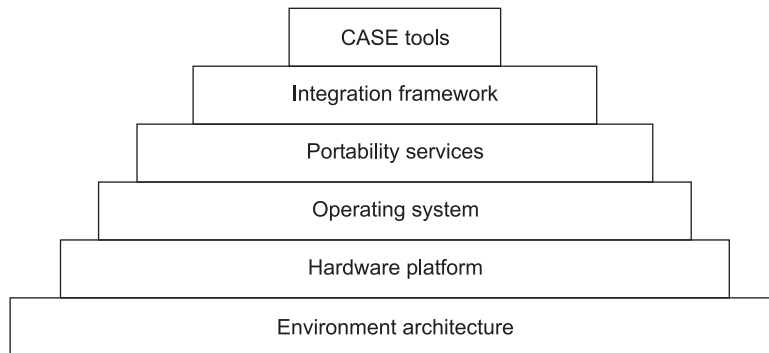


FIGURE 10.2 CASE Building Blocks

1. **Environment Architecture.** The environment architecture, composed of the hardware platform and operating system support including networking and database management software, lays the groundwork for CASE but the CASE environment itself demands other building blocks.
2. **Portability Services.** A set of portability services provides a bridge between CASE tools and their integration framework and the environment architecture. These portability services allow the CASE tools and their integration framework to migrate across different hardware platforms and operating systems without significant adaptive maintenance.
3. **Integration Framework.** It is a collection of specialized programs that enables individual CASE tools to communicate with one another and to create a project database.
4. **Case Tools.** Case tools are used to assist software-engineering activities (such as analysis modeling, code generation, etc.) by either communicating with other tools, the project database (integrated CASE environment), or as point solutions.

10.5 CASE SUPPORT IN SOFTWARE LIFE-CYCLE

There are various types of support that CASE provides during the different phases of a software life-cycle.

1. **Prototyping Support.** The prototyping is useful to understand the requirements of complex software products, to market new ideas and so on. The prototyping CASE tools requirements are as follows:

- Define user interaction
- Define the system control flow
- Store and retrieve data required by the system
- Incorporate some processing logic

A few features that are supported by prototyping tools include:

- Developing the graphical user interface (GUI). The user should be allowed to define all data entry forms, menus, and control.
- They integrate well with the data dictionary of a CASE environment.
- They should be able to integrate with the external user-defined modules written in high-level languages.
- The user should be able to define the sequence of states through which a created prototype can run.
- The prototype should support a mock-up run of the actual system and management of the input and output data.

2. **Structured Analysis and Design.** A CASE tool should support one or more of the structured analysis and design techniques. It should also support making of the fairly complex diagrams and preferably through a hierarchy of levels. The tool must also check the incompleteness, inconsistencies, and anomalies across the design and analysis through all levels of analysis hierarchy.

Analysis and design tools enable a software engineer to create models of the system to be built. The models contain a representation of data, function, and behavior (at the analysis level) and characterizations of data, architectural, component-level, and interface design. By performing consistency and validity checking on the models, analysis and design tools provide a software engineer with some degree of insight into the analysis representation and help to eliminate errors before they propagate into the design, or worse, into the implementation itself.

3. **Code Generation.** A support expected from a CASE tool during the code-generation phase includes the following:

- The CASE tool should support generation of module skeletons or templates in one or more popular programming languages.

- The tool should generate records, structures, and class definitions automatically from the contents of the data dictionary in one or more popular programming languages.
 - It should be able to generate database tables for a relational database-management system.
 - The tools should generate code for the user interface from prototype definitions for X-Windows and Windows-based applications.
4. **Test CASE Generator.** The CASE tool for the test case generator should have the following features:
- It should support both design and requirement testing.
 - It should generate test set reports in ASCII format, which can be directly imported into the test plan document.

In the testing phase, test-management tools are used to control and coordinate software testing for each of the major testing steps. Testing tools manage and coordinate regression testing, perform comparisons that ascertain differences between actual and expected output, and conduct batch testing of programs with interactive human/computer interfaces. In addition to the functions noted, many test-management tools also serve as generic test drivers. A test driver reads one or more test cases from a testing file, formats the test data to conform to the needs of the software being tested, and then invokes the software to be tested.

10.6 OBJECTIVES OF CASE

1. **Improve Productivity.** Most organizations use CASE to increase the speeds with which systems are designed and developed. Imagine the difficulties carpenters would face without hammers and saws. Tools increase the analysts' productivity by reducing the time needed to document, analyze, and construct an information system.
2. **Improve Information System Quality.** When tools improve processes, they usually improve the results as well. They:
 - Ease and improve the testing process through the use of automated checking.
 - Improve the integration of development activities via common methodologies.
 - Improve the quality and completeness of documentation.
 - Help standardize the development process.
 - Improve the management of the project.

- Simplify program maintenance.
- Promote reversibility of modules and documentation.
- Shorten the overall construction process.
- Improve software portability across environments.
- Through reverse engineering and re-engineering, CASE products extend the files of existing systems.

Despite the various driving forces (objectives) for the adoption of CASE, there are many resisting forces that also preclude many organizations from investing in CASE.

3. **Improve Effectiveness.** Effectiveness means doing the right task (i.e., deciding the best task to perform to achieve the desired result). Tools can suggest procedures (the right way) to approach a task. Identifying user requirements, stating them in an understandable form, and communicating them to all interested parties can be an effective development process.
4. **Organizations Reject CASE Because:**
 - The start-up cost of purchasing and using CASE.
 - The high cost of training personnel.
 - The big benefits of using CASE come in the late stages of the SDLC.
 - CASE often lengthens the duration of the early stage of the project.
 - CASE tools cannot easily share information between tools.
 - Lack of methodology standards within organizations. CASE products force analysts to follow a specific methodology for system development.
 - Lack of confidence in CASE products.
 - IS personnel view CASE as a threat to their job security.

Despite these issues, in the long-term, CASE is very good. The functionality of CASE tools is increasing and the costs are coming down. During the next several years, CASE technologies and the market for CASE will begin to mature.

10.7 CASE REPOSITORY

A CASE repository is a system-developer database. Synonyms include dictionary and encyclopedia. It is a place where developers can store system models, detailed descriptions and specifications, and other products of system development.

Analysts use CASE repositories for five important reasons:

- To manage the details in large systems.

- To communicate a common meaning for all system elements.
- To document the features of the system.
- To facilitate analysis of the details in order to evaluate characteristics and determine where system changes should be made.
- To locate errors and omissions in the system.

To limit the amount of narrative needed to describe relationships between data items and at the same time to show the structural relationship clearly, analysts often use formal notation in the data dictionary, a component of a CASE repository.

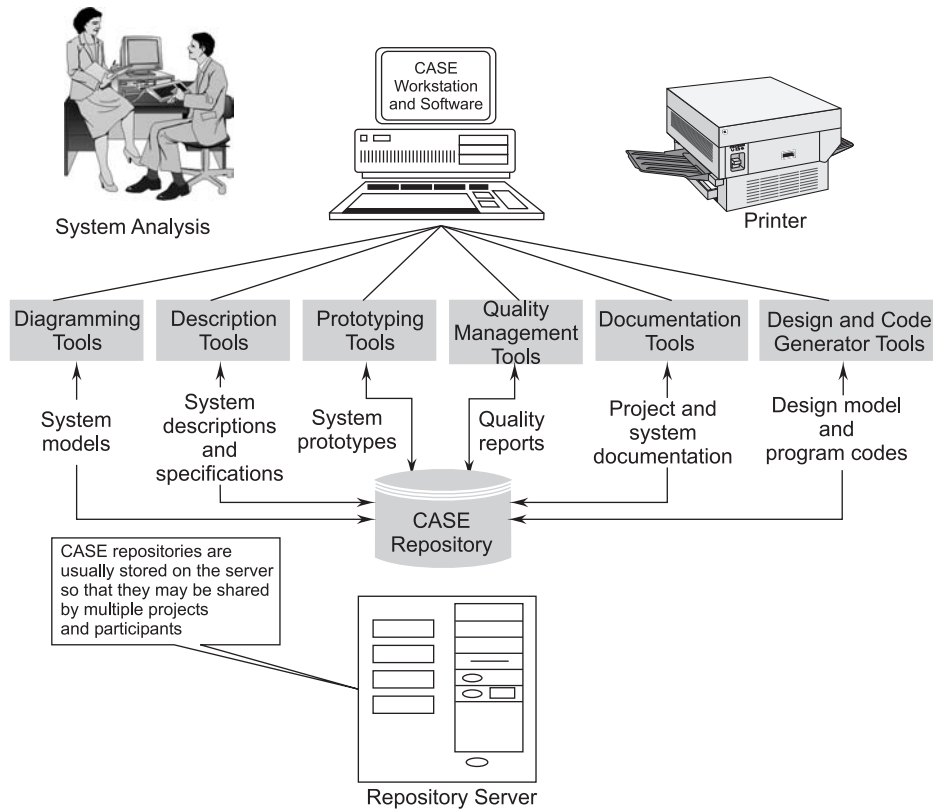


FIGURE 10.3 Case Repository

A data dictionary can be developed manually or using automated systems. Automated systems offer the advantage of automatically producing data elements, data structures, and process listings; they also perform cross-reference checking and error detection. The data dictionary is a repository of all data definitions for all organizational applications and is used to manage and control access

to the information repository, another component of the CASE repository. The information repository provides automated tools used to manage and control access to business information and application portfolios.

The CASE repository is an idea central to I-CASE. Integrated-CASE tools rely on common terminology, notations, and methods for system development across all tools. Within an I-CASE environment, all diagrams, forms, reports, and programs can be automatically updated by the single change to the data-dictionary definition. Besides specific tool integration, there are two additional advantages of using a comprehensive CASE repository that relate to project management and reusability. The CASE repository provides a wealth of information to the project manager and allows the manager to exert an appropriate amount of control over the project. If all organizational systems were created using CASE technology with a common repository, it would be possible to reuse significant portions of prior systems in the development of new ones.

10.8 CHARACTERISTICS OF CASE TOOLS

All CASE tools have the following characteristics:

- A graphic interface to draw diagrams, charts, models (uppercase, middlecase, lowercase).
- An information repository, a data dictionary for efficient information-management selection, usage, application, and storage.
- Common user interface for integration of multiple tools used in various phases.
- Automatic code generators.
- Automatic testing tools.

10.9 CASE CLASSIFICATION

CASE classifications help us understand the different types of CASE tools and their role in supporting software process activities. There are various ways of classifying CASE tools, each of which gives us a different perspective on these tools. In this section, we discuss CASE tools from three of these perspectives, namely:

- A functional perspective where CASE tools are classified according to their specific function.

- A process perspective where tools are classified according to the process activities which they support.
- An integration perspective where CASE tools are classified according to how they are organized into integrated units which provide support for one or more process activities.

10.9.1 List of CASE Tools

S.No.	Application	CASE Tool	Purpose of Tool
1.	Planning	Excel spreadsheet, MS Project, PERT/CPM Network, Estimation tools	Functional Application: Planning, scheduling, control
2.	Editing	Diagram editors, Text editors, Word processors	Speed and Efficiency
3.	Testing	Test-data generators, File comparators	Speed and Efficiency
4.	Prototyping	High-level modeling language, User-interface generators	Confirmation and certification of RDD and SRS
5.	Documentation	Report generators, Publishing imaging, PPT presentation	Faster structural documentation with quality of presentation
6.	Programming and Language-processing integration	Program generators, Code generators, Compilers, Interpreters interface, connectivity	Programming of high quality with no errors, system integration
7.	Templates	—	Guided systematic development
8.	Re-engineering tools	Cross-references systems, program re-structuring systems	Reverse-engineering to find structure, design, and design information
9.	Program analysis tools	Cross-reference generators Static analyzers, dynamic analyzers	Analyzes risks, functions, features

10.10 CATEGORIES OF CASE TOOLS

The schematic diagram of CASE tools is shown in Figure 10.4.

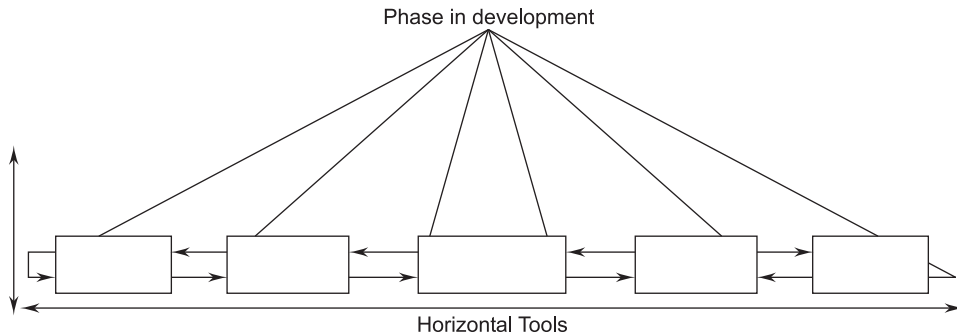


FIGURE 10.4 Categories of CASE Tools

CASE tools are divided into the following two categories:

1. Vertical CASE tools
2. Horizontal CASE tools

1. **Vertical CASE Tools.** Vertical CASE tools provide support for certain activities within a single phase of the software life-cycle.

There are two subcategories of vertical CASE tools:

- (i) *First Category.* It is the set of tools that are within one phase of the life-cycle. These tools are important so that the development in each phase can be done as quickly as possible.
- (ii) *Second Category.* It is a tool that is used in more than one phase, but does not support moving from one phase to the next. These tools ensure that the developer does move onto the next phase as appropriate.

2. **Horizontal CASE Tools.** These tools support the automated transfer of information between the phases of a life-cycle. These tools include project management, configuration-management tools, and integration services.

The above two categories of CASE tools can further be broken down into the following:

- (a) **Upper CASE Tools/Front-end CASE Tools.** These CASE tools are designed to support the analysis and design phases of the SDLC. All analysis, design, and specification tools are front-end tools. These tools also include computer-aided diagramming tools oriented towards

a particular programming design methodology, and more recently including object-oriented design.

The general types of upper CASE tools are listed below:

- *Diagramming tools*: Diagramming tools enable system process, data, and control structures to be represented graphically. They strongly support analysis and documentation of application requirements.
- *Form and report generator tools*: They support the creation of system forms and reports in order to show how systems will “look and feel” to users.
- *Analysis tools*: Analysis tools enable automatic checking for incomplete, inconsistent, or incorrect specifications in diagrams, forms, and reports.

- (b) **Lower CASE or Back-end Tools**. These CASE tools are designed to support the implementation and maintenance phases of the SDLC. All generator, translation, and testing tools are back-end tools.

The general types of lower CASE tools are:

- *Code Generators*: Code generators automate the preparation of computer software. Code generation is not yet perfect. Thus, the best generator will produce approximately 75 % of the source code for an application. Hand-coding is still necessary.
- (c) **Cross life-cycle CASE or Integrated Tools**. These CASE tools are used to support activities that occur across multiple phases of the SDLC. While such tools include both front-end and back-end capabilities, they also facilitate design, management, and maintenance of code. In addition, they provide an efficient environment for the creation, storage, manipulation, and documentation of systems.
- (d) **Reverse-engineering Tools**. These tools build bridges from lower CASE tools to upper CASE tools. They help in the process of analyzing existing applications and perform and database code to create higher level representations of the code.

10.11 ADVANTAGES OF CASE TOOLS

The major benefits of using CASE tools include the following:

- Improved productivity
- Better documentation
- Improved accuracy

- Intangible benefits
- Improved quality
- Reduced lifetime maintenance
- Opportunity to non-programmers
- Reduced cost of software
- Produce high-quality and consistent documents
- Impact on the style of a working of company
- Reduce the drudgery in a software engineer's work
- Increase speed of processing
- Easy to program software
- Improved coordination among staff members who are working on a large software project
- An increase in project control through better planning, monitoring, and communication

10.12 DISADVANTAGES OF CASE TOOLS

1. **Purchasing of CASE Tools is Not an Easy Task.** The cost of CASE tools is very high. For this reason small software development firms do not invest in CASE tools.
2. **Learning Curve.** In general, programmer productivity may fall in the initial phase of implementation as users need time to learn this technology.
3. **Tool Mix.** It is important to make proper selection of CASE tools to get maximum benefits from the tools, as the wrong selection may lead to the wrong results.

10.13 REVERSE SOFTWARE ENGINEERING

10.13.1 Definition

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.

10.13.2 Purpose of Reverse Engineering

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.

10.13.3 Reverse-Engineering Process

The reverse-engineering process is illustrated in Figure 10.5. The process starts with an analysis phase. During this phase, the system is analyzed using automated tools to discover its structure. In itself, this is not enough to re-create the system design. Engineers then work with the system source code and its structural model. They add information to this, which they have collected by understanding the system. This information is maintained as a directed graph that is linked to the program source code.

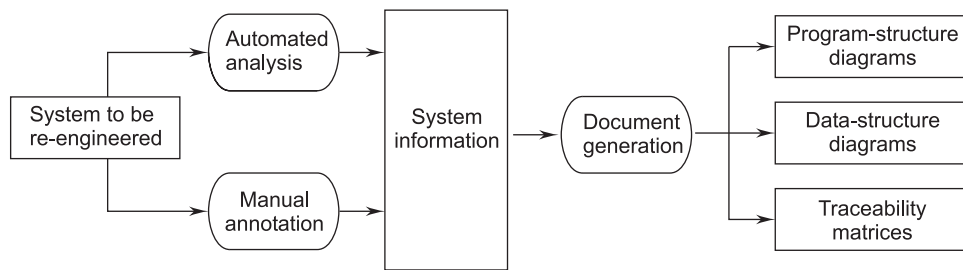


FIGURE 10.5 The Reverse-engineering Process

Information-store browsers are used to compare the graph structure and the code and to annotate the graph with extra information. Documents of various types, such as program and data structure diagrams and traceability matrices can be generated from the directed graph. Traceability matrices show where entities in the system are defined and referenced. The process of document generation is an iterative one as the design information is used to further refine the information held in the system repository.

10.13.4 Reverse-Engineering Tasks

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:

1. **Mapping between application and program domains.** The task of the reverse engineer is to reconstruct the mapping from the application domain to the program domain as shown in Figure 10.6.

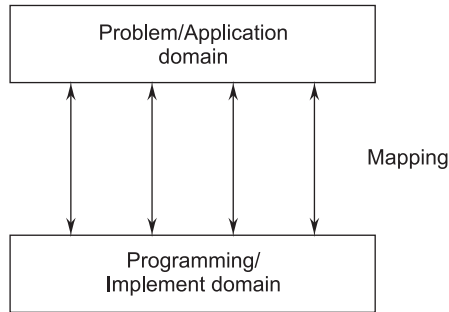


FIGURE 10.6 Mapping Between Application and Domain Programs

2. **Mapping between concrete and abstract levels.** The software-development process goes 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.
3. **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 over the 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.
4. **Finding missing links between program 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. The reverse-engineering process should determine the semantics of a given program from its syntax.
5. **To extract reusable components.** Based on the premise that the use of existing program components can lead to an increase in productivity and improvement in product quality, the concept of reuse has increasingly become popular among software engineers.

10.13.5 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 “re-documentation.” If on the other hand, the resulting product is at a higher level of abstraction, the operation is known as “design recovery” or specification recovery as shown in Figure 10.7.

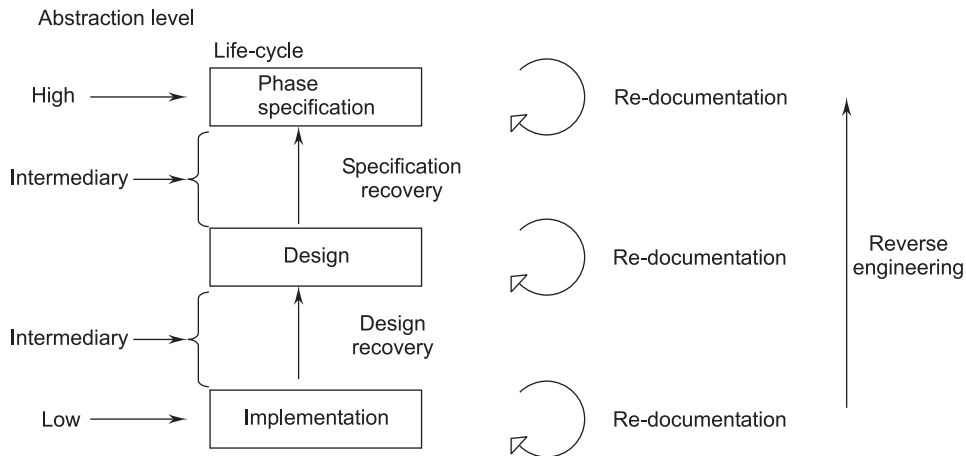


FIGURE 10.7 Levels of Abstraction

1. **Re-documentation.** Re-documentation is the recreation of a semantically equivalent representation within the same relative abstraction level. The goals of this process are threefold:
 - Firstly, to create alternative views of the system as to enhance understanding; for example, the generation of hierarchical data flows or control-flow diagrams 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.
2. **Design Recovery.** Design recovery entails identifying and extracting meaningful higher-level abstractions beyond those obtained directly from examination of the source code. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains. The recovered design, which is not necessarily the original design, can then be used for redeveloping the system.

10.13.6 Characteristics of Reverse Engineering

The various characteristics of reverse software engineering are as follows:

1. **Abstraction Level.** The abstraction level of a reverse-engineering process and the tools used to affect it refers to the sophistication of the design information that can be extracted from the source code. Ideally, the abstraction level should be as high as possible. As the abstraction level increases the software engineer is provided with information that will allow easier understanding of the program.
2. **Completeness.** The completeness of a reverse-engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases. For example, given a source code listing, it is relatively easy to develop a complete procedural design representation. Simple design representations may also be derived, but it is far more difficult to develop a complete set of UML diagrams or models.
3. **Interactivity.** Interactivity refers to the degree to which the human is “integrated” with automated tools to create an effective reverse-engineering process. In most cases, as the abstraction level increases, interactivity must increase or completeness will suffer.
4. **Directionality.** If the directionality of the reverse-engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a re-engineering tool that attempts to restructure or regenerate the old program.
5. **Extract Abstractions.** The core of reverse engineering is an activity called extract abstractions. The engineer must evaluate the old program and from the (often undocumented) source code, develop a meaningful specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

10.13.7 Application Areas of Reverse Engineering

The different application areas of reverse software engineering include:

- Program comprehension
- Re-documentation
- Recovery of design approach and design details at any level of abstraction
- Identifying re-usable components
- Identifying components that need restructuring
- Recovery business rules

10.14 SOFTWARE RE-ENGINEERING

10.14.1 Introduction to Re-Engineering

Re-engineering means to re-implement systems to make them more maintainable.

In re-engineering, the functionality and system architecture remains the same but it includes re-documenting, organizing and restricting, modifying and updating the system. It is a solution to the problems of system evolution. In other words,

Re-engineering essentially means having a re-look at an entity, such as a process, task, designs, approach, or strategy using engineering principles to bring in radical and dramatic improvements.

The re-engineering approach attacks five parameters, namely: management philosophy, pride, policy, procedures, and practices to bring in radical improvements impacting cost, quality, service, and speed. When re-engineering principles are applied to business process then it is called Business Process Re-engineering (BRP).

10.14.2 Principles of Software Re-engineering

The principles of re-engineering when applied to software-development processes are called software re-engineering. It affects positively software cost, quality, service to the customer, and speed of delivery. Software, whether a product or system, deals with business processes making them faster, smarter, and automatic in response to delivery and execution. In software re-engineering, we may resort to one or more of the following:

- Redefining software scope and goals.
- Redefining RDD and SRS by way of additions, deletions, and extensions of functions and features.
- Redesigning the application design and architecture using new technology, upgrades, and platforms, interfacing to new technologies to make the process faster, smarter, and automatic.
- Resorting to data restructuring, improving database design, code restructuring to make the size smaller and more efficient in operations.
- Rewriting the documentation to make it more user friendly.

10.14.3 Re-engineering Process

Figure 10.8 illustrates a possible re-engineering process. The input to the process is a legacy program and the output is a structured, modularized version of the

same program. At the same time as program re-engineering, the data for the system may also be re-engineered.

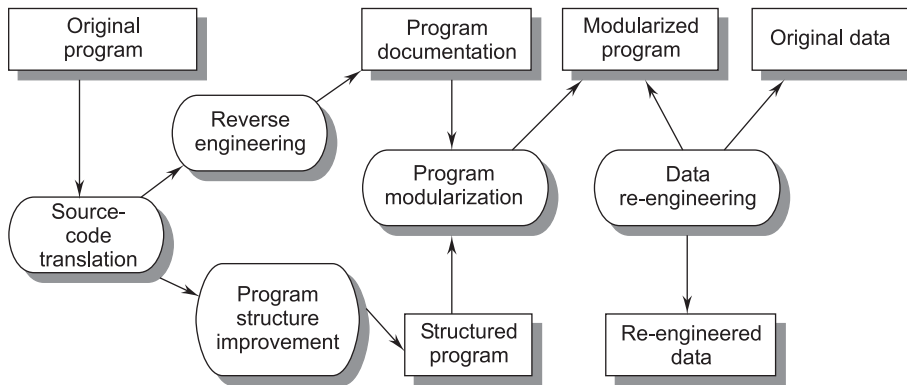


FIGURE 10.8 Re-engineering Process

The re-engineering process includes the following activities:

- **Source-code translation:** In source-code translation the programming language of an old program is converted into the modern version of the same language or to a new language.
- **Reverse engineering:** In reverse engineering the program is analyzed and important and useful information is extracted from it which helps to document its functionality.
- **Program structure improvement:** In program structure improvement the control structure of the program is analyzed and modified to make it easier to read and understand.
- **Program modularization:** In program modularization redundancy of any part is removed and related parts are grouped together.
- **Data re-engineering:** In data re-engineering the data processed by the program is changed to reflect program changes.

10.14.4 Software Re-engineering Process Model

Re-engineering of information systems is an activity that will absorb information technology resources for many years. That's why every organization needs a pragmatic strategy for software re-engineering.

Re-engineering is a rebuilding activity. To implement re-engineering principles we apply a software re-engineering process model. The re-engineering paradigm shown in Figure 10.9 is a cyclical model.

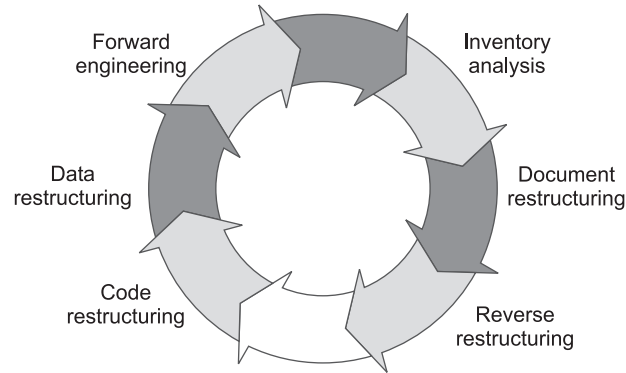


FIGURE 10.9 Software Re-engineering Process Model

There are six activities in the model:

1. Inventory analysis
2. Document restructuring
3. Reverse engineering
4. Code restructuring
5. Data restructuring
6. Forward engineering

The activities of the software re-engineering process model are described as follows:

1. **Inventory Analysis.** Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability, and other locally important criteria, candidates for re-engineering appear. Resources can then be allocated to candidate applications for re-engineering work.

It is important to note that the inventory should be revisited on a regular cycle. The status of applications (e.g., business criticality) can change as a function of time and as a result, priorities for re-engineering will shift.

2. **Document Restructuring.** Weak documentation is the trademark of many legacy systems. But what do we do about it? What are our options?

(i) *Creating documentation is far too time consuming:* If the system works, we'll live with what we have. In some cases, this is the correct approach. It is not possible to recreate documentation for hundreds

of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change, let it be!

- (ii) *Documentation must be updated, but we have limited resources:* We'll use a "document when touched" approach. It may not be necessary to fully re-document an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.
 - (iii) *The system is business critical and must be fully re-documented:* Even in this case, an intelligent approach is to pare documentation down to an essential minimum.
 - (iv) *Each of these options is viable.* A software organization must choose the one that is most appropriate for each case.
3. **Reverse Engineering.** Reverse engineering for software is quite similar. In most cases, however, the program to be reverse engineered is not a competitor's. Rather, it is the company's own work (often done many years earlier). The "secrets" to be understood are obscure because no specification was ever developed. Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse-engineering is a process of design recovery. Reverse-engineering tools extract data and architectural and procedural design information from an existing program.
 4. **Code Restructuring.** The most common type of re-engineering is code restructuring. Some legacy systems have relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted, and code is then restructured (this can be done automatically). The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.
 5. **Data Restructuring.** A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, data architecture has more to do with the long-term viability of a program than the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring is a full-scale re-engineering activity. In most cases, data restructuring begins with a reverse-engineering activity. Current data architecture is dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

6. **Forward Engineering.** Applications would be rebuilt using an automated “re-engineering engine.” The old program would be fed into the engine analyzed, restructured, and then regenerated in a form that exhibited the best aspects of a software quality. CASE vendors have introduced tools that provide a limited subset of these capabilities that address specific application domains. Forward engineering, also called renovation or reclamation, not only recovers design information from existing software, but also uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.

10.14.5 Factors Affecting Re-engineering Costs

Factors that affect re-engineering costs include:

1. *Quality of software to be re-engineered:* There is the inverse relationship between the quality and the cost of the software.
2. *Tools available for re-engineering:* It is not cost effective to re-engineer a software system unless you can use CASE tools to automate most of the program changes.
3. *Availability of expert staff:* The re-engineering staff is not the same as the maintaining staff, and this will increase costs.
4. *Extent of data conversion required:* There is a direct relationship between the volume of data to be converted and the cost of the software.

10.14.6 Differences Between Forward Engineering and Re-engineering

Chikofsky and Cross (1990) call conventional development forward engineering to distinguish it from software re-engineering. This distinction is illustrated in Figure 10.10. Forward engineering starts with a system specification and involves the design and implementation of a new system. Re-engineering starts with an existing system and the development process for the replacement is based on the understanding and transformation of the original system.

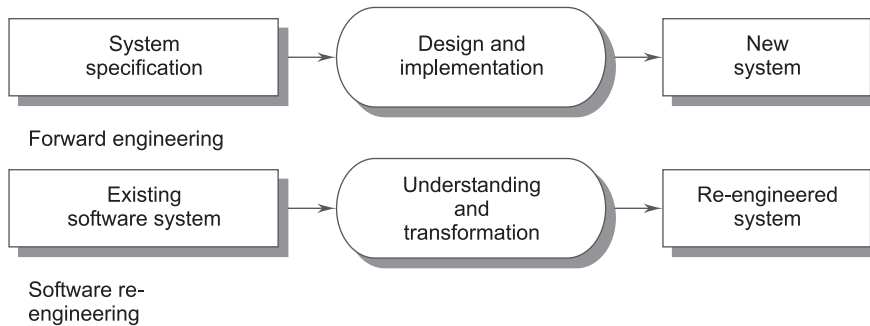


FIGURE 10.10 Forward Engineering and Re-Engineering

10.14.7 Advantages and Disadvantages

Re-engineering a software system has two key advantages over more radical approaches to system evolution.

- (i) **Reduced risk:** There is a high risk in redeveloping software that is essential for an organization. Errors may be made in the system specification; there may be development problems, etc.
- (ii) **Reduced costs:** The costs of re-engineering is significantly less than the costs of developing new software. Ulrich (1990) quotes an example of a commercial system where the re-implementation costs were estimated at 550 million. The system was successfully re-engineered for \$12 million. If these figures are typical, it is about four times cheaper to re-engineer than to rewrite.

The main disadvantages of software re-engineering are that there are practical limits to the extent that a system can be improved by re-engineering. It isn't possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes of radical reorganizing of the system-data management cannot be carried out automatically, so involve high additional costs. Although re-engineering can improve maintainability, the re-engineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

EXERCISES

1. Explain Computer-Aided Software Engineering (CASE) and the various types of CASE tools.
2. Explain the three most used CASE tools?
3. What are the advantages and disadvantages of CASE tools?

4. What are the different categories of CASE tools?
5. Explain how CASE supports a software-life-cycle.
6. Explain the building blocks of CASE.
7. What are the important characteristics of CASE tools?
8. What are the main advantages of using CASE tools? Describe some of the important features that a future generation CASE tool should support.
9. What is a CASE tool and a CASE environment? Why do integration tools increase the power of tools? Explain using some examples.
10. Give the main advantages of using CASE tools.
11. What is CASE? If a large complex software solution is required, why is the CASE approach recommended? How does the CASE approach affect the following:
 - (i) Requirements
 - (ii) Risk
 - (iii) Report generation
 - (iv) Documentation
 - (v) Programming efforts
12. Draw the schematic diagram of the architecture of a CASE environment and explain how the different tools are integrated.
13. Suggest potential benefits and practical problems of integrating CASE tools.
14. Give the architecture of a CASE environment.
15. Define re-engineering.
16. Define reverse engineering.
17. Discuss the levels of reverse engineering.
18. Differentiate between re-engineering and new development.
19. What are appropriate reverse engineering tools? Discuss any two tools in detail.
20. Give the differences between reverse engineering and re-engineering?

Chapter 11

CODING

11.1 INFORMATION HIDING

In the application of the information-hiding principle, the data structure is not directly used by other modules; it is used only through access functions.

The advantage of information hiding and decoupling data structures from the processing module is that such a system is easy to maintain, be there is a change in structure or process. That is, if the data structure is changed, the modification is limited to the structure and access function, and modules are not affected at all. The same is true when the structure is left intact but the process needs to be changed.

In object-oriented programming, the principle of information hiding is extensively used. In this approach, information that is not needed in that module is hidden from the module. The advantage is that the module controls the data hidden in it. Other modules are not allowed to access or modify the data.

It is defined as information captured in the data structure that should be hidden from the rest of the system. If it is used and some data structure is changed, then its effect is limited to the access to change information hiding that is supported by the object-oriented language.

Example:

```

class person
{
private:
    int name, age, qualification;
public:
    void person (int n, int a, int q)
    {
        name = n;
        age = a;
        qualification = q;
    }
    void display()
    {
        cout <<"name="<< name;
        cout <<"Age= "<< age;
        cout<< "Qualification = " << qualification;
    }
};

```

In the above example, the information is hidden with the help of a “private” access specifier, which is used for hiding the information accessed by the other classes. In this the variable’s name, age, and qualifications are initialized under the “private” keyword, so they are only accessed by the class “person,” i.e., the three data items name, age, qualifications are hidden with the help of the class concept.

Information hiding can reduce the coupling between modules and make the system more maintainable. Information hiding is also an effective tool for managing the complexity of developing software—by using information hiding we have separated the concern of managing the data from the concern of using the data to produce some desired results.

11.2 PROGRAMMING STYLE

The aim of a programming style is to optimize the code with desired results. This must be presented in the best possible manner. Now we will list some general rules with respect to programming style.

1. **Naming.** In a program, you are required to name the module, processes, and variables and so on. The naming style should not be cryptic and non-representative.

The name should represent the entity completely without confusion. Avoid cryptic names, unknown acronyms, or names totally unrelated to the entity. For example, purchase order should be named PO and not PRO, POrder, Vendor PO, and so on.

2. **Control Constructs.** It is desirable that as many single-entry and single-exist constructs as possible be used. In every language there are few control constructs, and we should use a few standard control constructs rather than using a wide variety of control constructs.
3. **Information Hiding.** Only access functions to data should be made visible and data should be hidden behind these functions.
4. **Gotos.** Gotos should be used sparingly and in a disciplined manner. Only when the alternative using gotos is more complex then gotos should be used. In any case, alternatives must be thought of before finally using a goto. If a goto must be used, forward transfers (or a jump to a later statement) are more acceptable than a backward jump.
5. **User-defined Type.** Modern languages allow users to define types, such as the enumerated type. When such facilities are available, these should be exploited where applicable.
6. **Nesting.** In control constructs, 'if-then-else' statements are used extensively to construct a control, based on given conditions. If the condition is satisfied one action is proposed; if not, then another action is proposed. If this condition-based nesting is too deep, the code becomes very complex. Let us use the example of pricing a product for a customer. Nesting in this control is as follows:

```

    If customer type large then price P1
    else medium then price P2
    else small then price P3

```

Instead of this, we can exercise the control in the following manner:

```

    if customer type large then price P1
    if customer type medium then price P2
    if customer type small then price P3

```

In both cases, the control construct will produce the same result.

7. **Module Size.** The module size should be uniform. Its size should not be too small or too big. If the module is too large, it is not functionally cohesive and if it is too small it might lead to unnecessary overhead. Module size should be based on the principle of cohesion and coupling.
8. **Program Layout.** A good layout is one that helps to read the program faster and to understand it better. The layout should be organized using proper indentation, blank spaces, and parentheses to enhance readability.

9. **Module Interface.** A module with a complex interface (has multiple functions) should be carefully examined and avoided. If a complex interface has more than five parameters then it should be carefully examined and they must be broken into multiple modules with a simpler interface.
10. **Robustness.** A program is robust if it does something planned even for exceptional conditions. A program might encounter exceptional conditions in such forms as incorrect input, the incorrect value of some variable, and overflow. If such situations do arise, the program should not just “crash” or “core dump”; it should produce some meaningful message and exit gracefully.
11. **Side Effects.** When a module is invoked, it sometimes has side effects of modifying the program state beyond the modification of parameters listed in the module interface definition. Such side effects should be avoided wherever possible and if a module has side effects, they should be properly documented.

11.3 INTERNAL DOCUMENTATION

Program documentation is one of two types: external, addressing information about the program and internal, which has to be close to the program, program statement, and program block to explain it, then and there. Internal documentation of the program is done through the use of comments. All programming languages provide a means of writing comments in the program. The comment is a text written for the user, reader, or programmer to easily understand and it is executed in any manner. A comment generally helps at the time of maintenance. It not only explains the program, or program statement, but also provides points on caution, condition of applicability, and assumptions considered important for programmers to know before any action is taken for modification.

Whether it is the entire program of the module or a block of programs or a statement in the program, comments should provide information on the following:

- Functionality
- Parameters and their role and usage
- Attributers of inputs, i.e., assumptions values, range, max and min, etc.
- Mention of global variables

In addition to this information, referential data should be provided, such as:

- When modified last

- Author of the program
- Information on compilation and testing

The main objective of internal documentation is to provide on-line help to the user and programmer to get a quick understanding of the program and the problem and to enable them to modify the program as fast as possible.

Internal documentation is comprised of the aspects of programs which are included in the syntax of the programming language. The important points are the:

- Meaningful names used to describe data items and procedures
- Comments relating to the function of the program as a whole and of the modules comprising the program
- Clarity of style and formal, i.e., one instruction per line, indentation of related groups of instructions, blank lines separating modules
- Use of symbolic names instead of constants or literals in the procedural code

11.4 MONITORING AND CONTROL FOR CODING

Code reviews are mainly designed to detect defects that are originated during the coding phase; they can also be used to detect defects in the detached design. It was started with the purpose of detecting defects in the code.

After the successful completion of code, code inspection and reviews are held. Activities, such as code reading, symbolic execution, and static analysis should be performed and defects found by these techniques should be corrected before starting the code reviews. The main aim of this is to save human time and effort.

Code defects can be divided into two groups: logic and control defects and data operations and computations defects. Examples of logic and control defects are unreachable code, incorrect predicate, infinite loops, improper nesting of loops, unreferenced labels, etc.

Examples of data operations and computations defects are incorrect access of array components, missing validity tests for external data, improper initialization, misuse of variables, etc.

The following are some of the items that can be included in a checklist for a code review:

- When are the divisors tested for zero applicable?
- Is important data tested for being validated?
- Is the loop termination condition right?

- Are the number of loop executions “off by one”?
- Are the pointers set to NULL where ever needed?
- Are indexes initialized?
- Are all variables used?
- Are the arrays indexed within bounds?
- Are all output variables assigned?
- Are the local coding standards met?
- Are all branch conditions right?
- Are the labels unreferenced?
- Do data definitions exploit the typing capabilities?
- Do all pointers point to some object?
- Will a loop always terminate?
- Can statements placed in the loop be placed outside the loop?

11.5 STRUCTURED PROGRAMMING

Structured programming refers to a general methodology of writing good programs. A good program is one that has the following properties:

1. It should perform all the desired actions.
2. It should be reliable, i.e., perform the required actions within acceptable margins of error.
3. It should be clear, i.e., easy to read and understand.
4. It should be easy to modify.
5. It should be implemented within the specified schedule and budget.

Structured programs have the single-entry, single-exit property. This feature helps in reducing the number of paths for flow of control. If there are arbitrary paths for the flow of control, the program will be difficult to read, understand, debug, and maintain.

A program is one of two types: static structure or dynamic structure. The static structure is the structure of the text of the program, which is usually just a linear organization of statements of the program. The dynamic structure of the program is the sequence of statements executed during the execution of the program.

Both static and dynamic structures are the sequence of statements. The only difference is that the sequence of statements in a static structure is fixed, whereas in a dynamic structure it is not fixed. That means the dynamic sequence of statements can change from execution to execution. The static structure of a program can be

easily understood. The dynamic structure of a program can be easily seen at the time of execution.

11.5.1 Objectives of Structured Programming

The goal of structured programming is to ensure that the static structures and the dynamic structures are the same. That is, the objective of structured programming is to write programs so that the sequence of statements executed during the execution of a program is the same as the sequence of statements in the text of that program. As the statements in a program text are linearly organized, the objective of structured programming is to develop programs whose control flow during execution is linearized and follows the linear organization of the program text.

Clearly, no meaningful program can be written as a sequence of simple statements without any branching or repetition (which also involves branching). So, how is the objective of linearizing the control flow to be achieved? By making use of structured constructs. In structured programming, a statement is not a simple assignment statement, it is a structured statement.

11.5.2 Principles of Structured Programming

All structured program design methods are based upon the two fundamental principles stepwise refinement and three structured control constructs. The objective of program design is to transform the required function of the program, as stated in the program specification, into a set of instructions, which can easily be translated into a chosen programming language. The process of stepwise refinement is such an approach that the stated program function is broken down into subsidiary functions in progressively increasing levels of detail until the lowest level functions are achievable in the programming language.

The second principle of structured program design is that any program can be constructed using only three structured control constructs. The constructs selection, iterations, and sequence are shown in Figure 11.1 (*a, b, c, d*).

Any program independent of the technology platform can be written using these constructs, i.e., selection, repetition, sequence. These structures are the basis of structured programming.

The sequence construct is shown in Figure 11.1 (*b*) connecting two boxes of tasks by an arrow. The selection construct is shown in Figure 11.1 (*a*), where process is selected as the condition of the states indicated by the X and Y comparison.

An iteration is a program component which has only one part, occurring zero or more times. The number of times the subcomponent will be executed depends upon when the condition becomes false. This situation can occur on first entry or after a number of repetitions of the program.

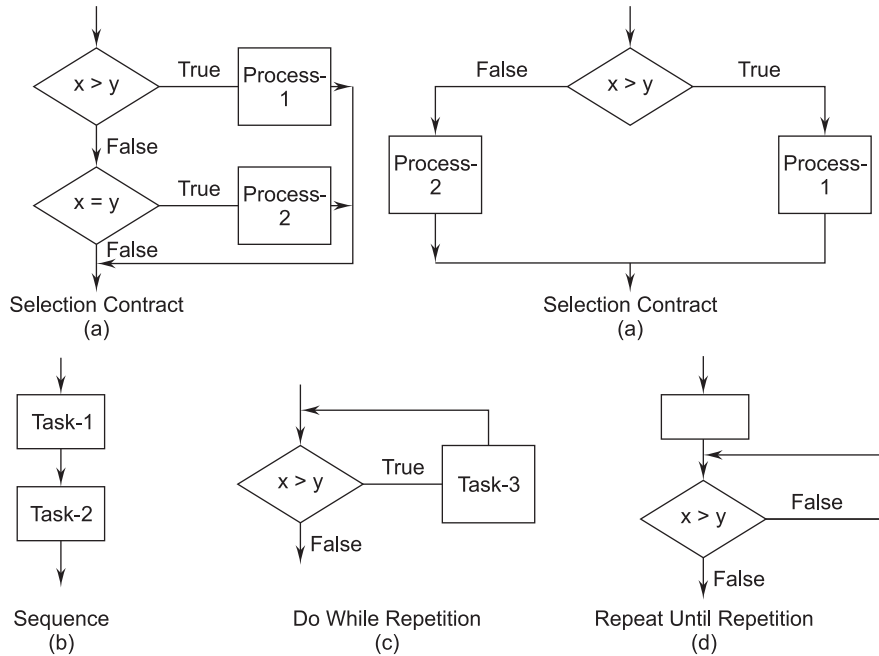


FIGURE 11.1 Basics of Structured Programming: Selection, Iterations, and Sequence

Figure 11.1 (c and d) show repeat constructs of two types.

The fourth category of construct is nesting, made of the three basic constructs discussed so far. Figure 11.2 shows the nesting construct.

When we nest the construct, we are changing the layers of procedural design.

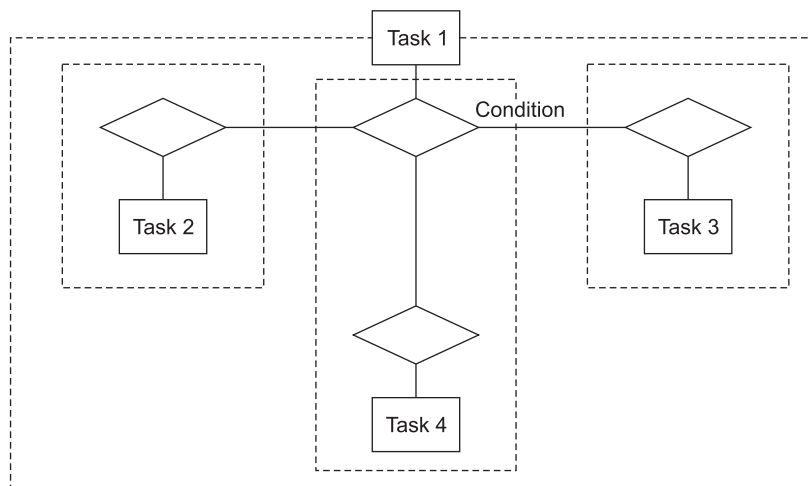


FIGURE 11.2 Nesting Construct

11.5.3 Key Features of Structured Programming

The key feature of a structured program is that it has a single-entry and a single-exit. We can therefore study the program statement by statements and in sequence. The most commonly used single entry and single exit statements are:

Selection: if customer type is X
 then price is \$100
 else price is \$90

Iteration: While customer type is X do use price formula P1. Repeat P1 until type is X.

Sequencing: Task 1, Task 2, Task 3.

Extensive use of these statements in the program construct creates a linear flow. If readability and verification are the essence of a good construct, then a structured program is the method to achieve it.

11.5.4 Advantages of Structured Programming

The advantage of structured programming is that it is very convenient to put logic systematically into the program. Due to the ease of handling complex logic, the user, reader, and programmer understand the program easily.

Another distinct advantage of structured programming is that it is easy to verify, conduct reviews, and test the structured programs in an orderly manner. If errors are found, they are easy to locate and correct.

11.6 FOURTH-GENERATION TECHNIQUES

Fourth-Generation Technique (4GT) includes the application of tools and techniques for expeditious development of software solutions. Besides expeditious development, fourth-generation techniques help to control effort, resources, and cost of development. Commonly used fourth-generation techniques in development models are mentioned here. They are:

- Report Generation
- Database Query Languages
- Data Manipulation
- Screen Definition and Interaction
- Code Generation
- Connectivity, such as ODBC, JDBC, and Interfacing
- Web-engineering Tools

- Graphics, Spreadsheet Generation Capability
- CASE Tools

The application of 4GT tools calls for a specific environment and also the requisite specifications of requirement, design, and architecture. These specifications help successful application of these tools.

The 4GT paradigm for software engineering focuses on the ability to specify software using specialized language forms or a generic notation that describes the problem to be solved in terms that the customer can understand.

For small applications, it may be possible to move directly from the requirements gathering step to implementation using a non-procedural fourth-generation language (4GL). However, for larger efforts, it is necessary to develop a design strategy for the system, even if a 4GL is to be used.

To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software-engineering paradigms. In addition, the 4GT-developed software must be built in a manner that enables maintenance to be performed expeditiously.

11.6.1 Use of Fourth-Generation Techniques

4GT has a number of query languages, report writers, generators, program and application generators, and high-end object-oriented languages. Extensive use of 4GTs saves a lot of software-specific coding and allows rapid prototype development.

11.6.2 Advantages of 4GT

The advantage of the 4GT model is a dramatic reduction in software development time and greatly improved productivity for people who build software.

11.6.3 Disadvantages of 4GT

With very few exceptions, the current application domain for 4GT is limited to business-information system applications. The use of 4GT for large software development efforts demands as much or more analysis, design, and testing to achieve substantial timesaving that can be achieved through the elimination of coding.

11.6.4 Difference Between 3GLs and 4GLs

The various differences between 3GLs and 4GLs are given in Table 11.1.

TABLE 11.1 Differences Between 3GLs and 4GLs

Third-generation languages	Fourth-generation languages
Professional programmers are required to use this language.	May be used by a non-programming end user as well as a professional programmer.
Requires specification for how to perform tasks.	Requires specification for what task is to be performed (system determines how to perform the task).
All alternatives must be specified.	Default alternatives are built in; an end user needs not specify these alternatives.
Requires a large number of procedural instructions.	Requires far fewer instructions.
Code may be difficult to read, understand, and maintain.	Code is easy to understand and maintain because of English-like commands.
Language developed for batch operation.	Language developed primarily for on-line use.
Can be difficult to learn.	Easy to learn.
Difficult to debug.	Easy to debug.
Typically file-oriented.	Typically database-oriented.

EXERCISES

1. What is “information hiding” in programming?
2. Describe the various programming styles in software engineering.
3. What is “structured programming”? How do modern programming languages, such as PASCAL and C facilitate writing structured programs?
4. What are the advantages of writing structured programs versus unstructured programs?
5. Discuss some methods of monitoring and control of the software development process.
6. Enumerate the term monitoring and control in the context of coding in software development.
7. What is a fourth-generation language? How does it differ from a third-generation language?
8. What is a fourth-generation programming technique? What are its advantages and disadvantages compared to the third-generation technique?
9. Applications developed using 4GLs would normally be more efficient and run faster compared to applications developed using 3GLs. Discuss.

PART II

SOFTWARE DEVELOPMENT AND APPLICATIONS

Programming has always been considered an art. The mastery of software can be both easy and difficult. It depends on how much you are planning to learn. If you are learning only for personal use, then it is very easy.

Programmers, for decades, have struggled with volumes of code in an attempt to make their programs easy to use along with possessing the required functionality.

This part has been written to make you aware of the software options available and how they can be used for software development, and it also offers concepts, tools, and knowledge fortifiers, which aim to make your learning of Visual Programming with Visual Basic complete.

Chapter 12

INTRODUCTION TO SOFTWARE DEVELOPMENT

A sequence of commands used to generate a desired result is a program. The output depends upon the input given. We may have different sets of commands to achieve the same result.

A program is written in a computer language. There are several languages and GUIs that can be used to write a program, i.e., Cobol, Fortran, C, C++, Java, Visual Basic, and Visual Basic.NET, but we will discuss here only Visual Basic.

Before writing a program you must clearly understand the actual problem or requirement, and then you will write the program instructions to generate the desired output.



Let us write a small program to multiply two numbers. The different steps involved are:

- Step 1:** Input first number
- Step 2:** Input second number
- Step 3:** Processing (multiplying) the input numbers
- Step 4:** Store the processed data
- Step 5:** Display the stored result

12.1 PROGRAM PHASE

The different phases involved in writing a complete program are:

1. Identification of the exact problem
2. Development of a mathematical model
3. Design of an algorithm
4. Testing of the algorithm
5. Coding
6. Testing of the program
7. Documentation

12.2 HOW TO WRITE A GOOD PROGRAM

Use the following instructions to write a good program:

1. **Readability:** The program should be written in such a manner that it can be understood by other programmers easily. Follow some standards when writing the program. Never create your own style if other programmers have to work on the same program at the same time or later. Always use explanatory comments while writing the program. Variables should be named in such a manner that their use in the program can be understood easily. A long program should be divided into small sections and subroutines. There should be proper use of functions to avoid lengthy programs.
2. **Design:** Program design is the most important phase because it affects all the phases. The efficiency of the program depends upon the design. There should be simplicity and adaptability in the program and it should fulfill all the requirements of the user and must give the desired output.
3. **Efficiency:** Program efficiency is very important in program development. The program should be very efficient during the compilation and execution stage; otherwise, it will consume much time during compilation and execution.
4. **Debugging:** The errors that occur during compilation and execution of the program are called bugs, and tracing of an error once its existence has been confirmed is called debugging. The time spent in debugging may vary from 50% to 90% of total programming time. The debugging time is longer for a novice and unskilled programmer, whereas for an experienced programmer it may be less.

Some common errors which cause bugs are:

1. Errors in analysis

2. Errors in correct algorithm
3. Syntax errors
4. Data errors
5. Documentation errors
6. Operating system errors
7. Database connectivity errors
8. Missing supporting or linked files
9. Version support errors

These errors may cause an incomplete compilation and execution, termination, undesired output, or running in an infinite loop.

5. **Testing:** The program should be thoroughly tested before implementing it. It should give correct results and the desired output in all conditions. During testing all the executions must be tested at least once.

In Visual Basic the different levels of testing are:

1. Event level – Lowest level of testing
2. Form level – Testing of entire form
3. Module level – Testing of all linked forms of different modules
4. Project level – Testing of entire project
5. Field level – Testing of project with real-time data
6. Release level – Testing of package with setup
7. User level – Testing at the user end

12.3 PROGRAMMING TOOLS

Different tools used to generate program logic are:

1. **Algorithms:** A sequence of logic used to accomplish a particular task. An algorithm may be defined as an unambiguous procedure used to solve a problem.
2. **Flowcharts:** A graphical representation of an algorithm is called a flowchart.
3. **Pseudo-code:** A structural representation of an algorithm is called pseudo-code. It is used to represent the program logic which helps in understanding the process.
4. **Data-flow Diagrams (DFD):** The flow of data is represented by a DFD. Data-flow diagrams are made at different levels. In the first level there is a simple representation of data flow. In the second and third level the diagram is more illustrative, deeper, and complex.

Chapter 13

VISUAL BASIC 6.0

Visual Basic was first introduced in 1991. The introduction of Visual Basic created a new milestone in the field of Windows-based application development. It provides component-object model programming by which other components can be added in the application. The term 'Visual Basic' is made up of two words: Visual and Basic. By the 'Visual' method we create a Graphical User Interface (GUI). We can drag and drop the objects on the screen and can set their properties instead of writing lengthy codes. The 'Basic' part refers to the most popular programming language used: BASIC (Beginners All-purpose Symbolic Instruction Code). Visual Basic was generated from the original BASIC language. It is the easiest and fastest way to create an application. It provides a complete set of tools for rapid development of an application. It is an Integrated Development Environment where an application can be developed, tested, and run. We can create a web page or Internet Information Server (IIS) in Visual Basic. Many Windows applications use this language. Microsoft itself uses Visual Basic for the development of its applications.

13.1 HARDWARE AND SOFTWARE REQUIREMENTS FOR VISUAL BASIC

S.No.	Hardware/Software	Minimum Requirements	Recommended
1.	Processor	150 Mhz.	300 Mhz.
2.	RAM	16 MB	64 MB
3.	Hard Disk	1GB	4 GB
4.	Operating System	Win 95	Any higher version of Windows

13.1.1 Editions

There are three editions of Visual Basic:

1. **Visual Basic learning edition:** It includes all intrinsic controls, grid, tab and data bound controls. It is an introductory edition for beginners.
2. **Professional edition:** It includes all the features of the learning edition, ActiveX controls, and internet controls. This edition is for computer professionals.
3. **Enterprise edition:** It includes all the features of the professional edition, Automation Manager, Component Manager, Database Management Tools, and Microsoft Visual Source Safe. This edition is for commercial use.

13.2 APPLICATION TYPES

When we start Visual Basic, we will find different types of projects that can be developed.

1. **Standard EXE:** This is the most widely used application.
2. **ActiveX EXE:** It comes with the professional edition. ActiveX components are code-building components without an interface. It can be saved as an executable file and can be used in standard EXE projects.
3. **ActiveX DLL:** This is the same as ActiveX EXE but can be saved as DLL files (Dynamic Link Libraries).
4. **ActiveX control:** This project is used to develop custom controls. We can change the property and functionality of controls according to our needs and also use this control in standard EXE projects.
5. **ActiveX Document EXE:** This project is used in an environment where hyperlinking is supported. For example, Internet Explorer.
6. **ActiveX Document DLL:** The same as ActiveX Document EXE.

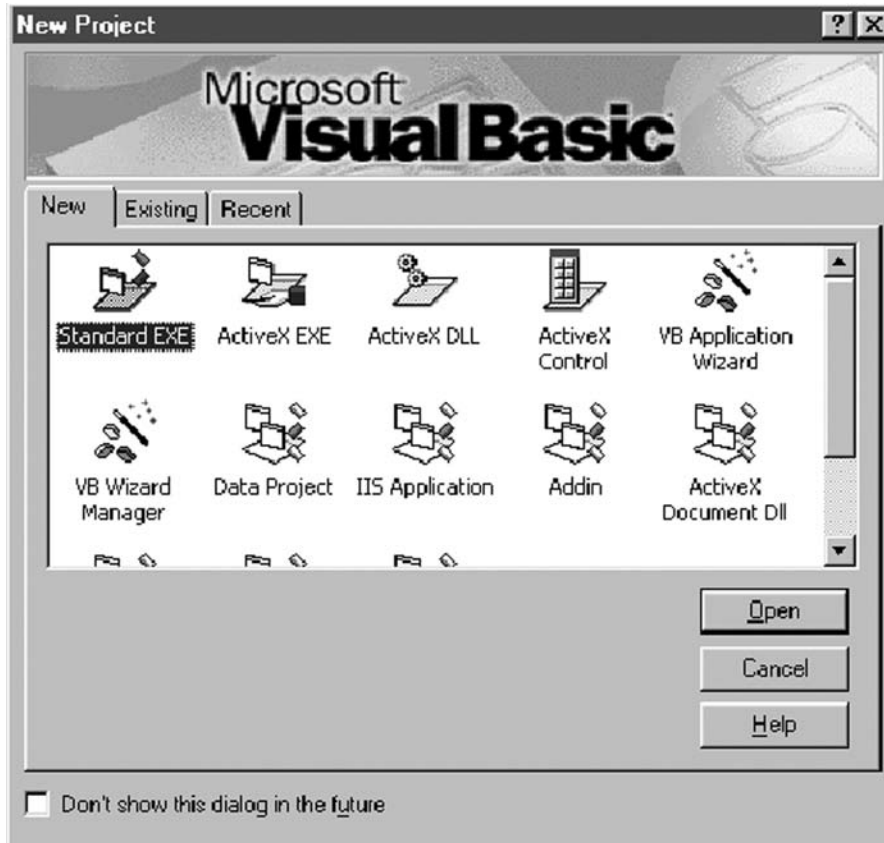


FIGURE 13.1

7. **VB Application Wizard:** This wizard helps to develop an application by selecting options and generating codes automatically (not used by professionals).
8. **VB Wizard Manager:** This is used to create a customized wizard. The wizard performs the operation according to the information given by the user.
9. **Data Project:** This is a project that creates a database application. It automatically adds the database connectivity control to the project. It also adds ActiveX Designers, such as Data Environment and Data Report to the project.
10. **DHTML Application:** This is used to develop dynamic HTML pages in Visual Basic.

11. **IIS Application:** The application that has to run on the web server is developed in this type of project. With the Internet Information Server, the application can be accessed by different users over the Internet.
12. **Addin:** The customized commands can be created by this project. These commands can be added in the Addin menu of the Standard EXE project.
13. **VB Enterprise Edition Control:** This is just like the Standard EXE Project without the options New, Existing, and Recent.

13.3 COMPILATION IN VISUAL BASIC

When a Visual Basic project is compiled, it generates an executable file. This executable file has pseudo-codes. When the executable file is run, the VBRUN30.DLL file converts the pseudo-code into machine-code. The pseudo-code is an intermediate code between Visual Basic code and machine-code. The microprocessor understands only machine-code. In the Professional and Enterprise editions of Visual Basic, the Visual Basic code can be directly compiled into machine-code.

13.3.1 Limitations

- The stack size is only 1 MB. The stack is the memory space where addresses of functions, procedures, and variables are stored.
- In a single form a maximum of 255 controls can be placed. If there is further need for extra controls you can make control arrays. All controls of one array are treated as a single control.
- The size of the module can be a maximum of 64 KB. If the code exceeds, add another module.
- The maximum number of lines in a form or module is 65534.
- The maximum number of characters in a line is 255.

13.4 VISUAL BASIC TERMINOLOGY

The terms used in Visual Basic are:

- **Form:** The screen where controls can be placed. No application is possible without including the form in the project.
- **Control:** Graphical objects, such as Label, Text Box, Command Button, etc.
- **Object:** This term refers to form, control, data report, etc.

- **Property:** The different attributes of an object. For example, Backcolor, Forecolor, Font, etc.
- **Method:** The action performed on an object.
- **Event:** Different events occur when an application is run.

13.5 INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Menu Bar

The Menu Bar shows the list of commands that can be used in the active window.

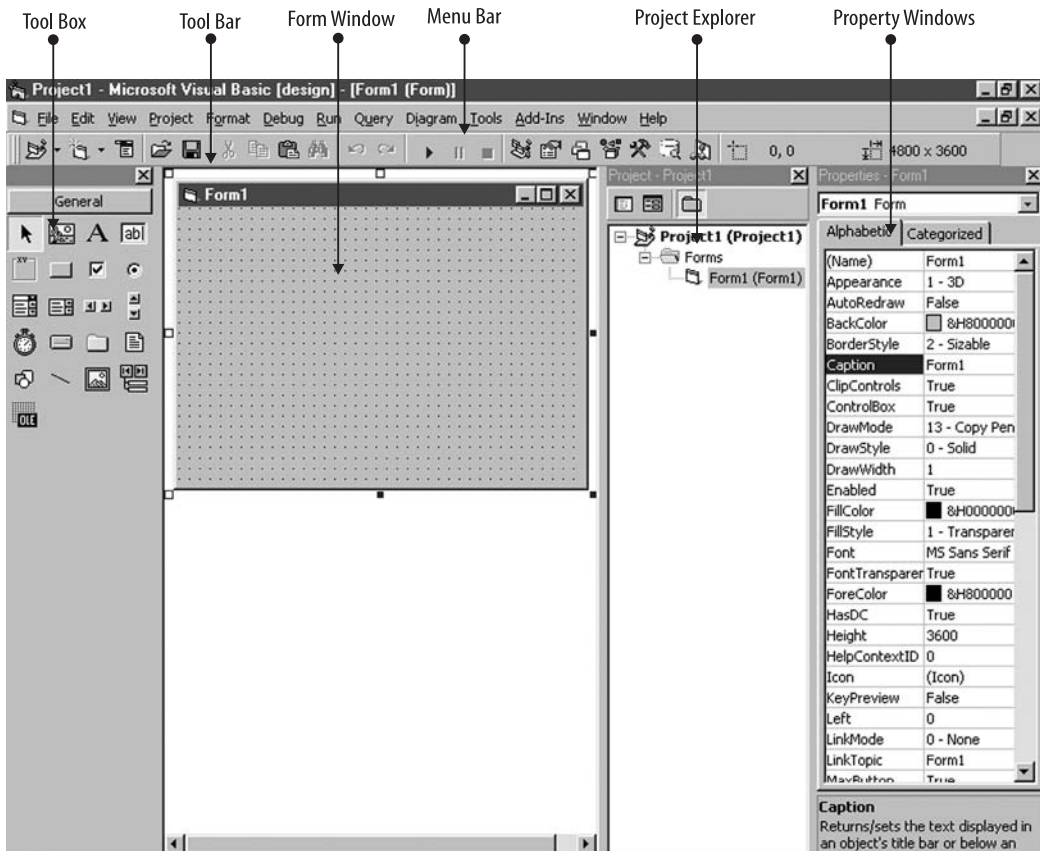


FIGURE 13.2

Tool Bar

We can directly access the commands without using the Menu Bar with the help of the Tool Bar. Some commonly used Tool Bars are:

- **Standard Tool Bar:** Gives the facility to Open a New Project, Add a New Form, Save Project, Cut, Copy, and Paste the objects and code both, and to run the project.
- **Edit Tool Bar:** Allows user to set and unset remarks and flags on the code line with some other facilities.
- **Debug Tool Bar:** Contains Run, Pause, and Break with some other options.
- **Form Editor Tool Bar:** Contains the commands of the Format Menu.
- **Form Window:** Allows user to place the controls on it to design the form.
- **Property Window:** Shows and sets the properties of different objects.

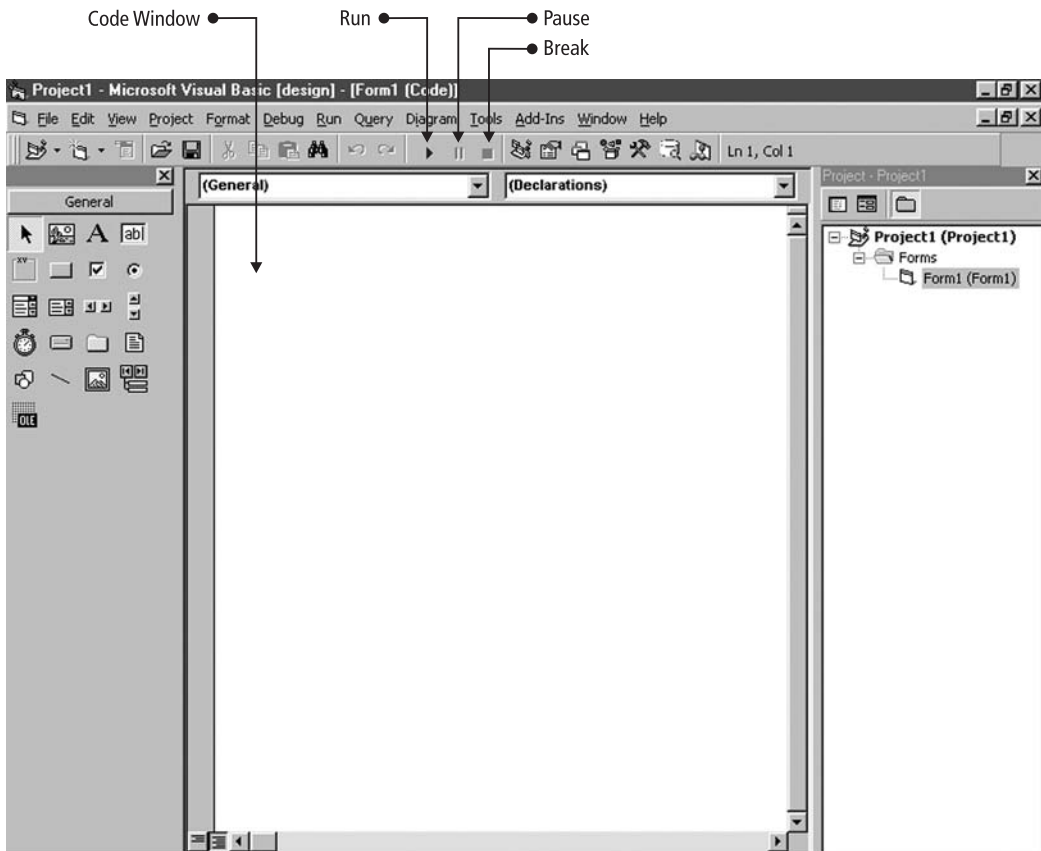


FIGURE 13.3

- **Tool Box:** Contains various controls that can be dragged on the form to design it. Some extra controls can be added in the Tool Box by right-clicking the Tool Box and selecting the component option.
- **Project Explorer:** Displays the components of the project, such as Forms, Module, and Data Report.
- **Code Window:** A Visual Basic code editor where a program is written.
- **Form Layout Window:** Allows user to set the position of the form in the application. The top-left corner of the desktop screen represents the coordinates – 0, 0.

Data Types

Data Type	Storage Size	Storage Type	Range
Byte	1 byte	Single and unsigned number	0-255
Boolean	2 bytes	Stores True/False	True or False
Integer	2 bytes	Stores integer	-32768 to 32767
Single	4 bytes	Floating-point numbers	For negative numbers: -3.402823E38 to -1.401298E-45 For positive numbers- 401298E-45 to 3.402823E38
Double	8 bytes	Floating-point numbers	For negative numbers: -1.79769313486232E308 to -4.94065645841247E-324 For positive numbers: 4.94065645841247E-324 to 1.79769313486232E308
Decimal	12 bytes	Decimal numbers	+/-79228162514264337593543 950335 to +/-7.9228162514264337593 543950335 (numbers with no decimal place to numbers with 28 decimal places)

Currency	8 bytes	Currency in number with 4 decimal places	- 922337203685477.5808 to 922337203685477.5807
String	10 bytes + String length	Stores String	1 to 2 billion characters
Date	8 bytes	Stores Date	1 Jan 100 to 31 Dec 9999
Variant	For numbers -16 bytes For String -22 bytes + String length	Can store any type of data	

Chapter 14

CONTROLS IN VISUAL BASIC

Visual Basic provides a number of intrinsic and extrinsic controls. Before starting the Visual Basic control let us take a look at the various properties of the form itself.

A form provides an interface for an application where the controls can be dragged to design the application. In a form's code window we can write the procedures for different events of controls used in the form.

Object Window

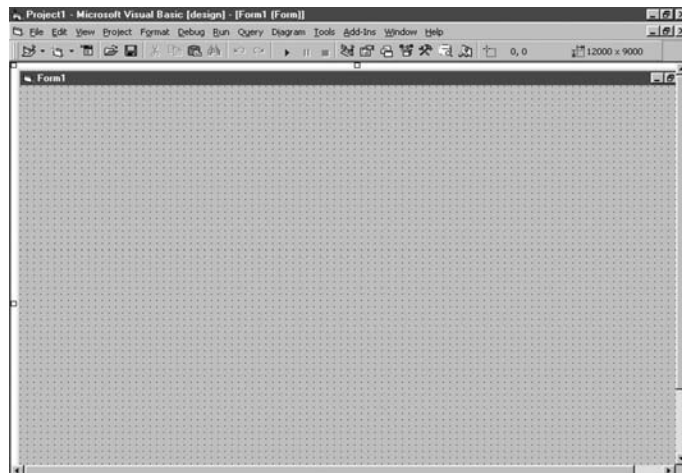


FIGURE 14.1

Code Window

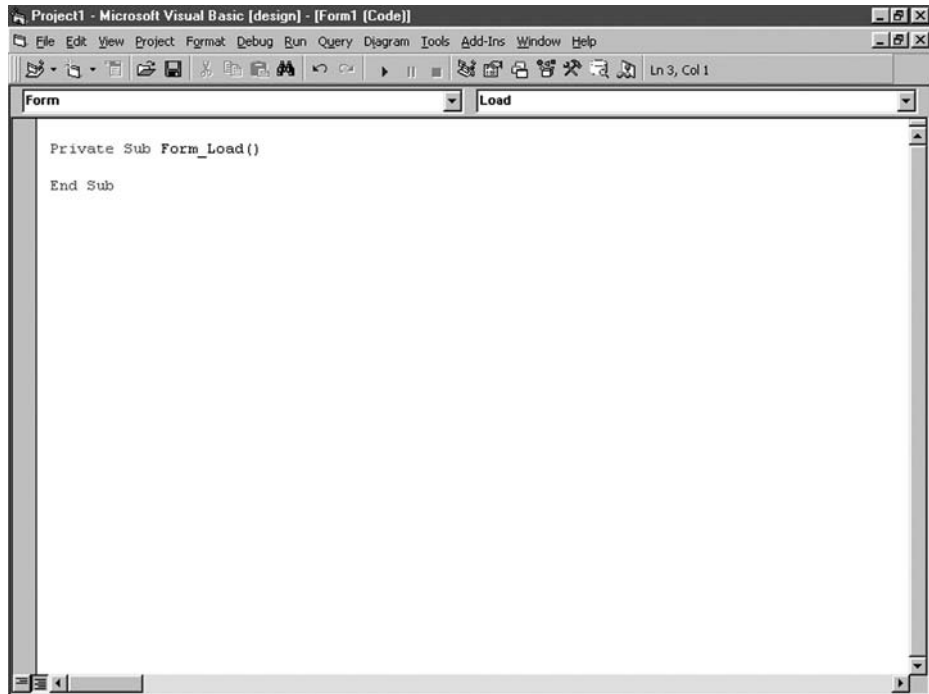


FIGURE 14.2

Form Properties

Property	Setting	Description
Name		Gives a name to the Form
Appearance	0-Flat	Form appears flat
	1-3D	Form appears three-dimensional
BackColor		Changes back color of the form
BorderStyle	0-None	Form appears without border and Control Box
	1-Fixed single	Form appears with border and only close button Form cannot be resized, maximized, or minimized

	2-Sizable	Form appears with border having maximize, minimize, and close buttons
Caption		Changes the caption of a Form
ControlBox	True	Form shows all controls, i.e., maximize, minimize, and close buttons
	False	Form does not show maximize, minimize, or close buttons, but shows top border
Enabled	True	Form becomes enabled
	False	Form becomes disabled
Font		Font of the form can be changed. Any control placed on the form will automatically set its font property as the form's font
Height		Measures height of the form in twips
Icon		Icon of the form can be changed by locating any icon file
Left		Distance of the form in twips from left of the screen
MaxButton	True	Shows maximize button in the form
	False	Does not show maximize button
MDIChild	True	Makes the form child of the MDI Form (Multiple Document Interface Form)
	False	Does not make the child of the MDI Form
MinButton	True	Minimize button becomes visible
	False	Minimize button becomes invisible
Moveable	True	Form can be moved using mouse after execution
	False	Form cannot be moved
Picture		A graphic can be displayed in the background of the Form by locating any picture file
ShowInTaskbar	True	Form can be shown in the Task Bar
	False	Form does not appear in the Task Bar

StartPosition	0- Manual	Form appears at the position given in Left and Top property
	1- Center owner	Form appears at the center of the screen
	2- Center screen	Form appears just below the previous form
Top		Distance in twips from the top of the screen
Visible	True	Form becomes visible
	False	Form becomes invisible
Width		Assigns the width of the form
Text		Shows text in the Text Box
WindowState	0- Normal	Normal-sized window appears
	1- Minimized	Form is minimized and form icon appears in the Task Bar
	2- Maximized	Form window is full screen

14.1 TOOL-BOX CONTROLS

Tool Box



FIGURE 14.3

Pointer

This is the first control in the Tool Box, mainly used to unselect the selected control.

Picture Box

Used to display images.

Property	Setting	Description
Name		Gives a name to the control
Alignment	0- None	Picture Box appears at user-defined position, i.e., Left and Top
	1- Align Top	Picture Box appears at the Top of the Form
	2- Align Bottom	Picture Box appears at the Bottom of the Form
	3- Align Left	Picture Box appears at the Left side of the Form
	4- Align Right	Picture Box appears at the Right side of the Form
Appearance	0-Flat	Picture Box becomes flat
	1-3 D	Picture Box becomes 3-D
AutoSize	True	Picture Box gets resized to the size of the graphic
	False	Picture Box does not resize
BackColor		Changes the back color of the control
BorderStyle	0- None	Control with no border
	1- Fixed Single	Control with fixed border
Height		Sets the height of the control
Left		Distance in twips from left side of the Form
Mouselcon		To locate the icon file if the Mouselcon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow type mouse icon appears

	2- Cross	Cross type mouse icon appears
	3- I-Beam	I Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
Picture		To locate and display the picture file
ToolTipText		Text to be shown when mouse pointer is over the control
Top		Distance in twips from top of the Form
Visible	True	Control becomes visible
	False	Control becomes invisible
Width		Sets the width of the control

Label

Property	Setting	Description
Alignment	0- Left Justify	Caption of the control is left aligned
	1- Right Justify	Caption of the control is right aligned
	2- Center	Caption of control is center aligned
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
AutoSize	True	Size of the control gets resized according to the length of the caption
	False	Size of the control is fixed
BackColor		Sets the back color of the control
BackStyle	0- Transparent	Back color of the control is transparent
	1- Opaque	Back color of the control is opaque and can be set to any color
BorderStyle	0- None	Control without border
	1- Fixed Single	Control with fixed border
Caption		Caption that is displayed with the control
Font		Sets the font of the control

ForeColor		Sets the fore color of the control
Height		Sets the height of the control
Left		Distance in twips from left side of the form
Mouselcon		To locate the icon file if the Mouselcon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow type mouse icon appears
	2- Cross	Cross type mouse icon appears
	3- I-Beam	I Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
ToolTipText		Text to be shown when mouse pointer is over the control
Top		Distance in twips from top of the Form
Visible	True	Control becomes visible

Text Box

Property	Setting	Description
Alignment	0- Left Justify	Text of the control is left aligned
	1- Right Justify	Text of the control is right aligned
	2- Center	Text of the control is center aligned
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
BackColor		Sets the back color of the control
BorderStyle	0- None	Control without border
	1- Fixed Single	Control with fixed border
Enabled	True	Text Box becomes enabled
	False	Text Box becomes disabled
Font		Sets the font of the control
ForeColor		Sets the fore color of the control

Height		Sets the height of the control
Left		Distance in twips from left side of the Form
Locked	True	No text can be entered during run-time
	False	Text can be entered during run-time
MaxLength		Maximum length of the text that can be entered
MultiLine	True	Text can be entered in more than one line
	False	Text can be entered only in one line
PasswordChar		If '*' is given it appears in Text Box in place of all the characters entered
Scroll Bars	0- None	No scroll bar appears (default setting)
	1- Horizontal	Horizontal scroll bar appears (If multiline property is True)
	2- Vertical	Vertical scroll bar appears (If multiline property is True)
	3- Both	Both scroll bars appear (If multiline property is True)
TabIndex		Sets the order of movement of the cursor when pressing Tab key (Index number starts from 0)
TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
Text		The text that appears in the Text Box
ToolTipText		Text to be shown when mouse pointer is over the control
Top		Distance in twips from the top of the Form
Visible	True	Control becomes visible

	False	Control becomes invisible
Width		Sets the width of the control

Frame

Property	Setting	Description
Name		Gives a name to the control
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
BackColor		Sets the back color of the control
BorderStyle	0- None	Control without a border
	1- Fixed Single	Control with a fixed border
Caption		Caption that is displayed with the control
Enabled	True	Control becomes enabled
	False	Control becomes disabled
Font		Sets the font of the control
ForeColor		Sets the fore color of the control
Height		Sets the height of the control
Index		Sets the control array
Left		Distance in twips from left side of the Form
Mouselcon		To locate the icon file if the Mouselcon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow type mouse icon appears
	2- Cross	Cross type mouse icon appears
	3- I-Beam	I-Beam type mouse icon appears
	99- Custom	Mouse icon can be customized

ToolTipText		Text to be shown when mouse pointer is over the control
Top		Distance in twips from the top of the Form
Visible	True	Control becomes visible
	False	Control becomes invisible
Width		Sets the width of the control

Command Button

Property	Setting	Description
Appearance	0-Flat	Control appears flat
	1-3 D	Control has 3-D appearance
BackColor		Sets the back color of the control
Cancel	True	Command Button can be executed by pressing Esc key
	False	Esc key does not work to execute Command Button
Caption		Caption that is displayed with the control
Default	True	Command Button can be executed by pressing the Enter key
	False	Enter key does not work to execute the Command Button
Enabled	True	Control becomes enabled
	False	Control becomes disabled
Font		Sets the font of the control
ForeColor		Sets the fore color of the control
Height		Sets the height of the control
Index		Sets the control array

Left		Distance in twips from left side of the Form
Mouselcon		To locate the icon file if the Mouselcon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow type mouse icon appears
	2- Cross	Cross type mouse icon appears
	3- I-Beam	I-Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
Picture		To locate and display the picture file
Style	0- Standard	Standard setting, does not show image
	1-Graphical	Image appears in the control if an image is set in the Picture property
TabIndex		Sets the order of movement of cursor on pressing Tab key (Index starts from 0)
TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
ToolTipText		Text to be shown when mouse pointer is over the control
Top		Distance in twips from top of the Form
Visible	True	Control becomes visible
	False	Control becomes invisible
Width		Sets the width of the control

Check Box

Property	Setting	Description
Alignment	0- Left Justify	Caption of the control is left aligned
	1- Right Justify	Caption of the control is right aligned

	2- Center	Caption of control is center aligned
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
Caption		Caption that is displayed with the control
DisabledPicture		Sets picture when the control is disabled (if Style property is set to Graphical)
DownPicture		Sets picture when the control is pressed
Enabled	True	Control becomes enabled
	False	Control becomes disabled
Font		Sets the font of the control
Forecolor		Sets the fore color of the control
Height		Sets the height of the control
Index		Sets the control array
Left		Distance in twips from left side of the Form
Mouselcon		To locate the icon file if the mouse icon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow type mouse icon appears
	2- Cross	Cross-type mouse icon appears
	3- I-Beam	I-Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
	Picture	To locate and display the picture file
Style	0- Standard	Standard setting, does not show image
	1-Graphical	Image appears in the control if an image is set in the Picture property
TabIndex		Sets the order of movement of the cursor upon pressing the Tab key (Index starts from 0)

TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
ToolTipText		Text to be shown when mouse pointer is over the control
Top		Distance in twips from top of the Form
Value	0- Unchecked	Check Box is unchecked
	1- Checked	Check Box is checked
	2- Grayed	Check Box is grayed
Visible	True	Control becomes visible
	False	Control becomes invisible
Width		Sets the width of the control

Option Button

Property	Setting	Description
Alignment	0- Left Justify	Caption of the control is left aligned
	1- Right Justify	Caption of the control is right aligned
	2- Center	Caption of control is center aligned
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
Caption		Caption that is displayed with the control
DisabledPicture		Sets picture when the control is disabled (if Style property is set to Graphical)
DownPicture		Sets picture when the control is pressed
Enabled	True	Control becomes enabled
	False	Control becomes disabled
Font		Sets the font of the control
Fore color		Sets the fore color of the control

Height		Sets the height of the control
Index		Sets the control array
Left		Distance in twips from left side of the Form
Mouselcon		To locate the icon file if the mouse icon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow-type mouse icon appears
	2- Cross	Cross-type mouse icon appears
	3- I-Beam	I-Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
Picture		To locate and display the picture file
Style	0- Standard	Standard setting, does not show image
	1-Graphical	Image appears in the control if an image is set in the Picture property
TabIndex		Sets the order of movement of the cursor upon pressing the Tab key (Index starts from 0)
TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
ToolTipText		Text to be shown when mouse pointer is over the control
Top		Distance in twips from top of the Form
Value	True	Option Button is selected
	False	Option Button is unselected
Visible	True	Control becomes visible
	False	Control becomes invisible
Width		Sets the width of the control

Combo Box

Property	Setting	Description
Name		Gives a name to the control
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
BackColor		Sets the back color of the control
Enabled	True	Control becomes enabled
	False	Control becomes disabled
Font		Sets the font of the control
ForeColor		Sets the fore color of the control
Height		Sets the height of the control
Left		Distance in twips from left side of the Form
List		Adds items in a Combo Box
Locked	True	View only, no item can be selected
Mouselcon		To locate the icon file if the mouse icon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow type mouse icon appears
	2- Cross	Cross type mouse icon appears
	3- I-Beam	I-Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
Sorted	True	Items in the Combo Box are displayed in alphabetical order
	False	Items in the Combo Box do not display in alphabetical order
Style	0- Dropdown Combo	Can write and view list, and select the item
	1- Simple Combo	No list view, selection by only Up and Down arrow key, can write

	2- Dropdown List	No write, selection by list view
TabIndex		Sets the order of movement of the cursor upon pressing the Tab key (Index starts from 0)
TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
Text		The text that appears in the Combo Box
Top		Distance in twips from top of the Form
Visible	True	Control becomes visible
	False	Control becomes invisible
Width		Sets the width of the control

List Box

Property	Setting	Description
Name		Gives a name to the control
Appearance	0- Flat	Control appears flat
	1- 3-D	Control has 3-D appearance
BackColor		Sets the back color of the control
Enabled	True	Control becomes enabled
	False	Control becomes disabled
Font		Sets the font of the control
ForeColor		Sets the fore color of the control
Height		Sets the height of the control
Left		Distance in twips from left side of the Form
List		Adds items in a List Box
MultiSelect	0- None	Only one item can be selected

	1- Simple	More than one item can be selected by mouse click
	2- Extended	A range of items can be selected by mouse dragging
Sorted	True	Items in the Combo Box are displayed in alphabetical order
	False	Items in the Combo Box do not display in alphabetical order
Style	0- Standard	Standard list appears
	1- Checkbox	List appears with check box

Horizontal Scroll Bar

Property	Setting	Description
Name		Gives a name to the control
Enabled	True	Control becomes enabled
	False	Control becomes disabled
LargeChange		Scale of change in Scroll Bar value when mouse is clicked on the Scroll Bar ribbon
Max	32767	Maximum value of the Scroll Bar
Min	0	Minimum value of the Scroll Bar
Mouselcon		To locate the icon file if the mouse icon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow type mouse icon appears
	2- Cross	Cross type mouse icon appears
	3- I-Beam	I-Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
Value		Current value of Scroll Bar pointer

Vertical Scroll Bar

Property	Setting	Description
Name		Gives a name to the control
Enabled	True	Control becomes enabled
	False	Control becomes disabled
LargeChange		Scale of change in Scroll Bar value when mouse is clicked on the Scroll Bar ribbon
Max	32767	Maximum value of the Scroll Bar
Min	0	Minimum value of the Scroll Bar
Mouselcon		To locate the icon file if the mouse icon property is set to 99-Custom
MousePointer	0- Default	Default mouse icon appears
	1- Arrow	Arrow-type mouse icon appears
	2- Cross	Cross-type mouse icon appears
	3- I-Beam	I-Beam type mouse icon appears
	99- Custom	Mouse icon can be customized
Value		Current value of the Scroll Bar pointer

Timer

Used to generate an event at a regular interval. The interval is defined at the interval property of the Timer.

Property	Setting	Description
Name		Gives a name to the control
Enabled	True	Timer gets enabled
	False	Timer gets disabled
Interval		(any time interval in milliseconds starting from 0)

Drive List Box

Displays current active drives on the system.

Property	Setting	Description
Name		Gives a name to the control
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
BackColor		Sets the back color of the control
Enabled	True	Control gets enabled
	False	Control gets disabled
Left		Distance in twips from left side of the Form
TabIndex		Sets the order of movement of cursor on pressing the Tab key (Index starts from 0)
TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
Visible	True	Control becomes visible
	False	Control becomes invisible

Dir List Box

Displays all folders of the currently selected drive.

Property	Setting	Description
Name		Gives a name to the control
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
BackColor		Sets the back color of the control
Enabled	True	Control gets enabled
	False	Control gets disabled

Left		Distance in twips from left side of the Form
TabIndex		Sets the order of movement of the cursor upon pressing the Tab key (Index starts from 0)
TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
Visible	True	Control becomes visible
	False	Control becomes invisible

File List Box

Display all files of the currently selected folder.

Property	Setting	Description
Name		Gives a name to the control
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
Archives	True	Displays archived attribute files
	False	Does not display archived attribute files
BackColor		Sets the back color of the control
Enabled	True	Control gets enabled
	False	Control gets disabled
Hidden	True	Shows hidden files
	False	Does not show hidden files
Left		Distance in twips from left side of the form
Normal	True	Displays Normal attribute files
	False	Does not display Normal attribute files

Pattern		Sets type of files to be displayed (* For all files)
ReadOnly	True	Displays read-only files
	False	Does not display read-only files
System	True	Displays system files
	False	Does not display system files
TabIndex		Sets the order of movement of the cursor upon pressing the Tab key (Index starts from 0)
TabStop	True	Cursor moves to that control if the Tab key is pressed
	False	Cursor does not move to that control if the Tab key is pressed
Visible	True	Control becomes visible
	False	Control becomes invisible

Shape

Draws different types of shapes, e.g., rectangle, circle, oval, square, etc.

Property	Setting	Description
BackColor		Sets the back color of the control
BackStyle	0- Transparent	Back color is transparent
	1- Opaque	Back color is opaque
BorderColor		Sets the border color of the control
BorderStyle	0- Transparent	Border is transparent
	1- Solid	Border is solid
	2- Dash	Border with dashes
	3- Dot	Border with dots
	4- Dash- Dot	Border with dashes and dots
	5- Dash- Dot-Dot	Border with dashes, dots, and dots
	6- Inside solid	Border is solid

BorderWidth	1	Sets width of the border (min. value is 1)
Shape	0- Rectangle	Shape is a rectangle
	1- Square	Shape is a square
	2- Oval	Shape is a oval
	3- Circle	Shape is a circle
	4- Rounded Rectangle	Shape is a rounded rectangle
	5- Rounded Square	Shape is a rounded square

Line

Draws a single line of a different style.

Property	Setting	Description
BorderColor		Sets the border color of the control
BorderStyle	0- Transparent	Border is transparent
	1- Solid	Border is solid
	2- Dash	Border with dashes
	3- Dot	Border with dots
	4- Dash- Dot	Border with dashes and dots
	5- Dash- Dot-Dot	Border with dashes, dots, and dots
	6- InsideSolid	Border is solid
BorderWidth	1	Sets width of the border (min. value is 1)
Visible	True	Control becomes visible
	False	Control becomes invisible
X1		Position of start point of the line from left of the Form

X2		Position of end point of line from left of the Form
Y1		Position of start point of line from top of the Form
Y2		Position of end point of line from top of the Form

Image

Displays graphics of JPEG, GIF, Bitmap, and icon files. Graphics can be resized to fit in the Image Box.

Property	Setting	Description
Appearance	0- Flat	Control appears flat
	1- 3 D	Control has 3-D appearance
BorderStyle	0- None	Control without border
	1- Fixed Single	Control with fixed border
Picture		Sets picture from the image file
Stretch	True	Image is resized to fit in the Picture Box
	False	Image displays with its original size
Visible	True	Control becomes visible
	False	Control becomes invisible

Form Events

- **Activate:** When the Form gets focused, the Activate event occurs.
- **Deactivate:** When the Form loses focus, the Deactivate event occurs.
- **Initialize:** When the Form is created, the Initialize event occurs.
- **Load:** When the application is run and the Form is loaded in the memory, the Load event occurs.
- **QueryUnload:** This event occurs just before the Form is unloaded.
- **Terminate:** This event occurs after the Unload event.

Control Events

- **Click:** When an object is clicked.
- **DblClick:** When an object is double-clicked.
- **GotFocus:** When an object receives focus.
- **LostFocus:** When an object loses focus.
- **Change:** This event occurs when a user enters or changes the text in the Text Box or Combo Box.
- **DropDown:** When a list of the Combo Box begins to drop down, this event occurs.
- **Scroll:** When the scroll bar of a control is scrolled, this event occurs.
- **ItemCheck:** When any item of the List Box is checked or unchecked, this event occurs. This event is generated only when its Style property is set to 1-Checkbox.
- **Timer:** The event occurs each time after the time specified in milliseconds in the Interval property of the Timer control.
- **KeyPress:** The event occurs when any key is pressed on the keyboard. This event works with Text Box and Combo Box.
- **MouseOver:** When the mouse pointer comes over an object the MouseOver event occurs.

Chapter 15

VARIABLES AND OPERATORS IN VISUAL BASIC

15.1 VARIABLE NAMING CONVENTIONS

1. Must begin with a character.
2. Maximum length of variable is 255 characters.
3. Should not contain spaces or special characters except the underscore (_).

15.2 VARIABLE DECLARATION

To declare the variable private use DIM

```
DIM < variable name > as < Data Type >
```

For example, DIM productname as String

```
DIM quantity as Integer
```

To declare the variable public use PUBLIC

```
PUBLIC < variable name > as < Data Type >
```

For example, `PUBLIC productname as String`
`PUBLIC quantity as Integer`

15.3 SCOPE OF VARIABLES

Type of Declaration	Section of Declaration	Scope
DIM	Procedure	Can be accessed by that particular procedure only where the variable has been declared.
DIM	General declaration section of the form	Can be accessed by all the procedures of the form and the value can also be transferred from one procedure to another.
DIM	General declaration section of module	Can be accessed by all the procedures and the value can be used in that procedure only where it has been assigned to the variable. The value of the variable cannot be transferred to another procedure.
PUBLIC	General declaration section of module	The variable and value both can be accessed in any procedure of all the forms.

NOTE Duplicate declaration of the variable is not allowed in the current scope.

15.4 LOGICAL OPERATORS

Operator	Function of Operator	Example	Return Value
+	Adds two numbers	A=10+10	20
-	Subtracts right number from left number	A=20-10	10
*	Multiplies two numbers	A=20*10	200
/	Divides left number by right number	A=20/6	3.33
\	Divides left number by right number and returns only integer number	A=20\6	3

MOD	Divides two numbers and returns remainder	A=20 mod 6	2
^	Used to express the power of an exponent	A=10^10	100
&	Used to concatenate two strings	Myname="Mahesh" & "Gupta"	"Mahesh Gupta"

15.5 LOGICAL OPERATORS

1. AND

Checks both the expressions and returns True if both the expressions are true. If one of the expressions is False or Null, it returns False.

1 st Expression	2 nd Expression	Result
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
TRUE	NULL	NULL
FALSE	NULL	FALSE
NULL	FALSE	FALSE
NULL	TRUE	FALSE
NULL	NULL	NULL

2. EQV

Checks whether two expressions are identical or not. If both expressions are identical, returns True, otherwise returns False. If any expression is Null, the return value is also Null.

1 st Expression	2 nd Expression	Result
TRUE	TRUE	TRUE
FALSE	FALSE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE

3. NOT

Used to reverse the expression.

Expression	Result
TRUE	FALSE
FALSE	TRUE
NULL	NULL

4. OR

If either of the two expressions is True, the result is True.

1 st Expression	2 nd Expression	Result
TRUE	TRUE	TRUE
FALSE	FALSE	FALSE
NULL	NULL	NULL
TRUE	FALSE	TRUE
TRUE	NULL	TRUE
FALSE	TRUE	TRUE
FALSE	NULL	NULL
NULL	TRUE	NULL
NULL	FALSE	NULL

5. XOR

If only one expression is True, the result is True.

1 st Expression	2 nd Expression	Result
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

15.6 IF-ELSE STATEMENT

Case 1

```
If    <Condition>  Then
    <Statement 1>
    <Statement 2>
Else
    <Statement 3>
End If
```

Checks the condition, if True, it executes Statement 1 and Statement 2; otherwise, Statement 3 is executed.

Case 2

```
If      <Condition 1> Then
    <Statement 1>
ElseIf  <Condition 2> then
    <Statement 2>
End If
```

All conditions are evaluated sequentially. If True, the corresponding statement is executed.

15.7 DO WHILE STATEMENT

Case 1

```
Do while <Condition>
    <Statement>
Loop
```

Performs the Do While loop and the Statement is executed while the condition is True.

Case 2

```
Do while Not <Condition>
    <Statement>
Loop
```

Performs the loop until the condition becomes True.

You can use **Exit Do** to terminate the Do While Loop.

```
Do while <Condition>
    <Statement>
Exit Do
Loop
```

15.8 FOR LOOP

```
For    <Counter>=<Start> To <End>
      <Statement>
Next
```

The Counter is incremented by 1 each time the loop is performed. The loop is performed until the value of the Counter becomes equal to the < End > value. You can exit the For Loop by using the **Exit For** statement.

The Counter is incremented by 1 by default. You can increment it by any number by specifying **Step < no.>**

Look at the example given.

```
For N=1 to 100 Step 2
<Statement>
Next
```

In this For statement, Counter N will be incremented by two until the value of N reaches 100, which means the loop will be performed 50 times. You can also decrement the Counter by using a negative number of steps. Consider the above example to decrement the Counter.

```
For N=1 to 100 Step -2
<Statement>
Next
```

15.9 WITH-END WITH STATEMENT

Executes all the statements within the **With-End With** Block on a single object.

```
with  <Object Name>
      <Statement>
End with
```

Example:

```
With Text1
.Text = "Ishita"
.BackColor = vbRed
.ForeColor = vbBlack
End with
```

Chapter 16

FUNCTIONS IN VISUAL BASIC

Functions may be inbuilt or user-defined. We will discuss here only inbuilt Visual Basic functions.

1. ASC ()

Returns an ASCII value of the first character in the string.

Example:

Expression	Return Value
Result = Asc("M")	77
Result = Asc("Mahesh")	77
Result = Asc("m")	109

2. CBool ()

Converts its arguments to a Boolean function and returns False for a zero value and True for all other values.

Example:

Expression	Return Value
Result = CBool (0)	False
Result = CBool (1)	True
Result = CBool (2)	True

3. Cbyte ()

Converts an argument to a byte.

4. Ccur ()

Converts an argument to a currency data type.

Example:

```
Dim No as double
Dim Result as currency
No = 1000.123456
Result = CCur(No) 'Return Value is 1000.1234 (Currency Type)
```

5. CDate ()

Converts a string to a date type.

6. Cdec ()

Converts an argument to a decimal type.

7. Cdbl ()

Converts an argument to a double type.

8. Chr ()

Converts an ASCII value to its character representation.

Example:

Expression	Return Value
Result = Chr(77)	M
Result = Chr(32)	< Space Character >
Result = Chr(122)	z

9. CInt ()

Converts a character to an integer.

Example:

Expression	Return Value
Result = CInt("2")	2
Result = CInt("2.84")	3
Result = CInt("2.13")	2

The fractional value is rounded off.

10. CLng ()

Converts a character to a long.

11. CSng ()

Converts an argument to a single.

12. CStr ()

Converts an argument to a string.

13. CVar ()

Converts an argument to a variant.

14. Date ()

Returns current system date.

Example 1:

Print Date 'current system date will be displayed on the form

Example 2:

Dim D as date

D = date 'D will return current system date

15. DateAdd ()

Adds Day, Month, or Year in the given date and returns a new date.

Example:

Expression	Return Value
Result = DateAdd ("m", 2, Date)	Adds two months to current system date
Result = DateAdd ("yyy", 1, Date)	Adds one year to current system date
Result = DateAdd ("D", 10, Date)	Adds 10 days to current system date
Result = DateAdd ("Ww", 3, Date)	Adds 3 weeks to current system date
Result = DateAdd ("q", 2, Date)	Adds 2 Quarters to current system date

16. DateDiff ()

DateDiff (Interval, Date1, Date2)

Example:

DateDiff ("D", "01-01-05", "11-01-05") ' Returns 10

Use the following intervals:

m – Month
 YYYY – Year
 q – Quarter
 Ww – Week

17. DatePart ()

Returns a specified part of a given date.

DatePart (Interval, Date)

Use the following intervals:

Symbol	Description
Yyyy	To extract Year from date
Y	To extract Day of year from date
Q	To extract Quarter from date
M	To extract Month from date
D	To extract Day from date
W	To extract Weekday from date
Ww	To extract Week from date

18. Day ()

Returns the Day number from the date.

`Result = Day (Date)` 'Returns current day from system date

19. Format ()

Changes the output of an expression according to the format given.

Expression	Return Value
<code>Result = Format (Date,"dd-MM-yy")</code>	01-02-05 (If current Date is Feb. 1, 2005)
<code>Result = Format (Date,"dd-MMM-yy")</code>	01-Feb-05
<code>Result = Format (Date,"dd-MMM-yyyy")</code>	01-Feb-2005
<code>Result = Format (Time, "hh:mm:ss")</code>	12:05:15 (If current Time is 12 Hrs. 5 Min and 15 Sec. PM)
<code>Result = Format (Time,"h:m:s")</code>	12:5:15
<code>Result = Format(Time,"h:m:s AM PM")</code>	12:5:15 PM
<code>Result = Format (1928, "##.00")</code>	1928.00
<code>Result = Format (192.8, "##.00")</code>	192.80
<code>Result = Format (1928, "##,###.00")</code>	1,928.00
<code>Result = Format ("INDIA", "<")</code>	india
<code>Result = Format ("India", ">")</code>	INDIA

20. Hour ()

Extracts hour from time.

`Result = Hour (Time)`

21. IIf ()

Checks the condition, if found true, the first expression is returned; otherwise, the second expression is returned.

`Result = IIf (Marks > 33, "Pass", "Fail")`

22. Instr ()

Returns the position of the first string within the second.

```
Result = Instr (Startpos, String 1, String 2)
Startpos: Start position in String 1 from where String 2
          has to be searched.
String 1: String in which String 2 has to be searched.
String 2: String to be searched.
```

23. Isarray ()

Returns True if a variable is an array; otherwise, returns False.

```
Result = Isarray (Variable)
```

24. InputBox ()

Shows a dialogue box with a Text Box, an **OK** button, and a **Cancel** button. The Input given by a user is returned as a string: InputBox (Prompt, Title, Default, Xpos, Ypos).

Prompt : The message to be displayed in the dialogue box. It can have a maximum of 1024 characters and can be displayed in multi-lines in the dialogue box by using chr (13) or chr (10).

Title : The caption which is to be displayed in the title bar of the Text Box.

Default : The default string that is displayed in the Text Box.

Xpos : Position of the Input Box from the left side of the screen.

Ypos : Position of the Input Box from the top of the screen. If the X and Y position is not specified, the InputBox appears at the center of the screen. Title, Default, Xpos, and Ypos are optional.

25. IsDate ()

Returns True if the argument is date type; otherwise, returns False.

```
IsDate (Variable name)
```

26. IsEmpty ()

Returns True if a variable is not initialized.

```
IsEmpty (Variable name)
```

27. IsNull ()

Returns True if the expression has no data; otherwise, returns False.

28. IsNumeric ()

Returns True if an expression is a valid number.

29. Ltrim ()

Returns a string with no leading spaces.

```
Result = Ltrim (" world") 'Returns "world"
```

30. Rtrim ()

Returns a string with no trailing spaces.

```
Result = Rtrim ("world ") 'Returns "world"
```

31. Trim ()

Returns a string without leading and trailing spaces.

```
Result = Trim (" world ") 'Returns "world"
```

32. MsgBox ()

Displays a message box with command buttons and waits for a response from the user.

```
MsgBox (Prompt, Buttons, Title)
```

Prompt : The message to be displayed in the dialogue box. The message can be a maximum of 1024 characters. The message can be displayed in more than one line by using the chr(13) or chr(10) characters.

Button : The button that is displayed in the dialogue box. It may be the **OK** Button, **Cancel** Button, **Retry** Button, **Ignore** Button, etc. If no button is specified, the **OK** Button is displayed by default.

We can use any combination of the buttons or icons from the following list.

Syntax	Description
VbOKOnly	OK Button is displayed
VbOKCancel	Two Buttons, OK and Cancel, are displayed
VbYesNO	Yes and No Buttons are displayed
VbYesNoCancel	Yes, No, and Cancel Buttons are displayed
VbRetryCancel	Retry and Cancel Buttons are displayed
VbAbortRetryIgnore	Abort, Retry, and Ignore Buttons are displayed

VbMsgBoxHelpButton	Help Button is displayed
VbCritical	Displays Critical message icon
VbExclamation	Displays an icon with exclamation mark
VbQuestion	Displays a question mark icon
VbInformation	Displays an information icon
VbDefaultButton1	First Button is default
VbDefaultButton2	Second Button is default
VbDefaultButton3	Third Button is default
VbDefaultButton4	Fourth Button is default
VbSystemModal	All applications are terminated until user responds
VbMsgBoxRight	Displays Text with right alignment

Return Value of Command Buttons

Button Type	Return Value
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

You can use a combination of buttons and icons by using the '+' character.

Example:

```
MsgBox ("Do you want to print", VbYesNo + VbQuestion, "Print")
```

33. Month ()

Returns the month from the given date.

```
Result = Month (date)
```

34. Now ()

Returns the current date and time.

```
Result = Now
```

35. Second ()

Returns the second from date.

```
Result = Second (date)
```

36. Sqr ()

Returns the square root of a number.

37. String ()

Returns a string of the given characters and length specified.

```
Result = String (10, "-") 'Returns "-----"
```

38. Str ()

Converts a number to a string.

```
Result = Str (123.10) 'Returns "123.10"
```

39. Time ()

Returns current system time.

40. Val ()

Converts a string to a number.

```
Result = Val ("123") 'Returns "123"
Result = Val ("1234Hello") 'Returns "1234"
```

41. Weekday ()

Returns number of the weekday from a date.

```
Result = Weekday (date)
```

42. Year ()

Returns year from a date.

```
Result = year ( #1-Jan-2005#) 'Returns "2005"
```


43. MonthName ()

Converts month number into month name.

```
Result = MonthName (month (#01-05-2005#)) 'Returns "May"
```

44. Cos ()

Returns cosine of a given angle.

45. Fix ()

Returns the integer part of a number.

46. Lcase ()

Converts a string of uppercase to lowercase.

```
Result = Lcase ("WORLD") 'Returns "world"
```

47. Ucase ()

Converts a string of lowercase to uppercase.

```
Result = Ucase ("world") 'Returns "WORLD"
```

48. Left ()

Returns specified number of characters from left side of a string.

```
Result = Left ("world", 4) 'Returns "worl"
```

49. Right ()

Returns specified number of characters from right of a string.

```
Result = Right ("world", 4) 'Returns "orld"
```

50. Len ()

Returns length of a string.

```
Result = Len ("world") 'Returns "5"
```

51. Log ()

Returns natural logarithm of a given number.

52. Minute ()

Returns 0-59 representing the minute part of a time.

```
Result = minute (Time)
```

53. Replace ()

Replaces a character in a string with another character specified.

```
Result = Replace ("LAN", "L", "M") 'Returns "MAN"
```

54. Round ()

Rounds off the decimal part and returns an integer value.

```
Result = Round (19.51) 'Returns 20
```

```
Result = Round (19.50) 'Returns 20
```

```
Result = Round (19.49) 'Returns 19
```

55. Sgn ()

Returns a number indicating the sign of a given number.

```
Result = Sgn (100) 'Returns 1
```

```
Result = Sgn (-100) 'Returns -1
```

```
Result = Sgn (0) 'Returns 0
```

56. Space ()

Returns a string containing a specified number of spaces.

```
Result = Space (5) 'Returns a string with 5 spaces
```

57. Sin ()

Returns sine of a given angle.


58. Tan ()

Returns tangent of a given angle.

59. weekdayName ()

Returns the weekday name corresponding to a given value.

```
Result = weekdayName (2) 'Returns "Monday"
```

NOTE  Sunday is the first weekday.

Chapter 17

INTRODUCTION TO DATABASES

A **database** is an organized collection of data where each data is related to each other. The data contains useful information in the form of text, numbers, and dates. You can retrieve any information stored in a database at any time to make your decision quickly.

A Database Management System (DBMS) is the management of the data, i.e., storing, processing, and retrieving of data. The Relational Database Management System (RDBMS) is the management of all related data.

Some commonly used RDBMSs are:

S. No.	RDBMS	Company Name
1.	Oracle 8/8i/9i/11i	ORACLE Corporation
2.	Microsoft SQL Server 7.0/2000	Microsoft Corporation
3.	Sybase SQL Server	Sybase Incorporation
4.	Informix Server	Informix Software Incorporation

The database is made up of several tables.

17.1 TABLES

A table is a group of data having similar information. All the data belong to the same group. These data are further subdivided into several columns named Fields. Each field may have different data types and sizes but they are related to each other. The size of the field indicates the maximum amount of data that can be entered into that field.

Let's use an example of a school. Suppose a school has two departments:

1. Admission
2. Library

The **Admission** department does the admission of new students and collects the following information about the student:

Name

Father's Name

Date of Birth

Address

Monthly income

Name, Father's Name, and Address are stored in an alphanumeric format. Date of Birth is stored in a date type format and monthly income is stored in numeric format.

The **Library** department purchases new books and maintains the record of the books. The book's information is stored in the following columns:

Book Name

Subject

Author

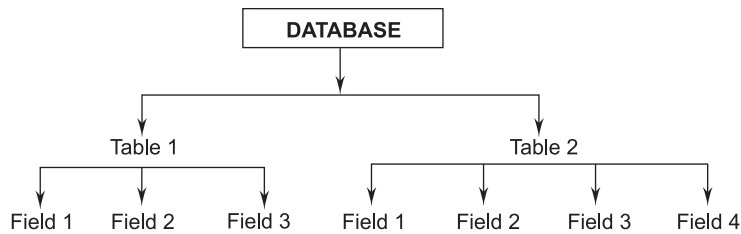
Publication

Date of purchase

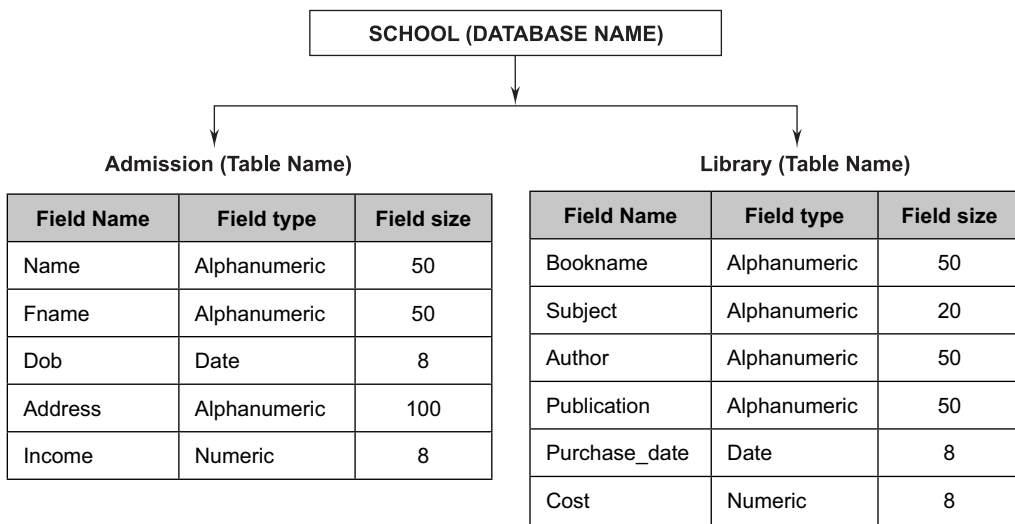
Book Name, Subject, Author, and Publication, are stored in alphanumeric format, date of purchase is stored in date type format, and book cost is stored in numeric format.

Now we create a database for the school.

17.2 STRUCTURE OF A DATABASE



A Field may be alphanumeric, numeric, or a date type.



17.3 KEYS

The main role of keys is to maintain data integrity in a database or table. Keys are one of the following types:

1. **Candidate Key:** The column that has a unique value in all the rows is a candidate key. The candidate key may be more than one in a table.
2. **Primary Key:** If any column is made a primary key, the data entered in that column must be unique across the entire column. The primary key column cannot be left blank. The primary key is one of the candidate keys.
3. **Alternate Key:** All candidate keys which are not primary keys are referred to as alternate keys.

4. **Composite Key:** If there is more than one field used as primary keys, the sets of these keys are called composite keys.
5. **Foreign Key:** When a primary key of one table also exists in another table, it is referred as a foreign key for the second table. The foreign key is used for making a relation between both the tables.

17.4 DATA INTEGRITY

For a successful database operation there must be data integrity. There are three types of data integrity.

1. **Entity Integrity:** Ensures that each record is unique and can be identified by a primary key, because a primary key field contains a unique value.
2. **Domain Integrity:** Ensures column level validation, i.e., each value of the field is validated before entering, whether it is correct or not, and only the correct value is allowed to be entered.
3. **Referential Integrity:** Ensures that for each value of a foreign key, a primary key exists in the master table.

Chapter 18

MS ACCESS 2000

18.1 CREATING A DATABASE IN MS ACCESS 2000

When starting MS Access a dialogue box with three option buttons appears.

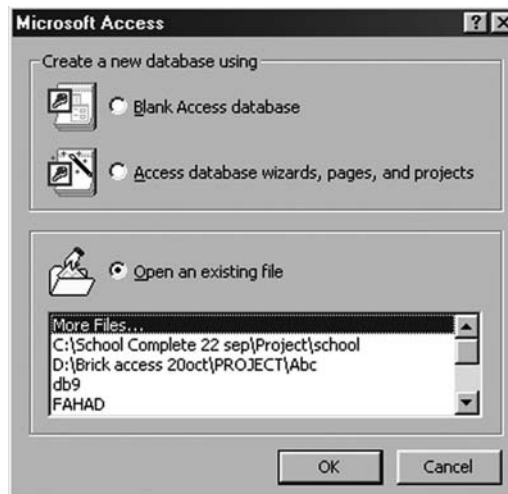


FIGURE 18.1

- **Blank Access Database** is used to create a new database by defining field name, data type, and field size.
- **Access database wizard** is used to create a database using a wizard.
- **Open an existing file** is used to open an already created database.

18.1.1 Access Database Wizard

To create the database with a wizard, first select the Database wizard option, then click the **OK** button. You will see the screen given below:

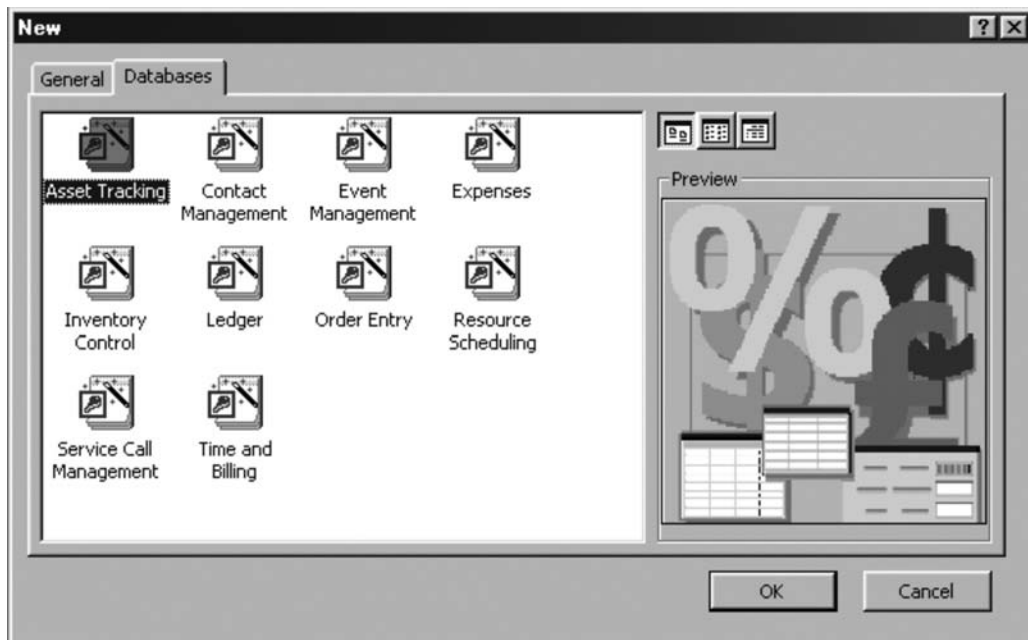


FIGURE 18.2

Select any type of database and click the **OK** button. You will see a screen called File New Database. Enter the filename in the box and click the **Create** button. Now follow the instructions and options given and click the **Next** button and lastly click the **Finish** button.

After gathering the information MS Access creates a database with the extension .mdb. The database contains different tables, forms, queries, and reports.



FIGURE 18.3

18.1.2 Blank Access Database

This option is used to create a customized database. To create a new database, select this option and click the **OK** button. On the next screen you will see a File New Database window. Give any valid filename in the File Name box and click the **Create** button to save the database file with the .mdb extension. You can select any path to save your database file by selecting the Save in box (see Figure 7.3).

The next window that appears is given below:

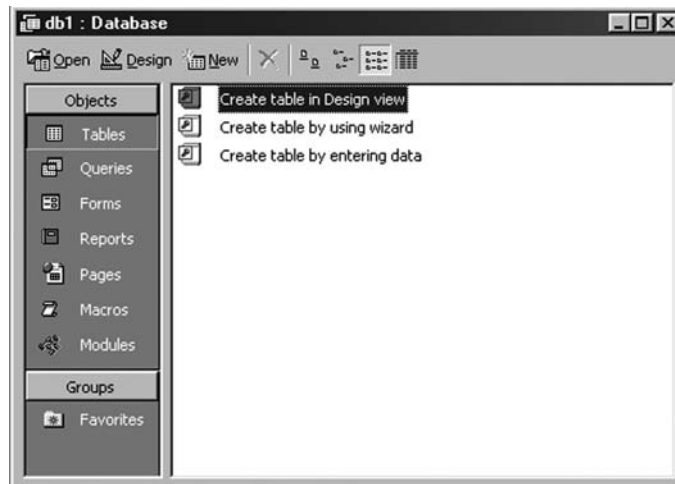


FIGURE 18.4

On the left side of the window, seven types of objects are given. Select the first object, i.e., Table (by default it is already selected) and click the **New** button. The options that come with the New Table dialogue box are (see Figure 7.4):

- **Datasheet View:** You can create a table by entering the data. Twenty fields are given in the table from field1 to field20. The data type of all the fields are by default text, but once you save the data, the data type of the fields are changed according to the data entered. You can rename the field by double-clicking the field name.
- **Table Wizard:** Creates a table using a wizard.
- **Design View:** A customized table can be created using Design View.
- **Import Table:** A table can be imported from another database using this option.
- **Link Table:** The table can be linked with another database and you can work with these options.



FIGURE 18.5

Select the Create table in the Design View option. As you select this option, on the tool bar the **Open** and **Design** buttons get enabled. Now click the **Open** button on the tool bar that gives a window to design the table.

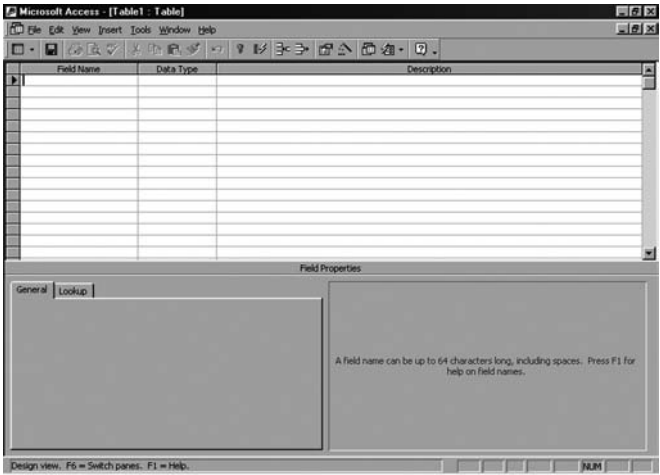


FIGURE 18.6

Give a field name in the Field Name column and select a data type from the Data Type column. A table with the following fields is shown in Figure 18.7.

Field Name	Data Type	Field Size
Name	Text	50
Age	Number	3
Date_of_birth	Date	8
Monthly_income	Currency	8

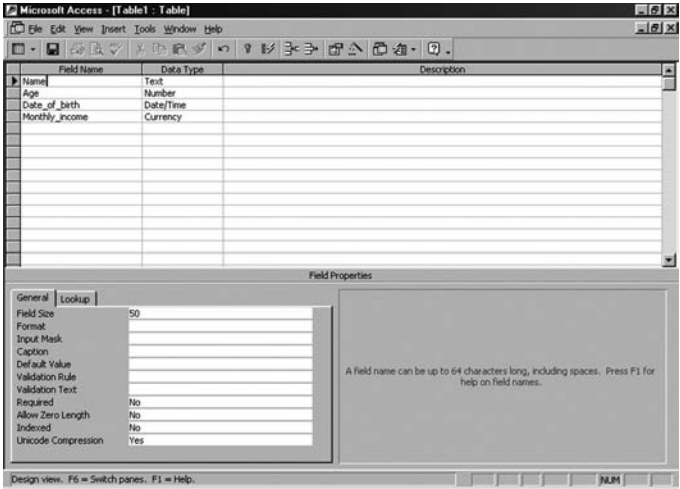


FIGURE 18.7

You can insert a new field between two existing fields. To do this select a field above which you want to insert a new field and then right-click. Now select the Insert Rows option. A field can also be deleted. To delete select the row you want to delete and right-click and select the Delete Rows option.

18.2 DATA TYPES

1. **Autonumber:** The data you enter in the table is automatically given a sequence of numbers starting from one.
2. **Text:** Stores data up to 255 characters. Data may contain text, numbers, special characters, and spaces.
3. **Memo:** Stores large amounts of text type data.
4. **Number:** Stores numeric data.
5. **Currency:** Stores money-type data with four decimal places.
6. **Date/Time:** Stores date, time, or both.
7. **Yes/No:** Stores boolean values, i.e., Yes or No, True or False.
8. **OLEObject:** Stores OLE (Object Linking and Embedding) objects, i.e., pictures, sound, or any other type of data.
9. **Lookup Wizard:** A field can be created from which a user can select a value.

18.3 FIELD PROPERTIES

1. Field Size

For Text and Memo Data Types

This property is used to set the field size of the text field. The text field can contain a maximum of 255 characters. The field size of a text field is 50 by default, but it can be changed according to the amount of data that this field will store. To save disk space do not give the size of the field more than the amount of data to be inserted.

Memo field can store 1.2 GB of data. It does not have the field size property.

For Number Data Types

1. **Byte:** Stores numbers from 0 to 255 without decimal places.
2. **Integer:** Stores numbers from -32000 to +32000 without decimal places.

3. **Long Integer:** Stores numbers from -2 billion to +2 billion without decimal places.
4. **Single:** Stores numbers from -3.4×10^{38} to $+3.4 \times 10^{38}$ with 6 decimal places.
5. **Double:** Stores numbers from -1.79×10^{308} to $+1.79 \times 10^{308}$ with 10 decimal places.

For Currency Data Types

Field size is fixed for the currency data type. It can store 15 digits to the left of the decimal point and 4 digits to the right of the decimal point.

2. Format

For Text and Memo Data Types

To set the format of the text field, you can use this property.

Symbol	Description
@	If Character is required
&	If Character is not required
>	Characters are converted into uppercase
<	Characters are converted into lowercase

For Number, Autonumber, and Currency Field

1. **General Number:** The number is displayed in the same format as it is entered.
2. **Currency:** The number is displayed in currency format, i.e., with a comma after 3 digits and with two decimal places. A dollar sign is displayed before the first digit.
3. **Euro:** The same as currency type but a Euro sign is displayed instead of a dollar sign.
4. **Fixed:** One digit is displayed before the decimal place and 2 digits are displayed after the decimal place.
5. **Standard:** The same as the currency data type but without a currency sign.
6. **Percent:** Numbers are displayed in % form, i.e., 1 as 100%, 2 as 200%, and .2 as 20%.
7. **Scientific:** Numbers are displayed in their scientific notations, i.e., 100 as 1.00E+02 and 16900 as 1.69E+04.

For Date Field

1. **General Date:** If a date is entered, it displays the date and if a time is entered, it displays time. But if both are entered it displays both.
2. **Long Date:** The date is displayed in the format weekday, month name, day, year.
Example: Monday, January 1, 2005.
3. **Medium Date:** The date is displayed in the format Month/Day/Year.
Example: 1/1/2005.
4. **Long Time:** The time is displayed in the format Hour : Minute : Second AM/PM.
Example: 12:10:54 AM.
5. **Medium Time:** Time is displayed in hours and minutes. Seconds are not displayed.
Example: 12:10 AM.
6. **Short Time:** Time is displayed as a 24-hour clock.

For Yes/No Field

Three types of formats are given in the list:

1. Yes and No
2. True and False
3. On and Off

3. Input Mask

This property is used to specify the format in which data is entered. The Text, Number, Date/Time, and Currency fields have this type of property.

Input Mask Character	Description
#	To input number, plus sign, minus sign, and space (Not Required)
0	To input number (Required)
9	To input number or space (Not Required)
L	To input a letter (Required)
A	To input a letter or number (Required)

a	To input a letter or number (Not Required)
?	To input a letter (Not Required)
C	To input any character (Not Required)
&	To input any character (Required)
>	To input character in uppercase
<	To input character in lowercase

4. Caption

This is a common property for all fields. When a table is opened, the caption is displayed as the heading of the field.

5. Default Value

The value entered in this property is set as default for that field but this value can be changed. All fields except Autonumber and OLE object have this property.

6. Validation Rules and Validation Text

A validation rule is an expression to test the data entered. If the entered data do not follow the validation rule the validation text is displayed.

7. Required

This property makes the field 'Required' or 'Not Required.'

8. Allow Zero Length

A string with zero length can be entered in the field if this property is set to 'Yes.'

9. Index

Creates an index of a field.

18.4 SAVING THE TABLE

When you first save the table it asks you to enter a table name. If no name is entered the table is saved with the table name Table 1. Before saving the table it asks the user to define a primary key if no primary key is defined.

18.5 MODIFYING THE TABLE

To modify the table structure select the table which you want to modify and then click the **Design Command** button. The table structure is displayed. Make the changes where you want and save the table again.

18.6 IMPORTING THE TABLE

To import the table from another database, right-click and select the import option. A dialogue window is displayed to select the database file (.mdb) from where you want to import the tables. Select the database file and open it. A list of tables is displayed. Now select the tables you want to import and click the **OK** button. The selected tables are copied in your database.

Chapter 19

ORACLE

The **Oracle** database has been developed by the Oracle Corporation. There are different versions, such as Oracle7/8/8i/9i/11i. Oracle is a Relational Database Management System (RDBMS) that works on a client server basis in a multi-user environment. Oracle has two parts:

1. Oracle Server tool
2. Oracle Client tool

The Oracle Server tool is installed on a server, whereas the Oracle Client tool is installed on the client machine.

The main functions of the Oracle Server are as follows:

- Database security
- Data-sharing management
- Maintaining data integrity
- Updating the database
- Retrieving information from the database

19.1 STARTING WITH ORACLE 8

1. Install Oracle 8 on the system.
2. Go to **Start** → **Programs** → **Oracle for Windows 95** → **SQL Plus 8.0**.

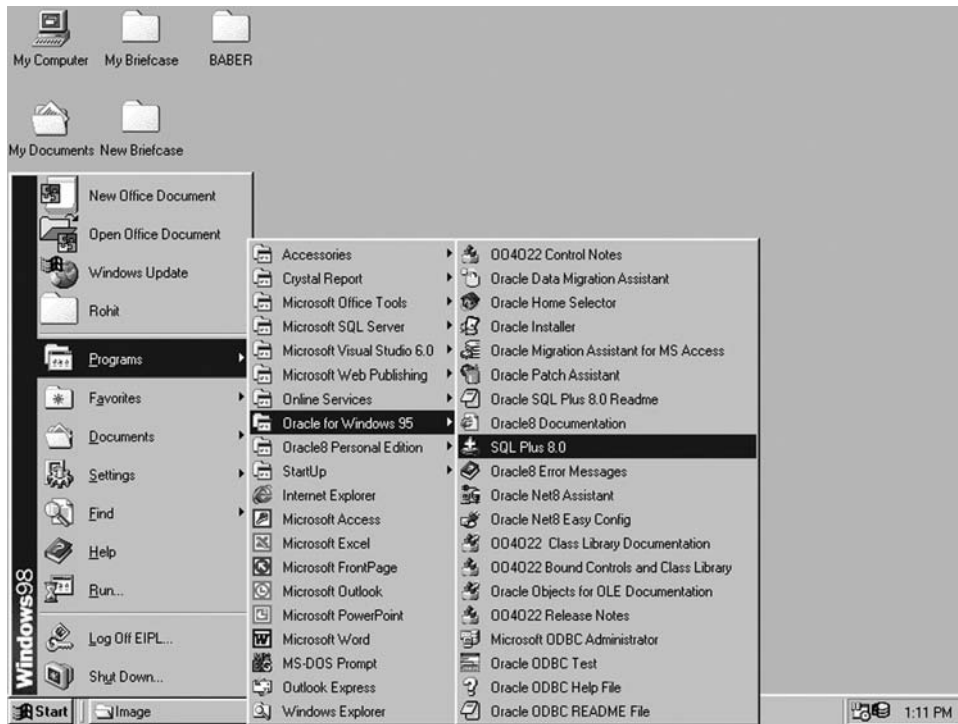


FIGURE 19.1

3. Click the SQL Plus submenu and you will see a log-on window.

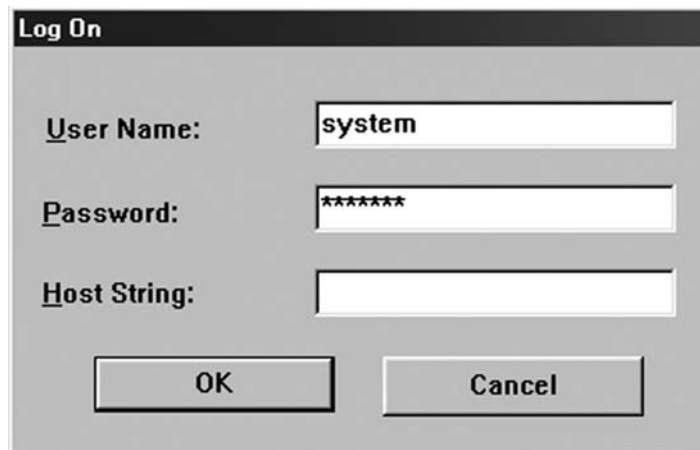


FIGURE 19.2

4. Give the username and password. In the host string box give the server name with which you want to connect. If you do not specify the host string Oracle searches for the user in the system.

Some default users are given and you can log on by using these usernames and passwords.

USERNAME	PASSWORD
System	Manager
Scott	Tiger

Give the username system and password manager and click the **OK** button. It will start the Oracle database.



FIGURE 19.3



Color figures available on the CD.

Now you can see a blue icon at the right side of the taskbar. When you are connected to Oracle a SQL> prompt appears.

19.2 HOW TO CREATE A NEW USER

You can create your own user name instead of working as a default user. It's a good practice to create your own user name and make all the tables with that user name. You can create a new user at the SQL> prompt or by using the Navigator.

User creation at the SQL> prompt

Write the following syntax:

```
SQL > create user mahesh identified by gupta;
      (a new user is created with the Username - mahesh and
       Password - gupta)
SQL > grant connect, dba, resource to mahesh;
      (Provides grants to the user - mahesh)
SQL > connect mahesh/gupta;
      (connects to the user mahesh)
```

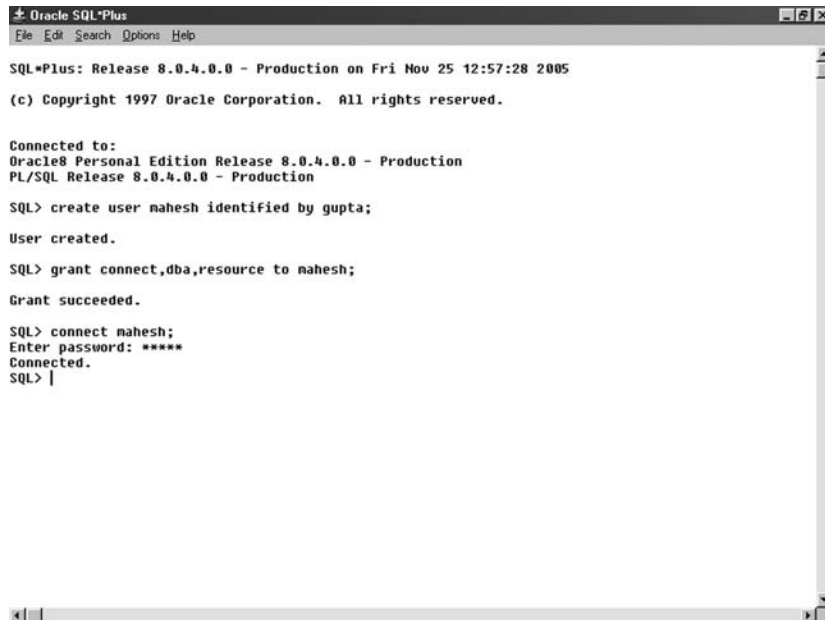


FIGURE 19.4

19.3 USER CREATION BY NAVIGATOR

1. Go to Start → Programs → Oracle8 Personal Edition → Oracle8 Navigator.

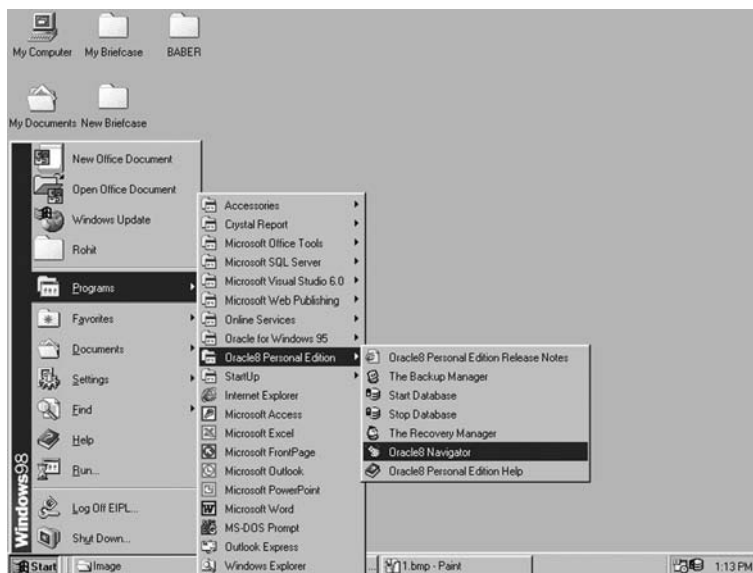


FIGURE 19.5

- Click the Oracle8 Navigator submenu and you will see a Navigator window.
- Go to the **Oracle8 Personal Edition** → **Local Database** → **User** and right-click User and select the New option (see the figure).

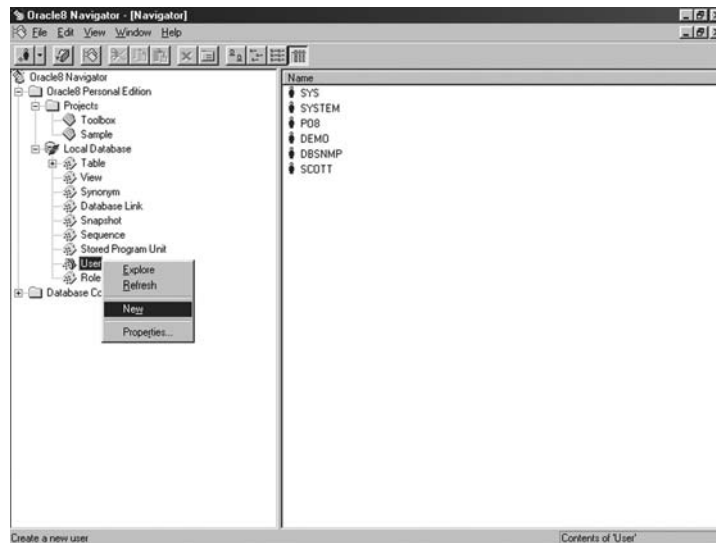


FIGURE 19.6

- In the General tab give the username and password (see the figure).

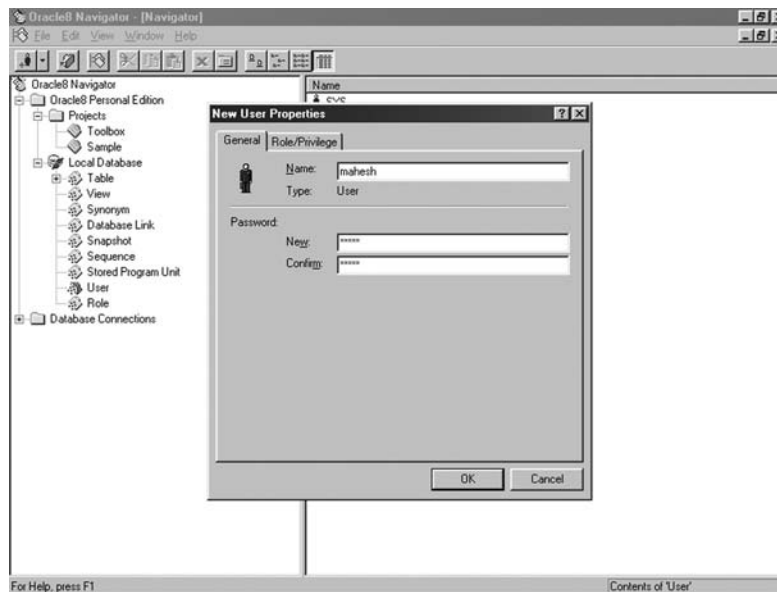


FIGURE 19.7

5. Go to the Role/Privilege tab (see the figure).

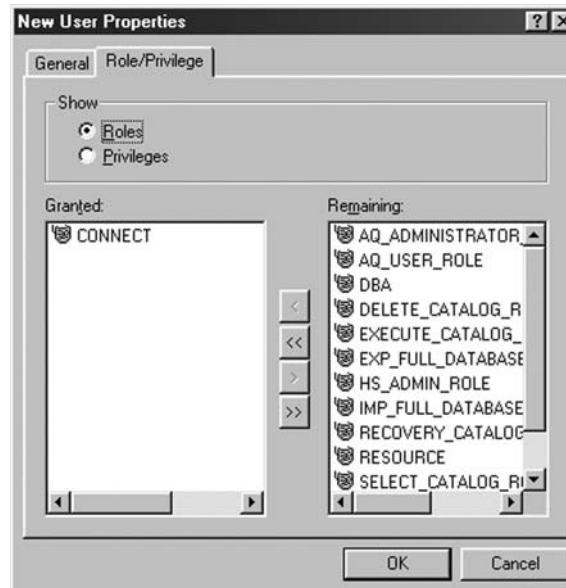


FIGURE 19.8

6. Add the DBA and RESOURCE Role from the Remaining list to the Granted list and click the **OK** button. This will create a new user (see the figure).

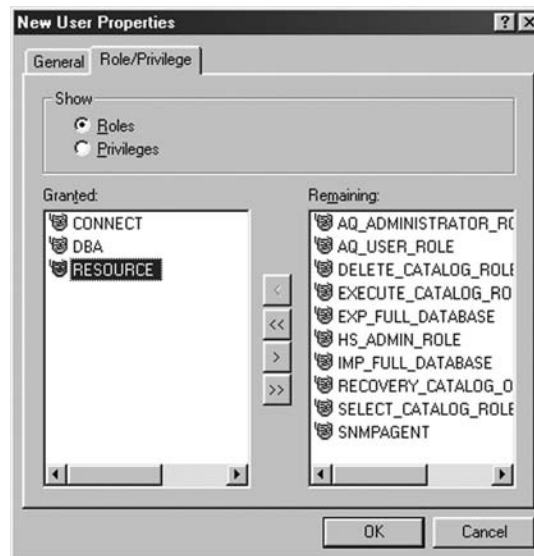


FIGURE 19.9

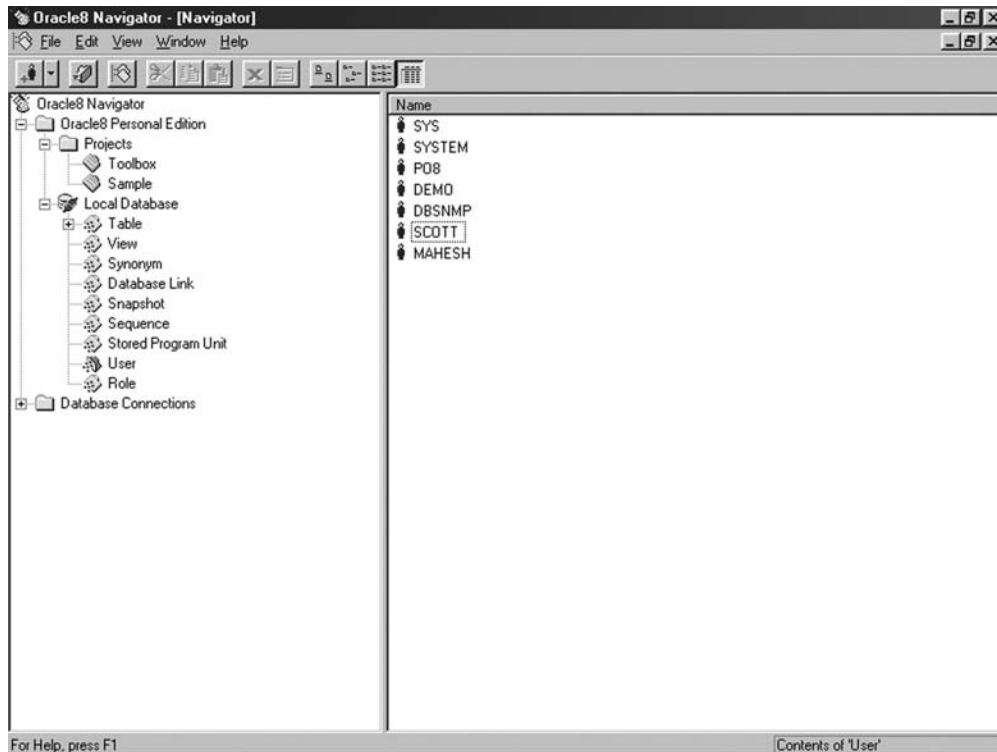


FIGURE 19.10

19.4 DATA TYPES IN ORACLE

1. **Char:** Stores alphanumeric data. Maximum size is 255 characters. If the length of the data inserted is less than the field size specified, it reserves the disk space for the rest of the field length.
2. **Varchar/Varchar2:** Stores alphanumeric data up to 2000 characters. The disk space is not reserved if the data inserted is less than the field length specified.
3. **Long:** Stores characters up to 2 GB.
4. **Number:** Stores numeric data up to 10^{125} digits. Numbers with 38 places after the decimal can be stored.
5. **Raw:** Stores binary data up to 255 characters. It can also store digitized images.
6. **Long Raw:** Stores binary data up to 2 GB.

19.5 SYNTAX AND QUERY IN ORACLE

Table Creation


Table Name – Student

Field Name	Field Type	Field Size
Name	Varchar2	30
Age	Number	2
Class	Varchar2	10
Admission_date	Date	
Father_monthly_income	Number	8.2

Table with Data

Name	Age	Class	Admission_date	Father_monthly_income
Anshul	12	Seventh	02-mar-03	20000
Shiprali	10	Fifth	12-apr-04	15000
Ishita	8	Third	02-apr-04	12000
Dinesh	15	Tenth	15-mar-01	18000
Komal	15	Tenth	25-mar-01	10000
Kavita	8	Third	22-apr-04	12000
Nishita	18	Twelfth	12-apr-04	22000
Nishant	15	Tenth	18-mar-01	14000
Ritika	12	Seventh	10-mar-01	8000

```
SQL> create table student (Name varchar2(30), Age number(2),
Class varchar2(10), Admission_date date, Father_monthly_income
number(8.2));
```

NOTE  Each statement is terminated with a semicolon.

To View All the Tables of the Currently Connected User

Syntax

```
SQL> select * from tab;
```

This statement will show all tables of the user.

To View the Structure of a Table

Syntax

```
SQL > desc student;
```

OR

```
SQL > describe student;
```

This statement will show the structure of the table student.

Creation of a Table From Another Table

Syntax

```
SQL > create table studentnew as select * from student;
```

Here studentnew is a new table to be created. The structure of the table studentnew will be the same as the table student. Data from the old table student will also be copied into the new table studentnew. If you do not want to copy the data into the table and want a table without any record, you can give a condition that is not being fulfilled.

Syntax

```
SQL> create table studentnew as select * from student where Age < 0;
```

You can create a new table with only a few selected fields from another table.

Syntax

```
SQL> create table studentnew (name,age,class) as select  
name,age,class from student;
```

Editing SQL Statements

In Oracle once the SQL statement is executed it cannot be modified. Yet there is another way to modify the SQL statement. It can be opened in Windows Notepad to edit it. To open the previously executed statement, type 'ed' at the SQL> prompt and press the Enter key.

```
SQL> ed
```

You will see Windows Notepad with the previously executed statement.

Follow these steps:

1. Modify the statement or check the statement to remove any errors.
2. Remove the forward slash (/) from the end of the last line.
3. Save the file.
4. Exit Notepad.
5. Type a forward slash (/) at the SQL> prompt and press the Enter key.

This will execute the statement corrected in Notepad.

Renaming a Table

To rename the table studentnew to studentold:

Syntax

```
SQL> rename studentnew to studentold;
```

Deleting a Table

Syntax

```
SQL> drop table studentold;
```

Modifying the Structure of a Table

There are some restrictions in modifying the structure of a table:

1. A column cannot be dropped.
2. The column name cannot be changed.
3. The size of the column cannot be decreased if data exists.
4. The data type of the column cannot be changed if data exists.

Modifying the Data Type and Size of the Column

Syntax

```
SQL> alter table student modify(Class number(14));
```

This statement changes the data type from varchar2 to number and size 10 to 14.

To Add a New Column in a Table

Syntax

```
SQL> alter table student add(date_of_birth date,address  
varchar2(50));
```

Data Insertion in a Table

Character and date values are inserted with single quotes (') at the beginning and at the end, whereas numeric values are inserted without single quotes. The date value is inserted in the format 'dd-mmm-yy', i.e., '01-APR-05.'

Syntax

```
SQL> insert into student values ('Anshul',12,'Seventh','02-MAR-  
03',20000);
```

Data Selection

To select all the records from a table:

Syntax

```
SQL> select * from student;
```

To select few fields from a table:

Syntax

```
SQL> select Name,Age,Class from student;
```

This statement will show only Name, Age, and Class of all records from the table student.

Conditional Searching

The records can be selected on the basis of a given condition by using 'Where' followed by the condition statement.

To search all the records of the student table where the name is 'Dinesh':

Syntax

```
SQL> select * from student where name = 'Dinesh';
```

To search all the records of the student table where age is less than 12 and father_monthly_income is less than 20000:

Syntax

```
SQL> select * from student where age <12 and father_monthly_income < 20000;
```

To select all the records from the student table where age is less than 12 and class is not Fifth:

Syntax

```
SQL> select * from student where age < 12 and class <> 'Fifth';
```

Here '<>' refers to the 'not equal to' operator.

To select records in ascending order of name:

Syntax

```
SQL> select * from student order by name;
```

To select records in descending order of name:

Syntax

```
SQL> select * from student order by name desc;
```

Records can be arranged in ascending or descending order on the basis of more than one field. For example, if you want to arrange the records in ascending order of name and under name you want to arrange them in the order of admission_date, look at the following syntax:

Syntax

```
SQL> select * from student order by name, admission_date;
```

To select unique records from the table:

Syntax

```
SQL> select distinct * from student;
```

To select unique columns from the table:

Syntax

```
SQL> select distinct Class from student;
```

Insertion of Data from Another Table

To insert all data in a table from another table:

Syntax

```
SQL > insert into studentnew select * from student;
```

To insert selected data in a table from another table:

Syntax

```
SQL > insert into studentnew select * from student where admission_
date < '01-APR-05';
```

To insert a selected column in a table from another table:

Syntax

```
SQL > insert into studentnew (name,age) select name,age from
student;
```

Modifying a Record

To modify the name of an existing record:

Syntax

```
SQL > update student set name ='Mahesh' where name = 'Dinesh';
```

To modify more than one column:

Syntax

```
SQL > update student set name = 'Mahesh', age = 14 where name =
'Dinesh';
```

To increment the father_monthly_income by 2000 for a given student:

Syntax

```
SQL > update student set father_monthly_income = father_monthly_
income + 2000 where name = 'Ishita';
```

Deleting Records

To delete all records from a table:

Syntax

```
SQL> delete from student;
```

To delete selected records from a table:

Syntax

```
SQL> delete from student where age < 12;
```

Use of NOT

To select all records from the student table except class 'Tenth' and 'Third':

Syntax

```
SQL> select * from student where not (Class = 'Tenth' or Class =  
'Third');
```

Use of BETWEEN

To select all records from the student table where father_monthly_income is between 15000 and 20000:

Syntax

```
SQL> select * from student where father_monthly_income between  
15000 and 20000;
```

Use of LIKE

LIKE is used to compare two strings:

% Sign is used to match string

_ Sign is used to match character

To select all records from the student table where the name starts with the letter 'A':

Syntax

```
SQL> select * from student where name like 'A%';
```

To select all records from the student table where the name ends with the letter 'A':

Syntax

```
SQL> select * from student where name like '%A';
```

To select all records from the student table where the name starts with the letter 'A' and ends with the letter 'A':

Syntax

```
SQL> select * from student where name like '%A%';
```

To select all records from the student table where the second character of the name is 'h':

Syntax

```
SQL> select * from student where name like '_h%';
```

Use of IN

To select all records from the student table where the class is 'Tenth' or 'Fifth':

Syntax

```
SQL> select * from student where class in ('Tenth','Fifth');
```

Use of NOT IN

To select all records from the student table where the class is neither 'Tenth' nor 'Fifth':

Syntax

```
SQL> select * from student where class not in ('Tenth', 'Fifth');
```

Grouping of Data

Sometimes you may need the data in a group. Suppose you want to see all the values of the column without duplicacy and want to make them groups.

To group the class in the student table:

Syntax

```
SQL> select class from student group by class;
```

To create one group under another group: (for example, to group the admission_date under the class)

Syntax

```
SQL> select class,admission_date from student group by  
class,admission_date;
```


19.6 FUNCTIONS

1. **Abs()** — Returns the absolute value of an integer, i.e., converts a negative value into a positive value.

Example: Select abs(field name) from tablename;

2. **Avg()** — Returns the average value and ignores null values.

Example: Select avg(field name) from tablename;

3. **Count()** — Returns the total number of rows in a table.

Example: Select count(*) from tablename;

4. **Initcap()** — Returns a string with the first letter in a capital letter.

Example: Select initcap('mahesh') from dual;

The function returns a string 'Mahesh.'

NOTE Dual is an Oracle work table where you can perform any operation.

5. **Length()** — Returns the length of the column value.

Example: Select length (field name) from tablename;

6. **Ltrim()** — Removes leading spaces from a string.

Example: Select ltrim('Ajay') from tablename;

The function returns 'Ajay.'

7. **Lower()** — Converts capital letters into small letters.

Example: Select lower('HELLO') from dual;

The function returns 'hello.'

8. **Max()** — Returns the maximum value of the column.

Example: Select max(father_monthly_income) from student;

The function returns 22000.

9. **Min()** — Returns the minimum value of the column.

Example: Select min(father_monthly_income) from student;

The function returns 8000.

10. **Round()** — Rounds off the number to given decimal places. If no decimal place is defined an integer is returned without decimal places.

Example 1: Select round(10.21) from dual;

The function returns 10.

Example 2: Select round(10.85) from dual;

The function returns 11.

Example 3: Select round(10.855, 2) from dual;

The function returns 10.86.

11. **Rtrim()** — Removes trailing spaces from a string.
Example: Select rtrim('Ajay ') from dual;
The function returns 'Ajay.'
12. **Sqrt()** — Returns the square root of a number.
Example: Select sqrt(100) from dual;
The function returns 10.
13. **Substr()** — Returns a string containing a specified number of characters from a given position. If the number of characters to be extracted is not specified, it returns a string from a given position to the last position.
Example 1: Select substr('Bombay',4,2) from dual;
The function returns 'ba'.
Example 2: Select substr('Bombay',4) from dual;
The function returns 'bay.'
14. **Sum()** — Returns the sum of the given column.
Example: Select sum(father_monthly_income) from student;
The function returns 131000.
15. **Sysdate()** — Returns current system date.
Example: Select sysdate from dual;

19.7 PRIMARY KEYS

Adding a Primary Key

If you want to set any column as a primary key, you can indicate it during table creation.

Example: To create a table 'Employee' with fields id, name, and address, set the id field as the primary key.

Syntax

```
SQL> create table employee (id varchar2(5) primary key, name  
varchar2(50), address varchar2(100));
```

Adding a Primary Key in an Existing Table

You can set a primary key in an existing table, provided that there are no duplicate values in that column.

Syntax

```
SQL> alter table employee add primary key(id);
```

Deleting a Primary Key

You can unset the primary key constraint from a field.

Syntax

```
SQL> alter table employee drop primary key;
```

19.8 DATA EXPORT

- Run the file c:\orawin95\bin\exp80.exe
- A DOS Window will appear. Give the username and password and press <ENTER>.

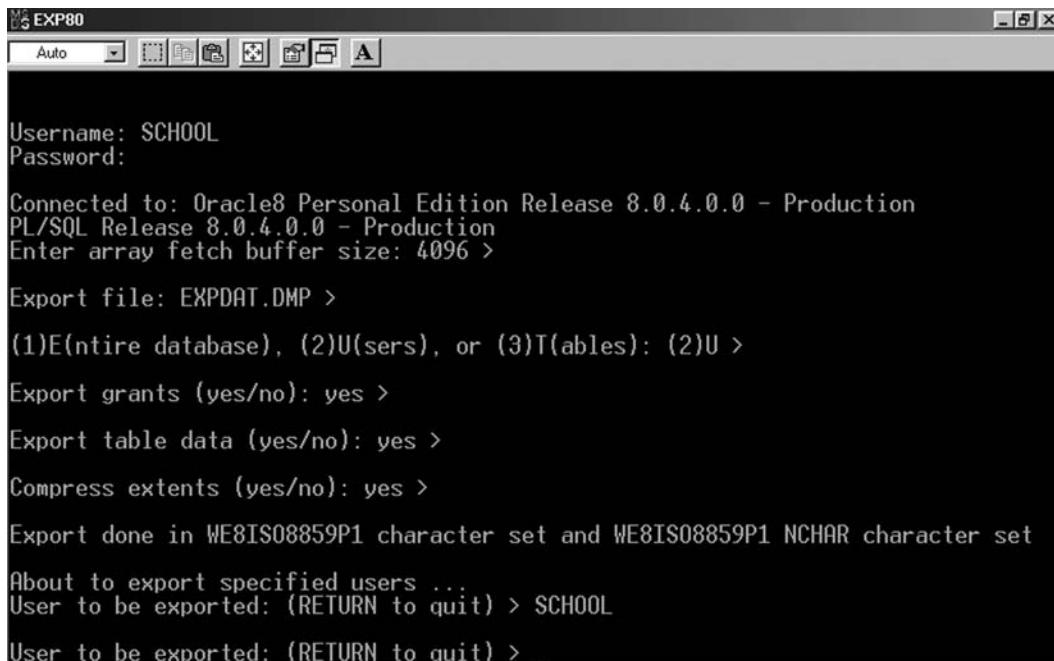


FIGURE 19.11

The following messages will appear; reply at each prompt:

Export file: EXPDAT.DMP >	Give the filename in which the data is to be exported. To give the default filename 'EXPDAT.DMP' press <ENTER>.
(1)E(ntire database), (2)U(sers), or (3)T(ables): (2)U >	Enter 1 if you want to export all databases with all users. Enter 2 if you want to export selected users. Enter 3 if you want to export selected tables.
Export grants (yes/no): yes >	Press <ENTER> (by default yes is given)
Export table data (yes/no): yes >	Press <ENTER> (give yes if you want to export tables with data; otherwise, give no, by default yes is given).
Compress extents (yes/no): yes >	Press <ENTER>.
User to be exported: (RETURN) to quit >	Give username to be exported.
User to be exported: (RETURN) to quit >	Press <ENTER> if you do not want to export other users.

This will export all/given users/tables with/without data with the given filename .dmp in the folder c:\orawin95\bin.

You can use this file to import data for another user. Rename this file as expdat.dmp before importing if you have changed the export file name during export.

19.9 DATA IMPORT

- First create the user for which you want to import data.
- Copy the file expdat.dmp to the folder c:\orawin95\bin.
- Run the file c:\orawin95\bin\Imp80.exe.
- A DOS window will appear. Give the username and password and press <ENTER>.

```

IMP80
Auto
Import: Release 8.0.4.0.0 - Production on Fri Nov 18 11:12:34 2005
(c) Copyright 1997 Oracle Corporation. All rights reserved.
Username: SCHOOL
Password:
Connected to: Oracle8 Personal Edition Release 8.0.4.0.0 - Production
PL/SQL Release 8.0.4.0.0 - Production
Import file: EXPDAT.DMP >
Enter insert buffer size (minimum is 4096) 30720>
Export file created by EXPORT:V08.00.04 via conventional path
List contents of import file only (yes/no): no >
Ignore create error due to object existence (yes/no): no >
Import grants (yes/no): yes >
Import table data (yes/no): yes >
Import entire export file (yes/no): no > Y

```

FIGURE 19.12

The following messages will appear; reply at each prompt:

Enter insert buffer size (minimum is 4096) 30720>	Press <ENTER>.
List the contents of import file only (yes/no): no>	Press <ENTER> (by default no is given).
Ignore create error due to object existence (yes/no): no >	Press <ENTER> (by default no is given).
Import grants (yes/no): yes >	Press <ENTER> (by default yes is given).
Import table data (yes/no): yes >	Press <ENTER> (give yes if you want to import tables with data; otherwise, give no, by default yes is given).
Import entire export file (yes/no): no >	Give yes if you want to import all the tables of the user; otherwise, give no and give the table names one by one, by default no is given).

This will import all/given tables with/without data from the file expdat.dmp to the current user.

Chapter 20

SQL SERVER 2000

20.1 WHAT'S NEW IN MICROSOFT SQL SERVER 2000?

Microsoft SQL Server 2000 includes several new features that make it an excellent database platform for large-scale Online Transactional Processing (OLTP), data warehousing, and e-commerce applications. Microsoft SQL Server 2000 extends the performance, reliability, quality, and ease-of-use of Microsoft SQL Server version 7.0. It is a much more powerful database than MS Access.

20.1.1 SQL Query Analyzer

SQL Query Analyzer includes a stored procedure debugger. It can be used for creating objects, such as database, tables, views, and stored procedures.

20.2 STARTING MICROSOFT SQL SERVER 2000

Before Running SQL Server 2000 Setup

- Create one or more domain user accounts, if installing SQL Server 2000 on a computer running Microsoft Windows NT or Microsoft Windows

2000 and you want SQL Server 2000 to communicate with other clients and servers.

- Log on to the operating system under a user account that has local administrative permissions or assign the appropriate permissions to the domain user account.
- Shut down all services dependent on SQL Server. This includes any service using ODBC, such as Microsoft Internet Information Services (IIS).
- Shut down Microsoft Windows NT Event Viewer and registry viewers (Regedit.exe or Regedt32.exe).

20.2.1 Operating System Requirements

The following table shows the operating systems that must be installed to use the various editions or components of Microsoft SQL Server 2000.

SQL Server edition or component	Operating System Requirements
Standard Edition	Microsoft Windows NT Server 4.0, Windows 2000 Server, Microsoft Windows NT Server Enterprise Edition, Windows 2000 Advanced Server, and Windows 2000 Data Center Server.
Personal Edition	Microsoft Windows Me, Windows 98, Windows NT Workstation 4.0, Windows 2000 Professional, Microsoft Windows NT Server 4.0, Windows 2000 Server, and all the more advanced Windows operating systems.
Developer Edition	Microsoft Windows NT Workstation 4.0, Windows 2000 Professional, and all other Windows NT and Windows 2000 operating systems.
Client Tools Only	Microsoft Windows NT 4.0, Windows 2000 (all versions), Windows Me, and Windows 98.
Connectivity Only	Microsoft Windows NT 4.0, Windows 2000 (all versions), Windows Me, Windows 98, and Windows 95.

NOTE

Microsoft Windows NT Server 4.0, Service Pack 5 (SP5), or later must be installed as a minimum requirement for all SQL Server 2000 editions.

SQL Server 2000 is not supported on Windows NT 4.0 Terminal Server.

For installation of SQL Server 2000 Personal Edition on Windows 98 computers without a network card, Windows 98 Second Edition is required.

20.3 INSTALLATION OF SQL SERVER 2000

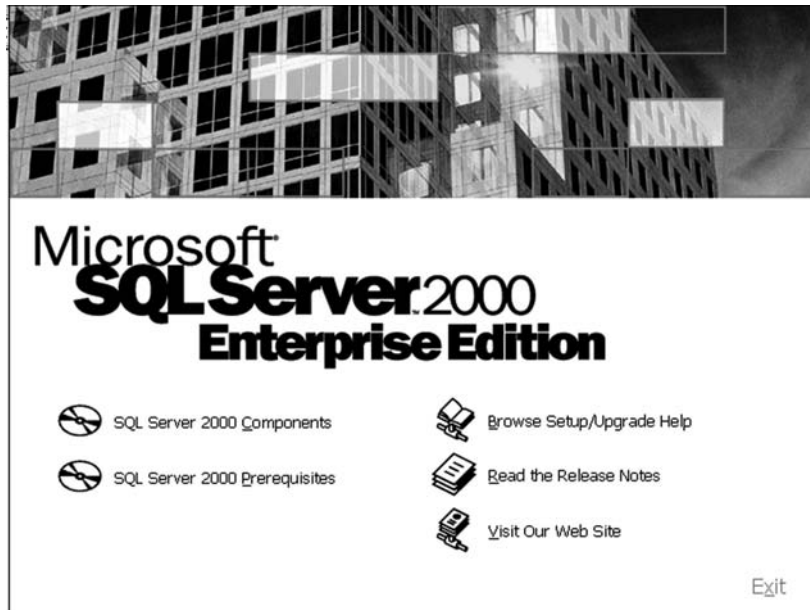


FIGURE 20.1

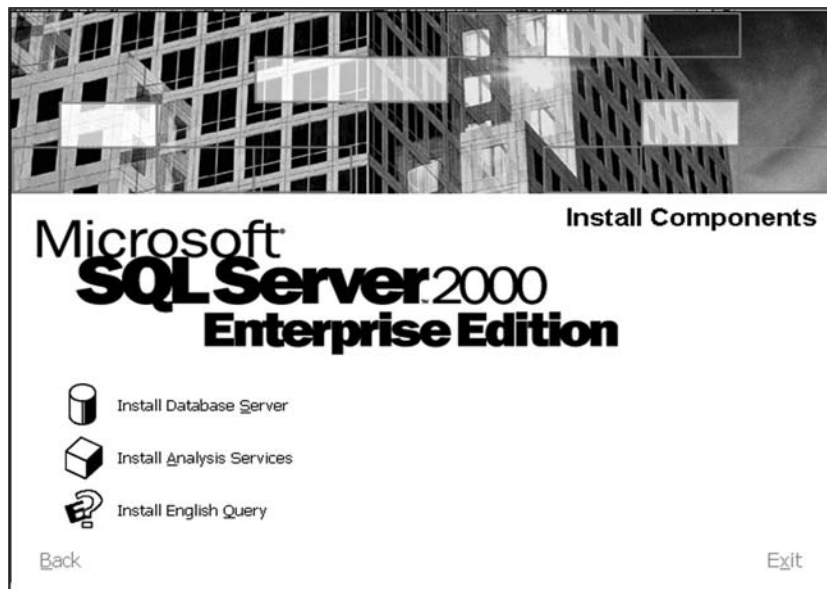


FIGURE 20.2

When you select **SQL Server 2000 Components** on the first screen, three options appear on the **Install Components** screen:

Install Database Server

Starts SQL Server Setup, with screens for selecting installation options.

Install Analysis Services

Installs Analysis Services on computers processing OLAP cubes.

Install English Query

Installs English Query on computers running English Query applications.

20.3.1 Choosing Components and Options to Install

You may have a database server, an internet server, or require a database on a client computer. If running database client/server applications, you may or may not require a database on your computer. You may need tools to administer a database server or you may want to run applications that access an instance of SQL Server.

20.3.2 Installing SQL Server on a Database Server

If installing a database server, install either SQL Server 2000 Enterprise Edition or SQL Server 2000 Standard Edition. If installing a personal database on your workstation, install SQL Server 2000 Personal Edition. These installations typically include the database engine, the client database-management tools, and the client connectivity components.

On a database server, you can install a default instance of SQL Server 2000 relational database engine. You can also install one or more named instances of the SQL Server 2000 database engine. Other than specifying an instance name, the setup options are similar to those for installing a default instance.

20.3.3 Using SQL Server with Client/Server Applications

For a computer running database client/server applications, such as Microsoft Visual Basic applications that connect directly to an instance of SQL Server, you have several options:

- If you require a personal database on your client computer, install the Personal Edition of SQL Server. This setup typically installs the client tools and client connectivity components along with the database engine.

- If you do not require a database on your computer but need to administer an instance of SQL Server on a database server or plan to develop SQL Server applications, install the option for Client Tools Only. This option includes the client connectivity components.
- If you want to run applications that access instances of SQL Server on database servers, install the Connectivity Only components.

20.3.4 Other SQL Server Components

For distributing SQL Server 2000 with applications, use the SQL Server 2000 Desktop Engine, a stand-alone database engine that independent software vendors can package with their applications.



NOTE The Desktop Engine has no graphical user interface and is not related to the SQL Server 7.0 Desktop Edition.

20.4 CREATING A DATABASE

20.4.1 Before Creating a Database, Consider That:

- The user who creates the database becomes the owner of the database.
- A maximum of 32,767 databases can be created on a server.
- The name of the database must follow the rules for identifiers.

Three types of files are used to store a database:

Primary files

These files contain the startup information for the database. The primary files are also used to store data. Every database has one primary file.

Secondary files

These files hold all the data that does not fit in the primary data file. Databases do not need secondary data files if the primary file is large enough to hold all the data in the database. Some databases may be large enough to need multiple secondary data files or they may use secondary files on separate disk drives to spread the data across multiple disks.

Transaction log

These files hold the log information used to recover the database. There must be at least one transaction log file for each database, although there may be more than one. The minimum size for a log file is 512 kilobytes (KB).

NOTE

Microsoft SQL Server 2000 data and transaction log files must not be placed on compressed file systems or a remote network drive (shared network directory).

It is recommended that you specify a maximum size to which the file is permitted to grow. This prevents the file from growing as data is added, until disk space is exhausted. To specify a maximum size for the file, go to the **Restrict filegrowth (MB)** option in the **Properties** dialogue box in the SQL Server Enterprise Manager.

20.5 HOW TO CREATE A DATABASE USING ENTERPRISE MANAGER

1. Expand the Microsoft SQL Servers in the Console Root and then expand the SQL Server Group.

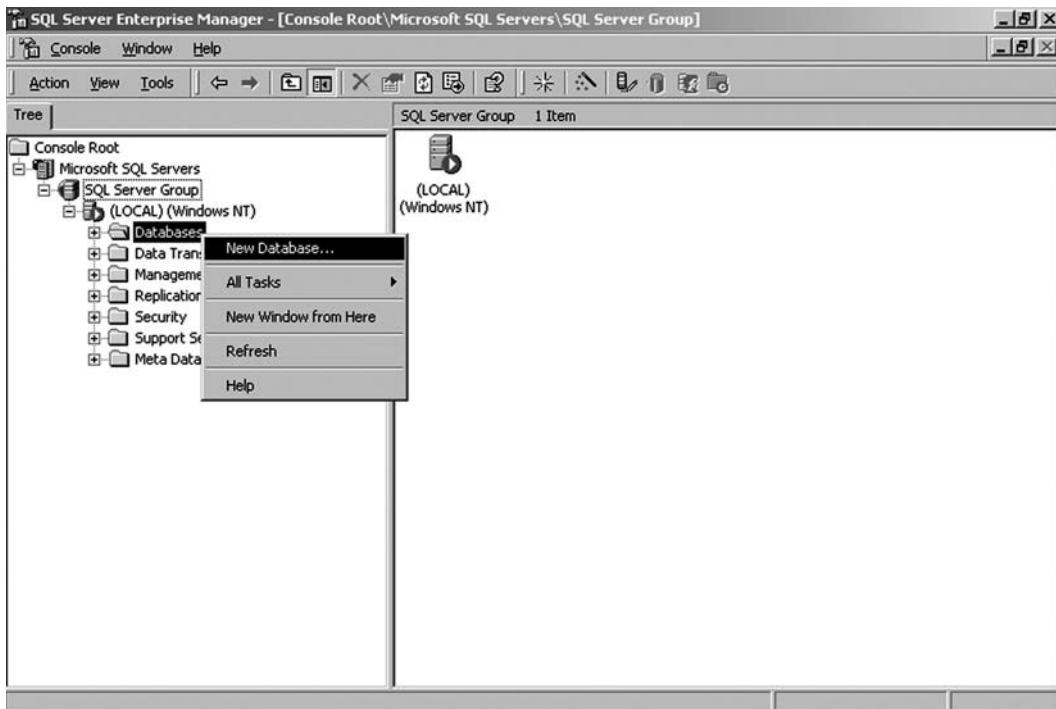


FIGURE 20.3

2. Go to the Local group and right-click **Databases** and then click the **New Database** option.

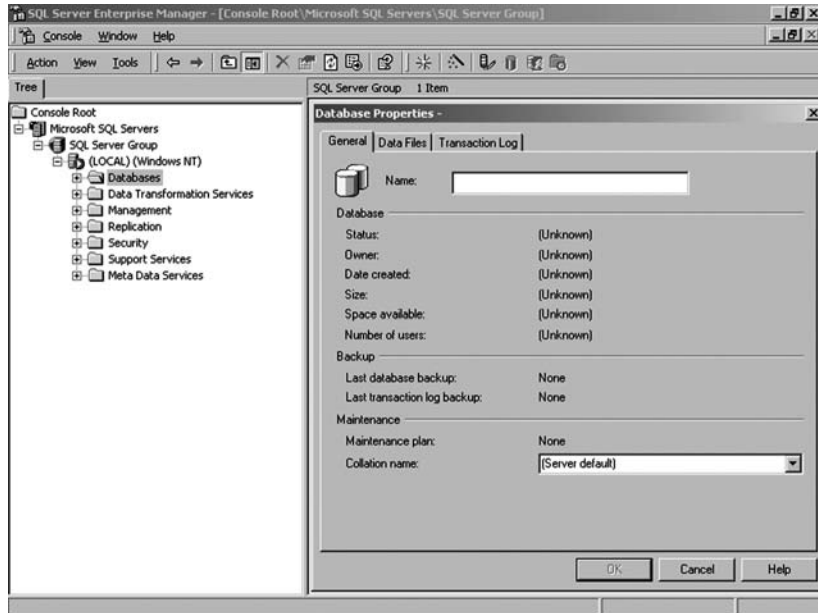


FIGURE 20.4

3. Enter a name for the new database in General properties.

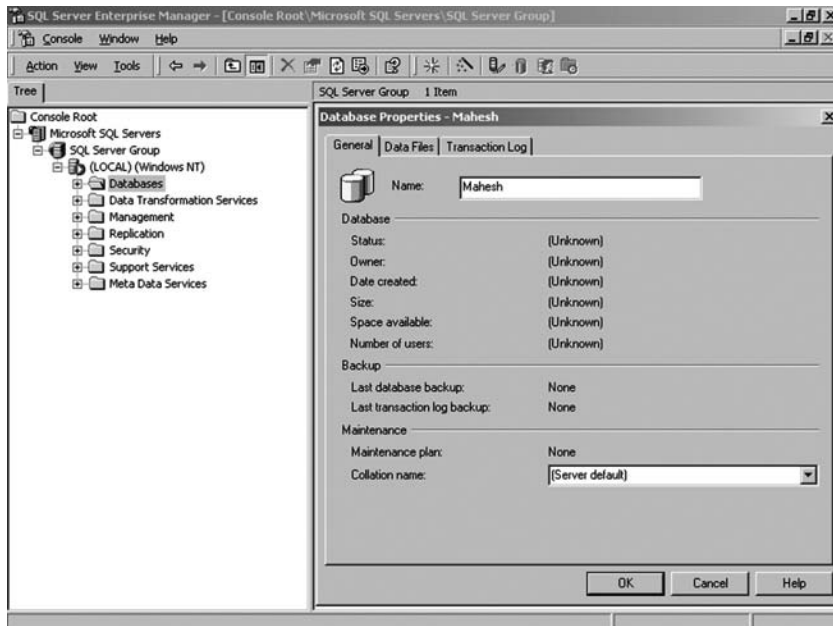


FIGURE 20.5

4. Go to the Data Files tab and select the path by clicking the **Location** button to save the data files and then click the **OK** button.

Click this button to select the storage path.

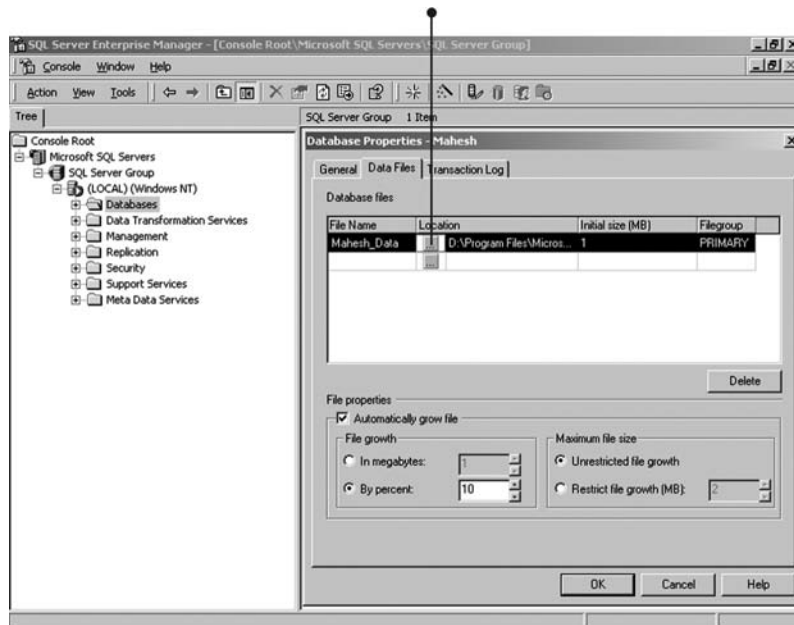


FIGURE 20.6

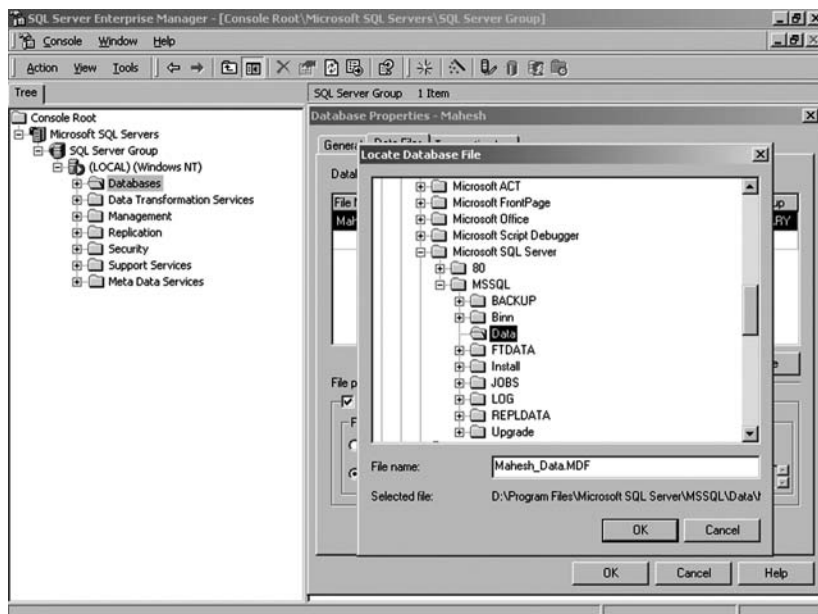


FIGURE 20.7

- Go to the Transaction Files tab and select the path by clicking the **Location** button to save the log files and then click the **OK** button.

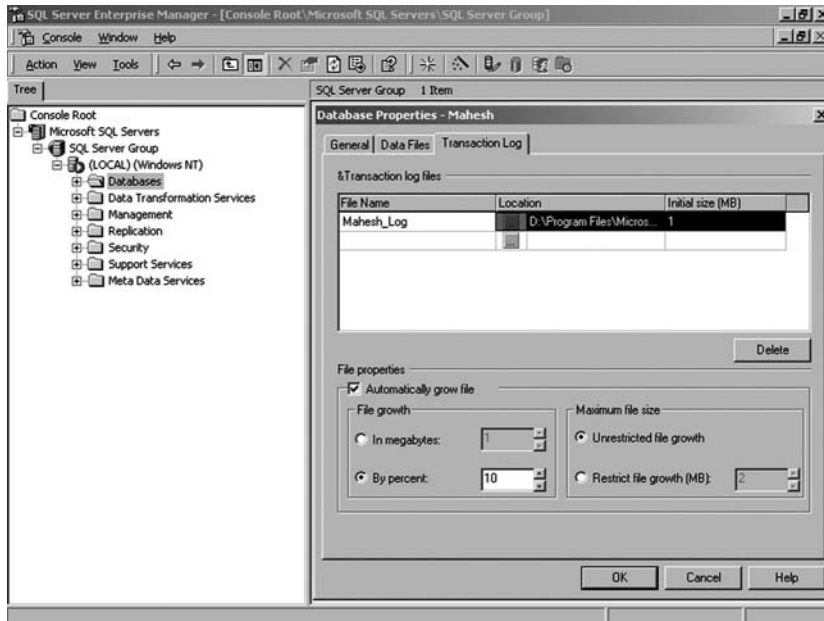


FIGURE 20.8

- A new database will be created with the name specified and the new database will appear in the database list.

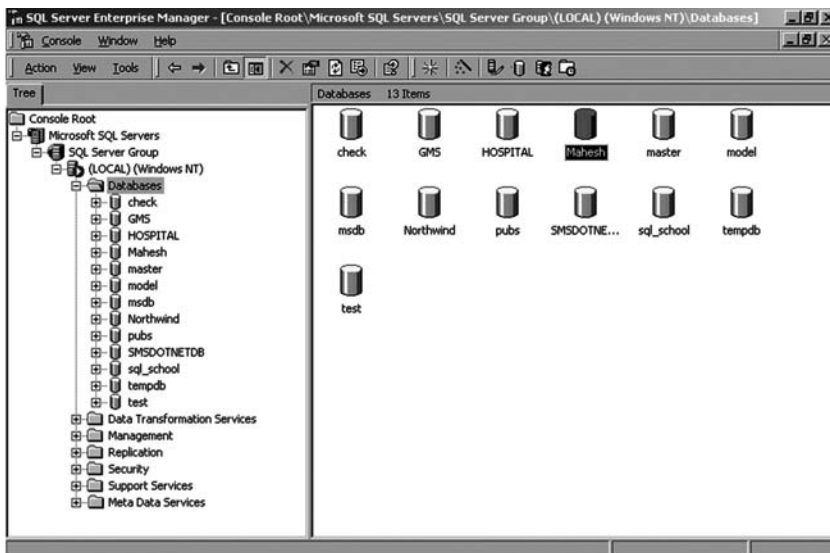


FIGURE 20.9

The primary database and transaction log files are created using the database name you specified as the prefix; for examples, Mahesh_Data.MDF and Mahesh_Log.LDF. The primary file contains the system tables for the database.

7. To change the default values for the new primary database file, click the General tab. To change the default value for the new transaction log file, click the Transaction Log tab.
8. To change the default values provided in the filename, location, initial size (MB), and file group (not applicable for the transaction log) columns, click the appropriate box to change and enter the new value.
9. To specify how the file should grow, select these options:
 - To allow the currently selected file to grow as more data space is needed, select **Automatically grow file**.
 - To specify that the file should grow by fixed increments, select **In megabytes** and specify a value.
 - To specify that the file should grow by a percentage of the current file size, select **By percent** and specify a value.
10. To specify the file-size limit, select these options:
 - To allow the file to grow as much as necessary, select **Unrestricted filegrowth**.
 - To specify the maximum size the file should be allowed to grow to, select **Restrict file growth (MB)** and specify a value.

20.6 CREATE A DATABASE USING THE CREATE DATABASE WIZARD IN ENTERPRISE MANAGER

1. Expand the Microsoft SQL Servers and the SQL Server Group and then expand the Server in which you want to create the database.
2. On the **Tools** menu, click **Wizards**.
3. Expand **Database**.
4. Double-click **Create Database Wizard**.
5. Complete the steps in the wizard.

20.7 CREATING A NEW TABLE

Go to the database **Mahesh** in SQL Server and select **Tables**. Twenty system tables will be displayed in the list. You can create new tables in the currently connected database.

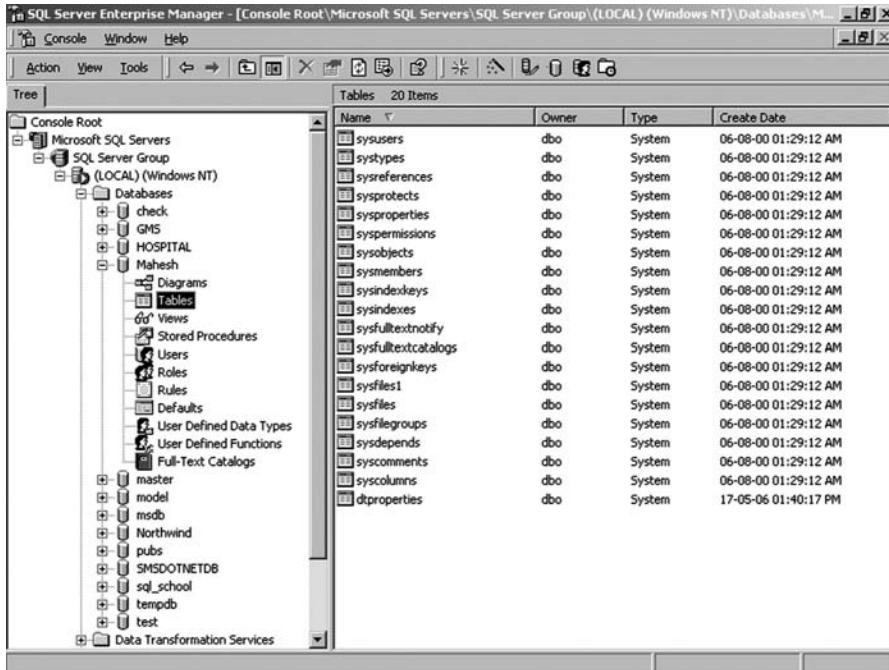


FIGURE 20.10

To create a new table, right-click at the right side of the window and select the option **New Table**.

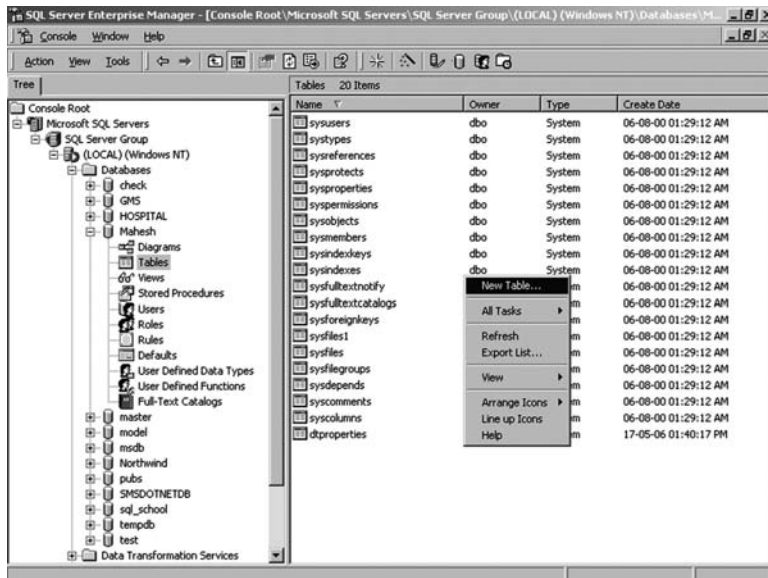


FIGURE 20.11

A screen, as shown in the following figure, will be displayed.

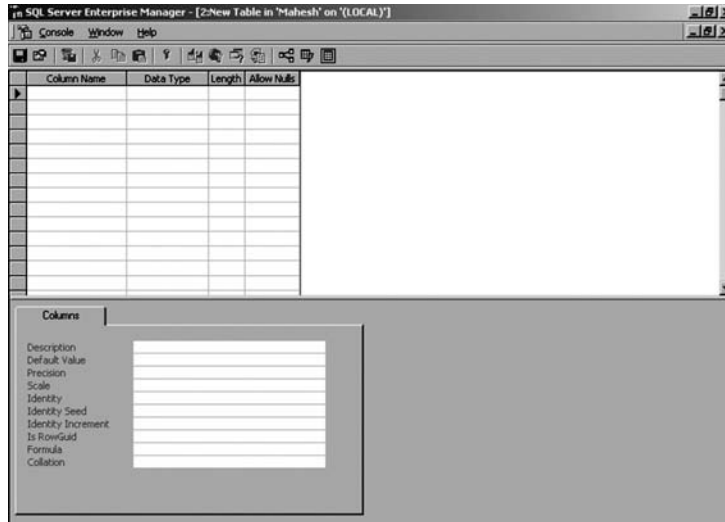


FIGURE 20.12

Give the Column Name in the first column, select Data Type in the second column, and specify Column Length in the third column and check or uncheck the Allow Nulls column to specify whether the Null values are allowed in this column. If the **Length** column is active, enter another value if you want to change the maximum data length that the data type can store.

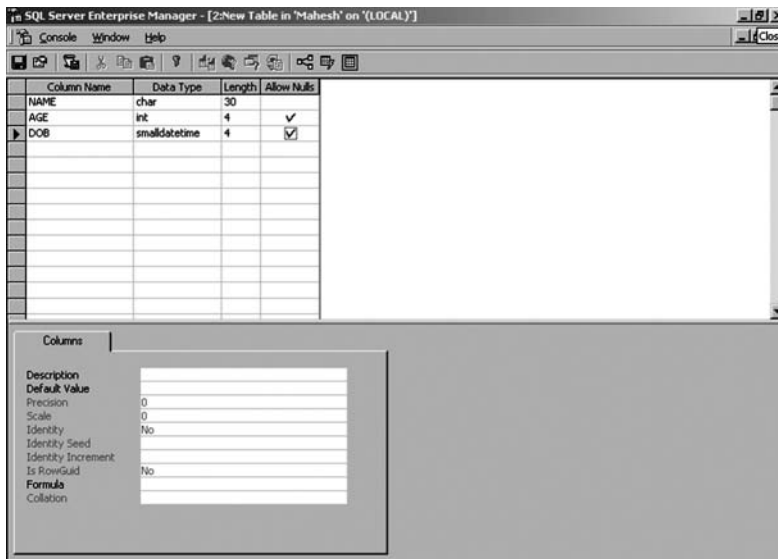


FIGURE 20.13

You can give a default value to any column in its **Default** property. Now save the table after designing the structure.

20.8 DATA TYPES

S.No.	Data Type	Length	Description
1	Bigint	8	Accepts numbers up to 19 digits.
2	Binary	50	Accepts binary numbers.
3	Bit	1	Accepts numbers up to 1 digit only.
4	Char	10	Accepts characters up to 10 digits.
5	Datetime	8	Accepts date in the format 'dd-mm-yy.'
6	Decimal	9	Accepts only integers without scale (does not accept decimal numbers).
7	Float	8	Accepts numbers with scale.
8	Image	16	Accepts binary numbers.
9	Int	4	Accepts numbers up to 10 digits.
10	Money	8	Accepts numbers up to 19 digits, out of which 4 are used to store decimal numbers.
11	Nchar	10	Accepts characters.
12	Ntext	16	Accepts characters.
13	Numeric	9	Accepts numbers.
14	Nvarchar	50	Accepts characters.
15	Real	4	Accepts numbers and converts the numbers above 8 digits into exponent form.
16	Smalldatetime	4	Accepts dates in the format 'dd-mm-yy.'
17	Smallint	2	Accepts numbers up to 5 digits.
18	Smallmoney	4	Accepts numbers with scale.
19	Sql_variant		Accepts binary numbers.
20	Text	16	Accepts characters.

21	Timestamp	8	Accepts binary numbers.
22	Tinyint	1	Accepts numbers up to 3 digits.
23	Uniqueidentifier	16	Stores 16 bytes binary values that operate as globally unique identifiers (GUIDs).
24	Varbinary	50	Accepts binary numbers.
25	Varchar	50	Accepts characters.

The data types that can have variable lengths are **binary**, **char**, **nchar**, **nvarchar**, **varbinary**, and **varchar**. The length of an **image**, **binary**, and **varbinary** data type is defined in bytes. The length of the numeric data types is the number of bytes required to hold the number of digits allowed for that data type. The length of the character string and Unicode data types is defined in characters.

20.8.1 Viewing the Structure of the Table

To view the structure of the table, double-click the table or select the table and press the Enter key.

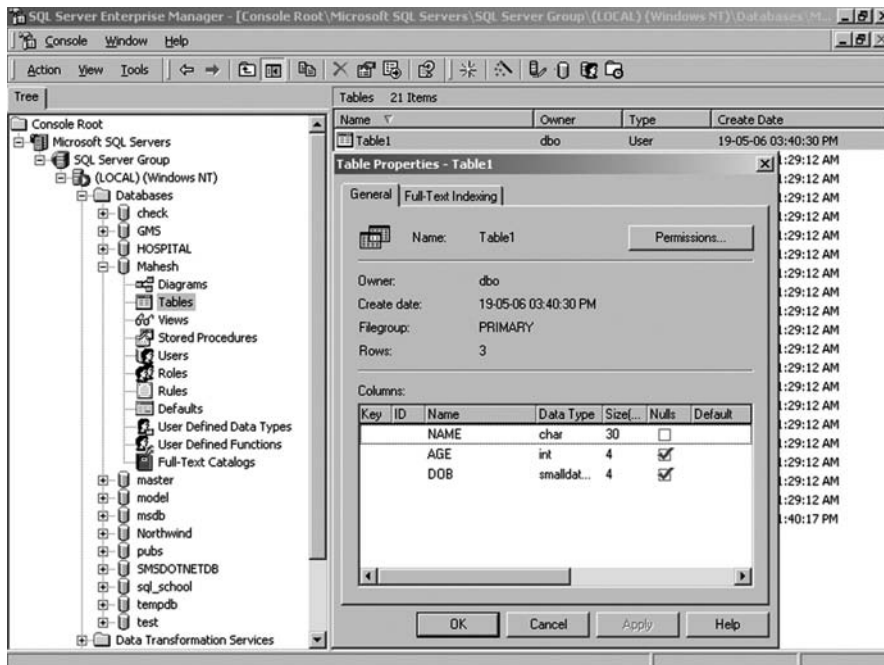


FIGURE 20.14

20.8.2 Modifying the Structure of an Existing Table

To modify the structure of an existing table, right-click the table and select the **Design Table** option.

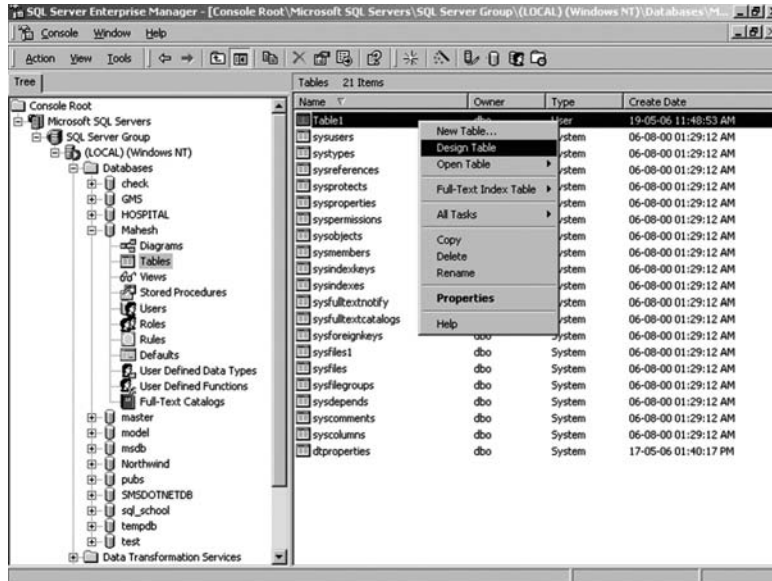


FIGURE 20.15

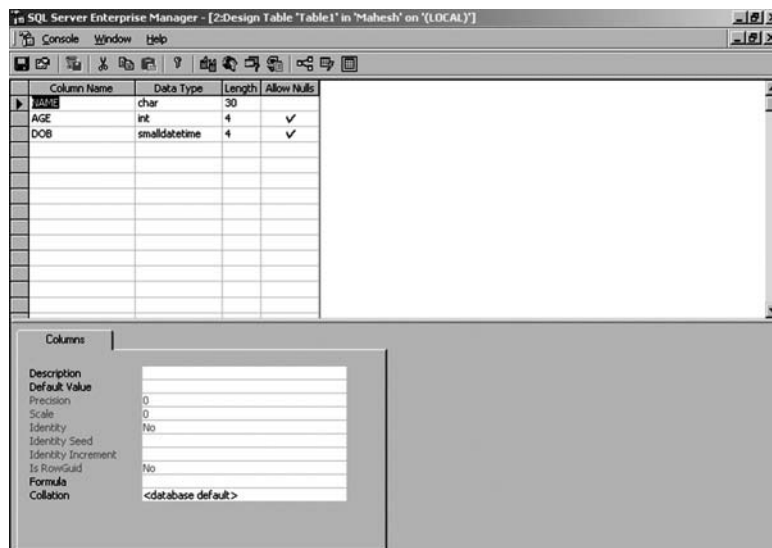


FIGURE 20.16

Make the changes in the table structure and save the table.

20.8.3 Dropping a Table

To drop a table, right-click the table and select the **Delete** option.

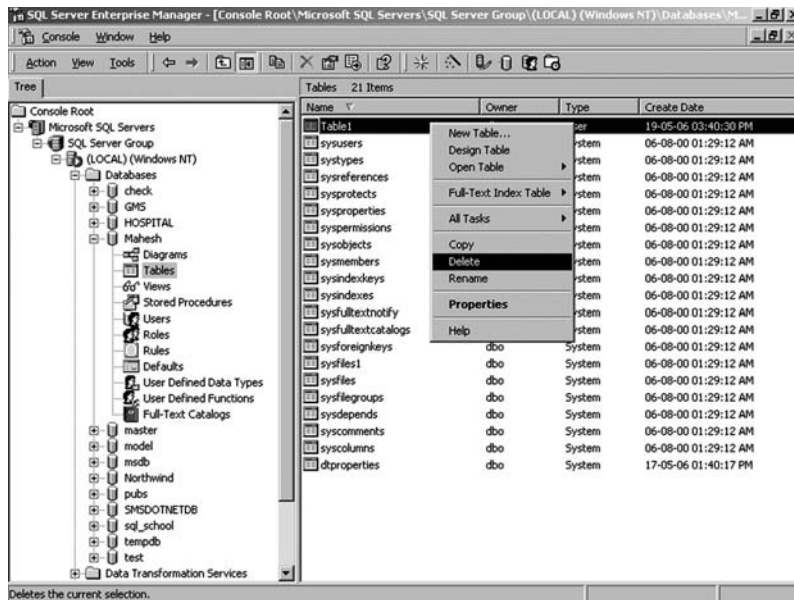


FIGURE 20.17

Click the **Drop All** button to drop the selected table.

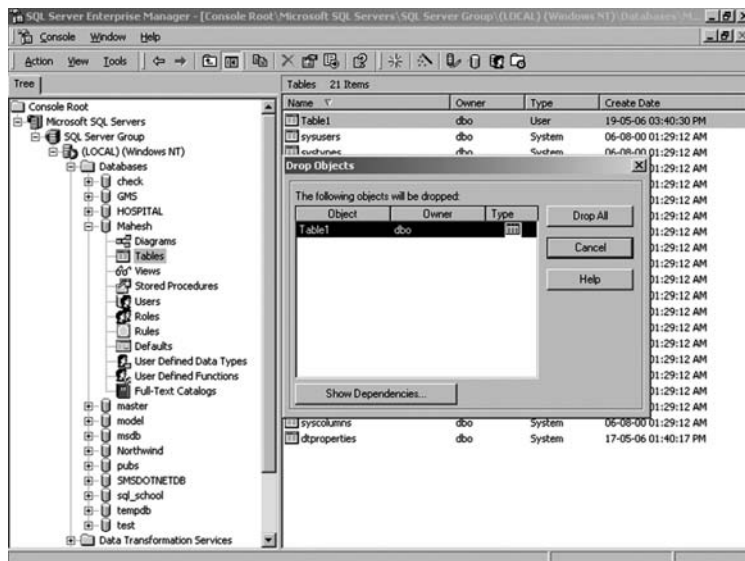


FIGURE 20.18

20.8.4 Opening a Table

There are three ways to view the contents of a table:

1. You can view all records of the table by selecting the option **Return all rows**.

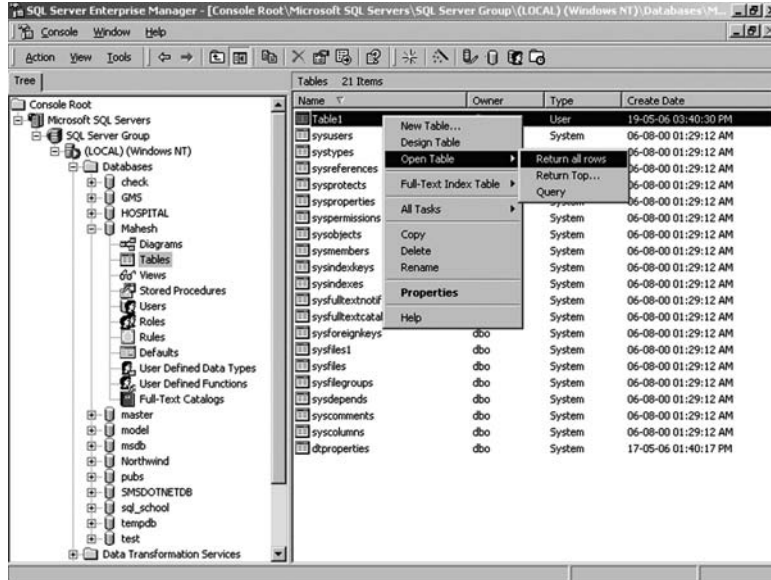


FIGURE 20.19

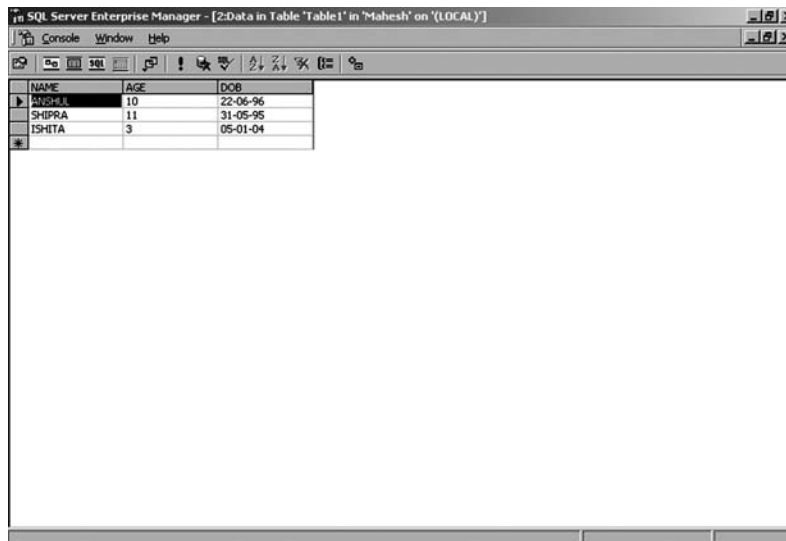


FIGURE 20.20

2. You can view as many records from the top as you need, by specifying the number of records. To do this select the option **Return Top**.

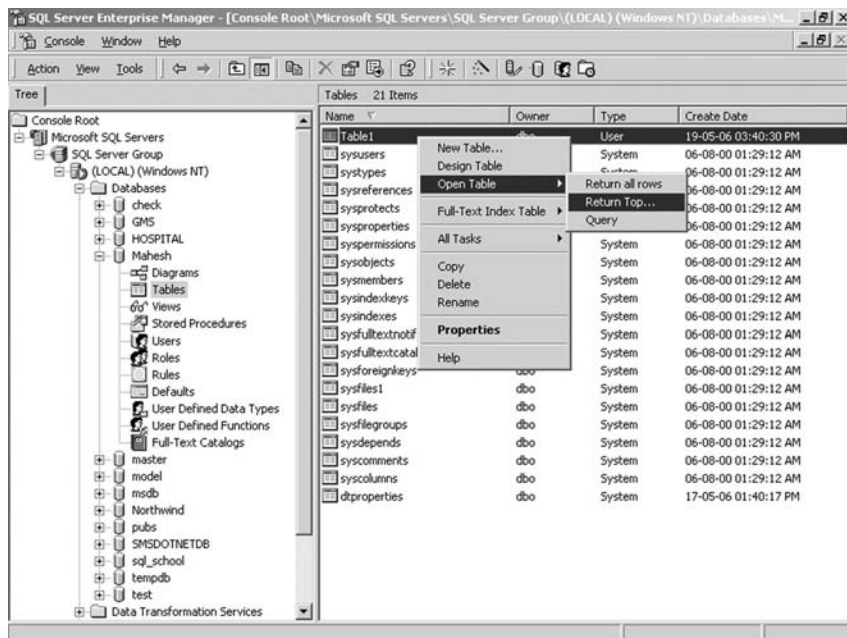


FIGURE 20.21

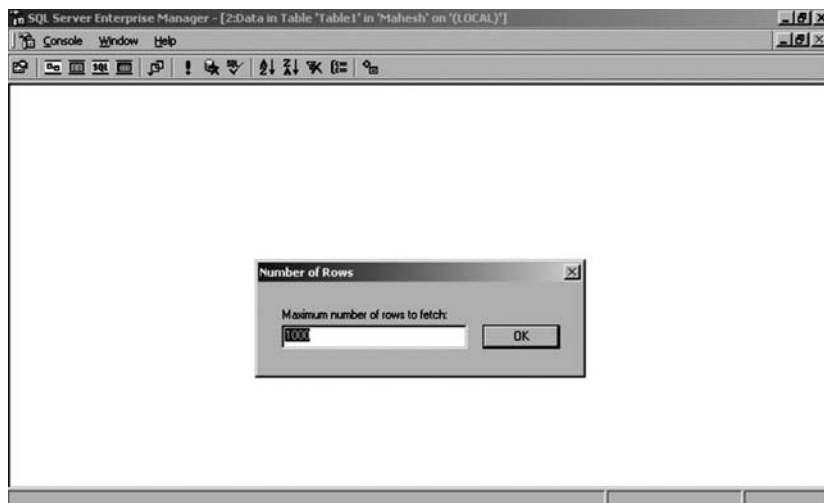


FIGURE 20.22

3. Selected records can be viewed using the **Query** option.

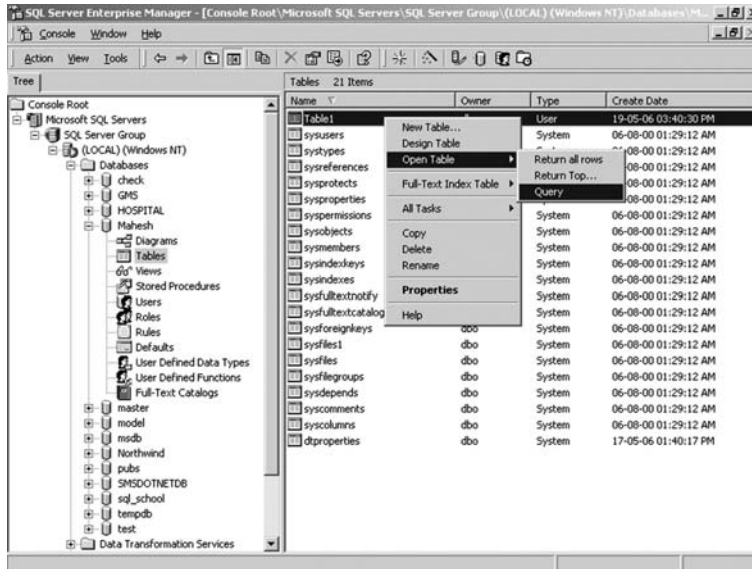


FIGURE 20.23

Write the query in the given box. To run the query press the F5 key or click the **Run** button.

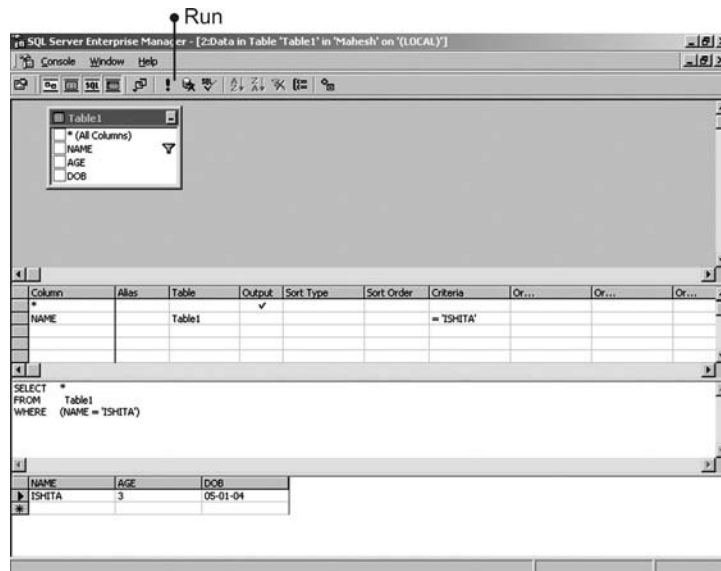


FIGURE 20.24

20.9 QUERY ANALYZER

SQL Query Analyzer is a graphical user interface for designing and testing SQL statements. SQL Query Analyzer offers:

- A free-form text editor for SQL statements.
- Color-coding of SQL syntax to improve the readability of complex statements.
- Object browser and object search tools for easily finding objects in a database.
- Templates used to speed development of the SQL statements for creating SQL Server objects. Templates are scripting files that include the basic structure of the SQL statements needed to create objects in a database.
- Results are shown in either a grid or a free-form text window.

20.10 HOW TO USE QUERY ANALYZER

Go to Microsoft **SQL Server** → **Query Analyzer** and give the Server name to connect with (SQL Query Analyzer can also be called from **SQL Server Enterprise Manager** → **Tools**). Now select the database.

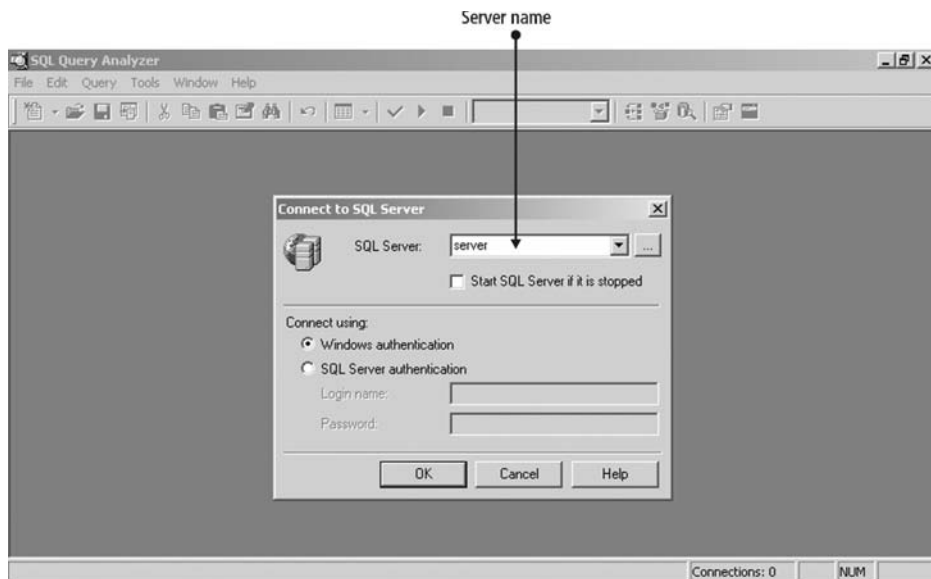


FIGURE 20.25

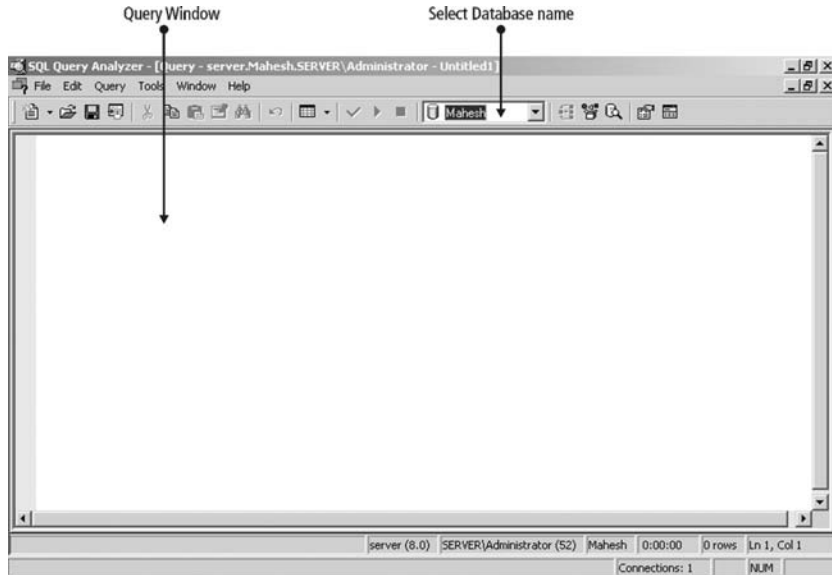


FIGURE 20.26

Write the query in the query window and press the F5 button to run the query.

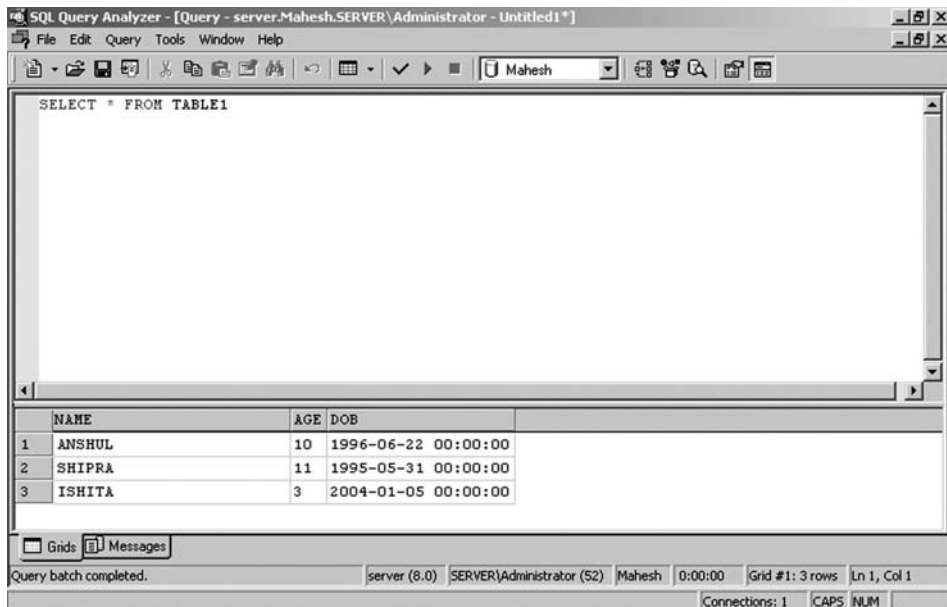


FIGURE 20.27

20.11 GENERATING AN SQL SCRIPT

A SQL script can be generated to create tables from one database to another database. To generate a script, take the following steps:

1. Go to the **Tools** menu and select the option **Generate SQL Script...**

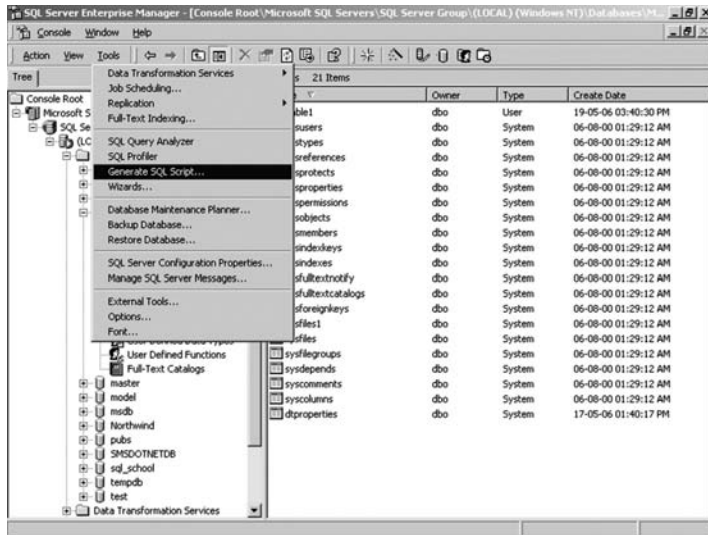


FIGURE 20.28

2. In the next screen click the **Show All** button to display all the tables of the database **Mahesh** in the object list.

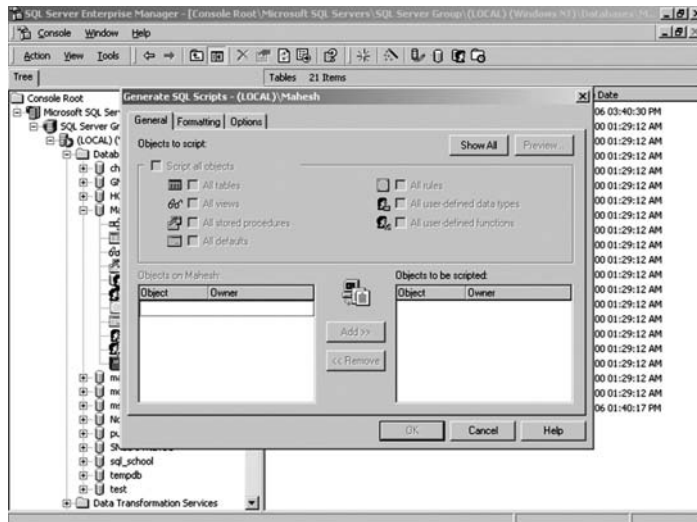


FIGURE 20.29

3. Select the table from left side of the object list for which you want to create the script and add it to the right side of the object list. You can create a script for a single table or for all tables of the database.

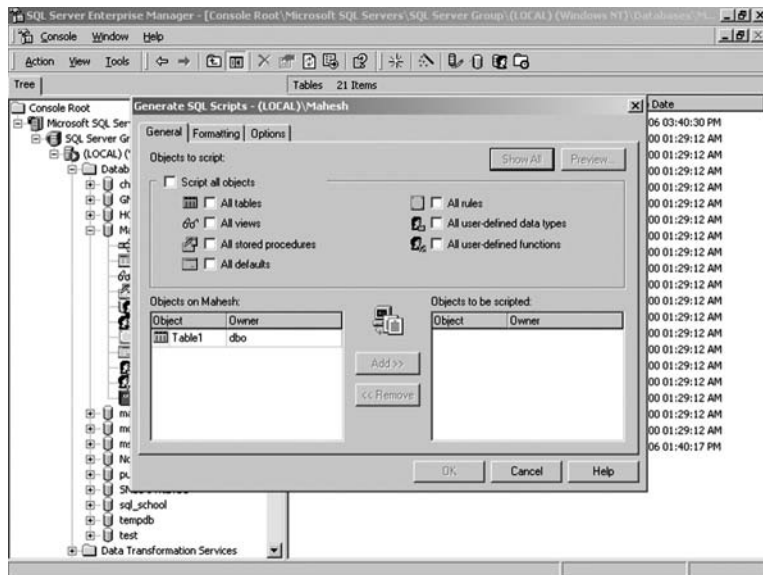


FIGURE 20.30

4. If you want to add all the tables click the **All Tables** checkbox.

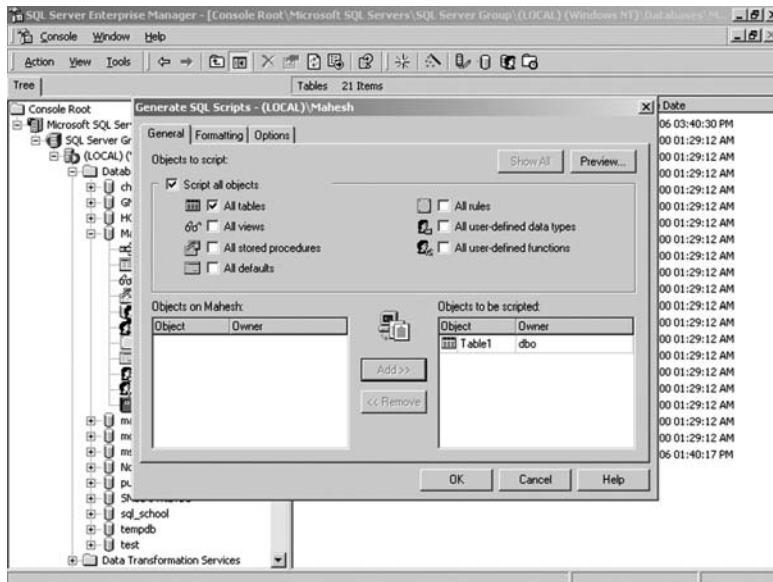


FIGURE 20.31

- Go to the **Formatting** tab and click the first checkbox **Generate the CREATE <object> command for each object**, if you want to generate only a create table script. Click the second checkbox **Generate the DROP <object> command for each object**, if you want to drop the table first (if it exists), then create another table with the same name.

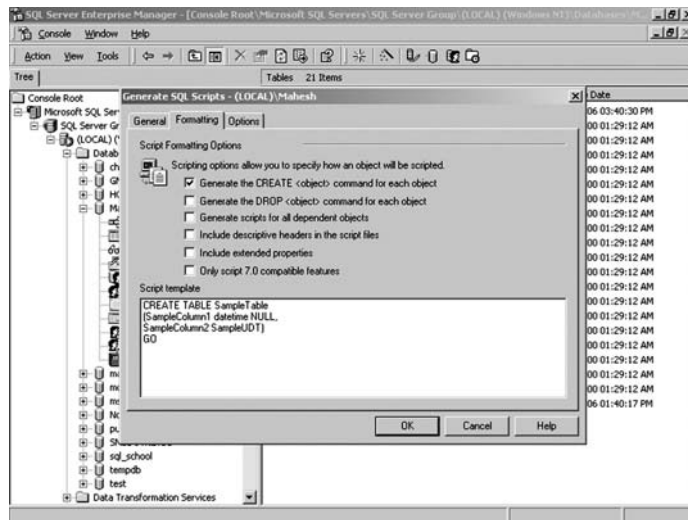


FIGURE 20.32

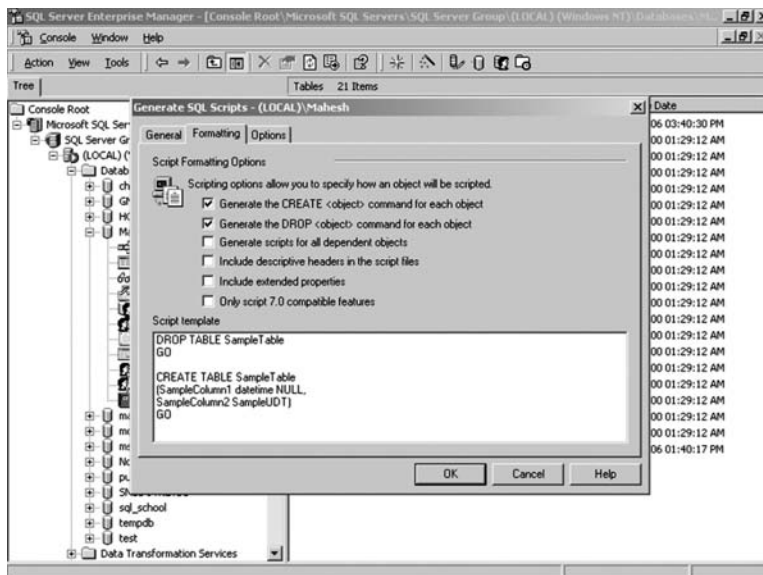


FIGURE 20.33

6. Click the **OK** button to continue.
7. Give a filename to save the script and click the **Save** button.

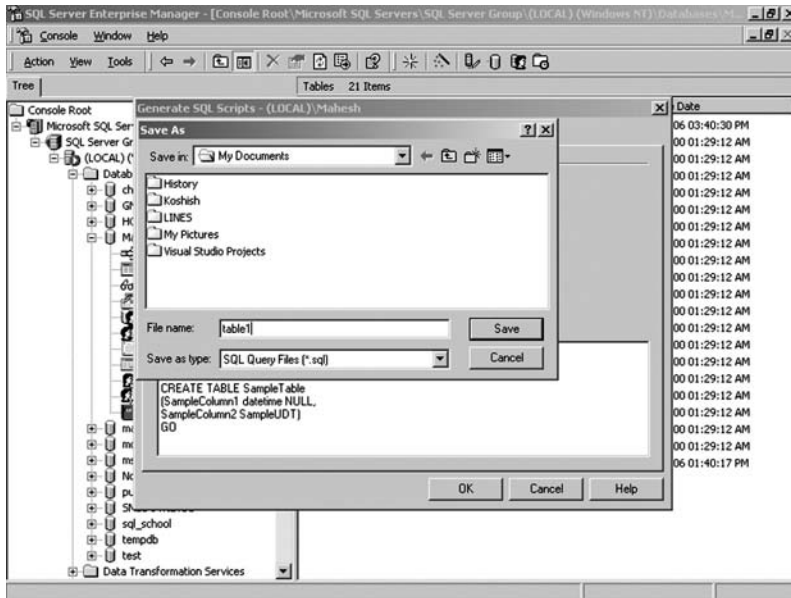


FIGURE 20.34

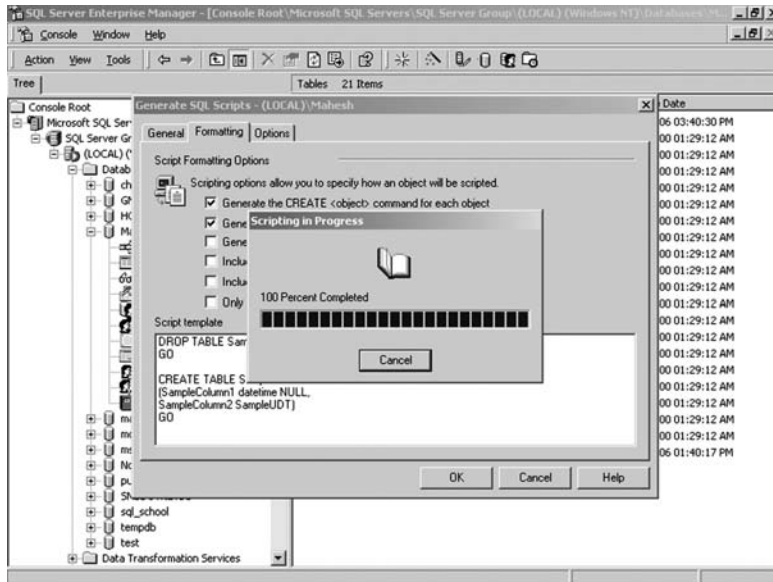


FIGURE 20.35



FIGURE 20.36

20.12 HOW TO USE THE SCRIPT

1. Go to Microsoft **SQL Server** → **Query Analyzer** and give the server name to connect with. Now select the database in which you want to run the script file to create the tables.
2. Go to **File** → **Open** menu and select the script file which you want to run.

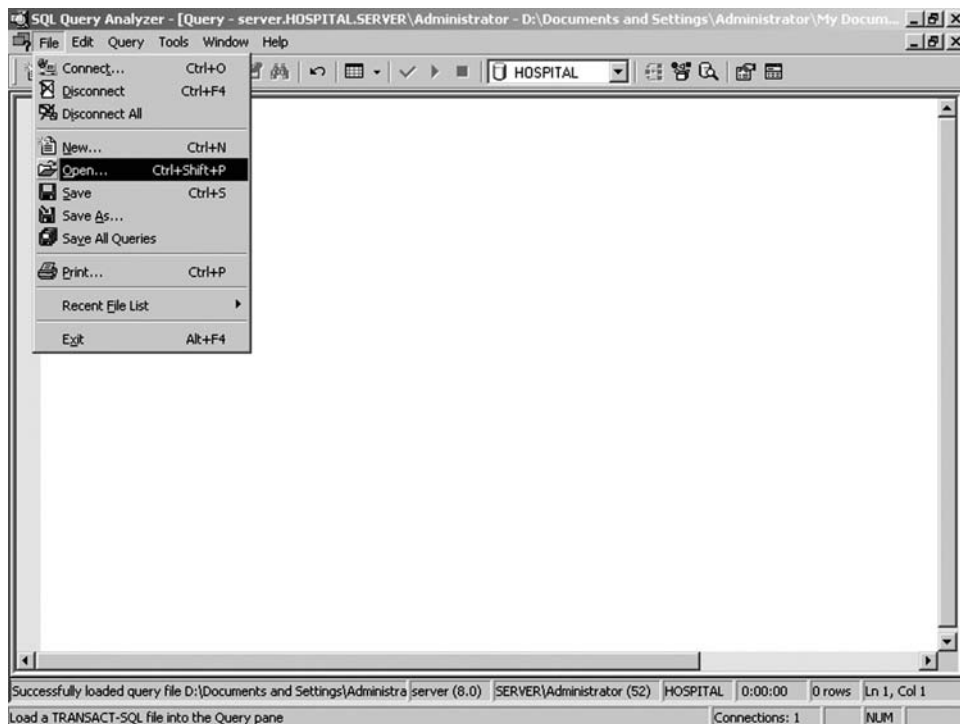


FIGURE 20.37

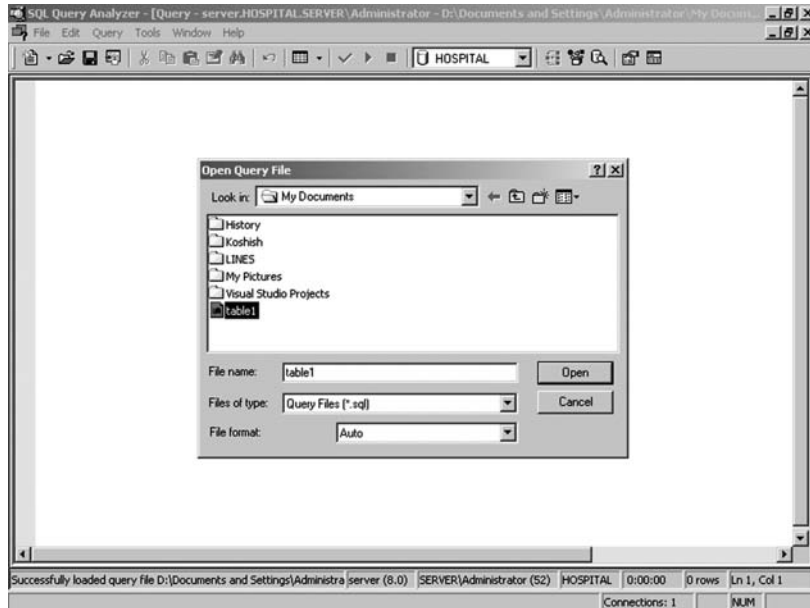


FIGURE 20.38

3. Press the F5 button to execute the query.

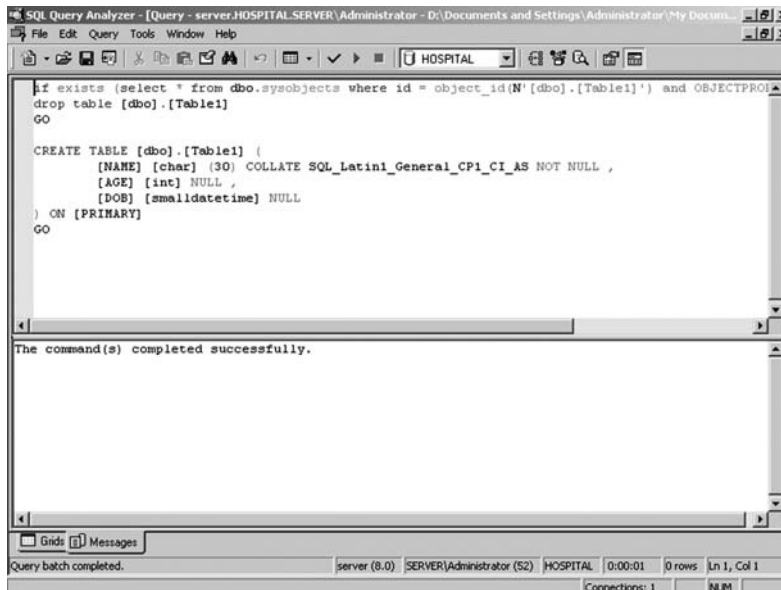


FIGURE 20.39

20.13 ATTACHING A DATABASE

Specify the name of the MDF (**master data file**) of the database to attach. Microsoft SQL Server cannot attach a database if more than 16 files are specified. To attach a database file with SQL Server take the following steps:

Right-click **Databases** and go to **All Tasks** and select the **Attach Database** option.

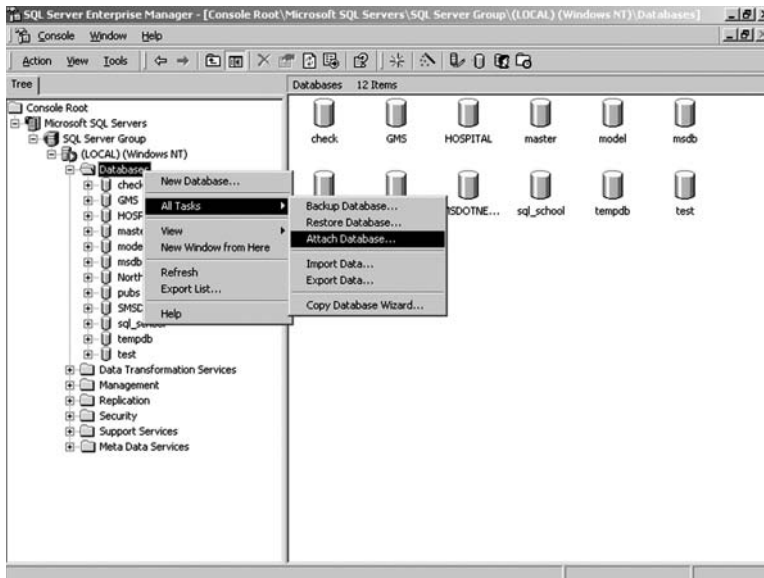


FIGURE 20.40

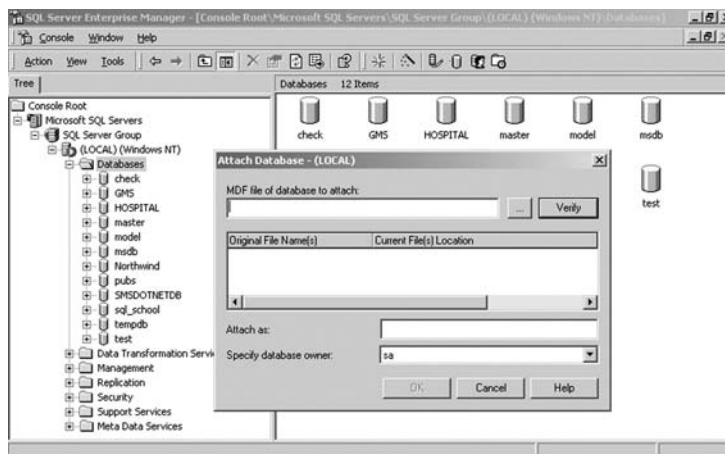


FIGURE 20.41

Click the Browse(---) button to search for the MDF file of the database to attach. Select the MDF file and click the **OK** button.

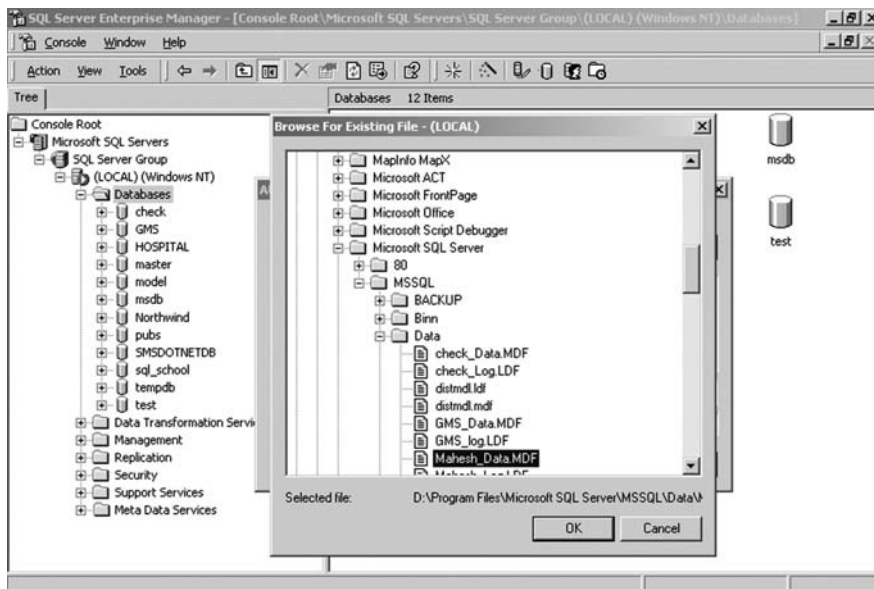


FIGURE 20.42

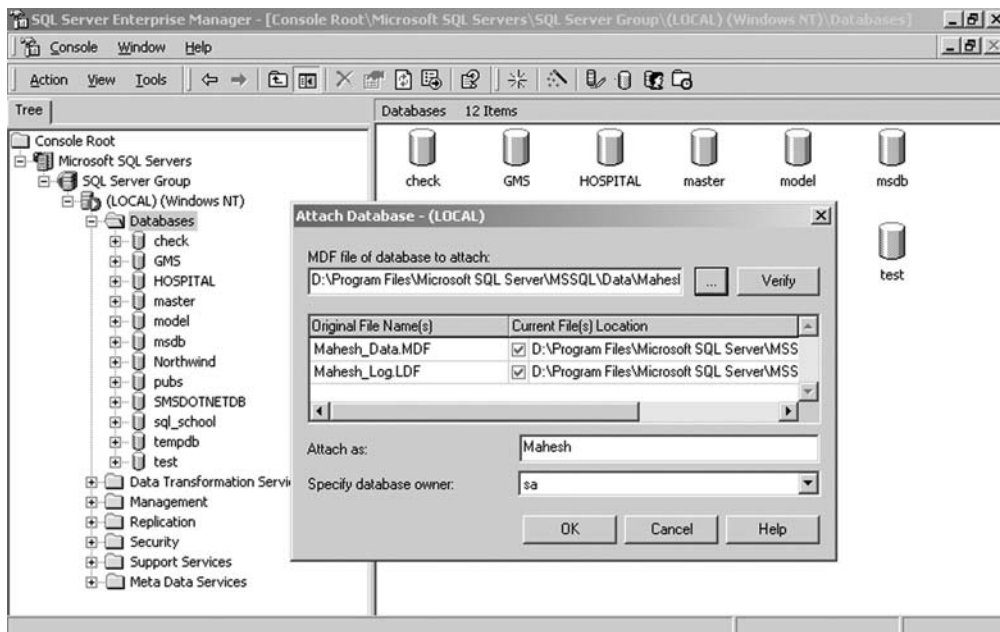


FIGURE 20.43

Click the **Verify** button to check that the selected MDF file is correct. In the **Original File Name(s)** column there is a list of all files in the database to attach. This includes data files and log files. In the **Current File(s) Location** column there is the path of all files. The current location of the MDF file must be in the column for **Attach** to work. If the SQL Server cannot find the files in the specified location, the **Attach** process fails. For example, if you have changed the default location of the file before you detached it, you must specify the current location for **Attach** to be successful. In the **Attach as** box specify the name for the database you are attaching. The database name should not match any existing database names. In the **Specify database owner** box, specify the database owner name.

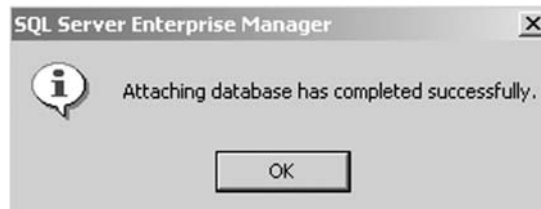


FIGURE 20.44

20.14 DETACHING A DATABASE

Keep the following guidelines in mind when detaching a database.

1. Stop connection

Stop if there is any connection to the selected database. You cannot detach a database while users are connected.

2. Database being replicated

See if the database is being replicated. You cannot detach a database while it is being replicated.

3. Status

View the status of the database. This will tell you whether the database is ready to be detached or not, based on the criteria in the previous options.

Now take the following steps;

Right-click **Databases** and go to **All Tasks** and select the **Detach Database** option.

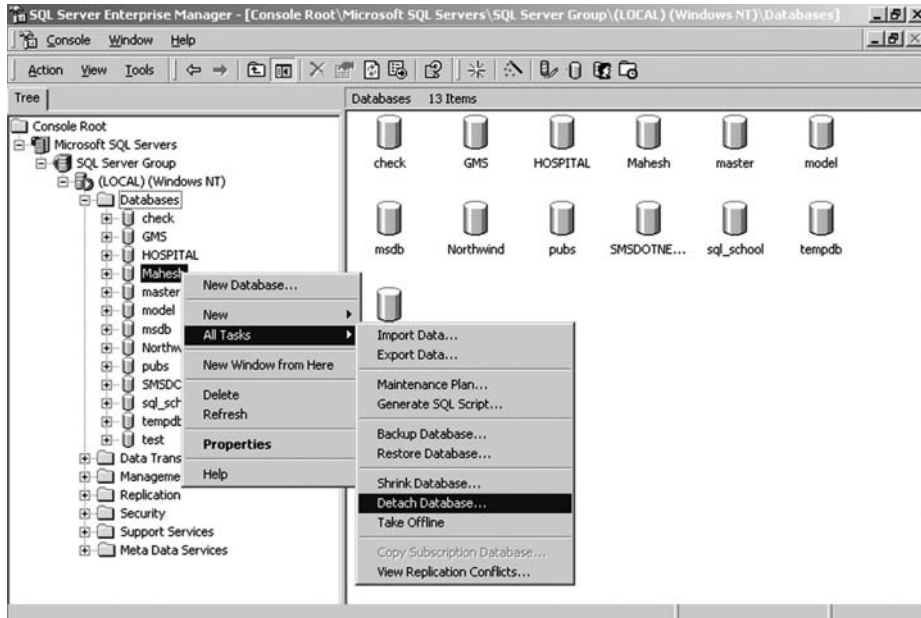


FIGURE 20.45

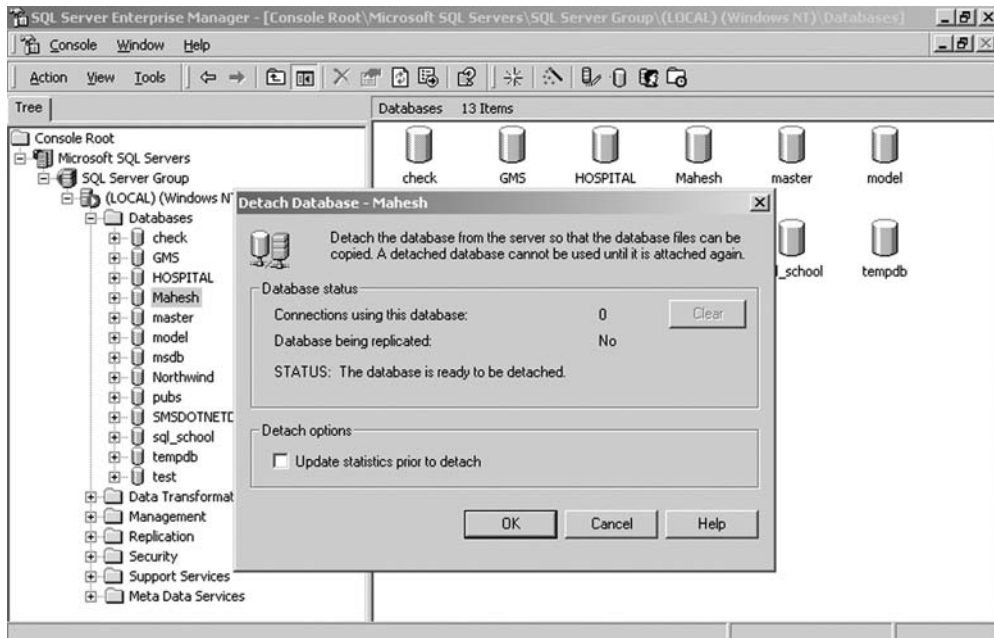


FIGURE 20.46

Check database status and click the **OK** button.

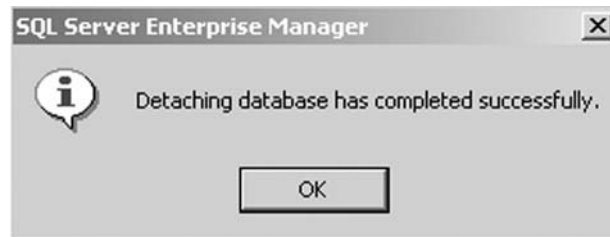


FIGURE 20.47

20.15 COPY DATABASE WIZARD

Users can run the **Copy Database Wizard** to copy or move databases and their objects to another server.

Go to **Microsoft SQL Servers → SQL Server Group → Server → Databases**. Right-click the **databases** and select **All Tasks → Copy Database wizard**. Follow the instructions that comes on each screen and complete the process.

20.16 IMPORTING AND EXPORTING A DATABASE

Microsoft SQL Server 2000 has several components that support importing and exporting data.

20.16.1 Data Transformation Services

Data Transformation Services (DTS) can be used to import and export data between two different OLEDB and ODBC data sources. A single DTS package can cover multiple tables. DTS packages are not limited to transferring data straight from one table to another; the package can specify a query as the source of the data. Take the following steps to Import or Export data:

Step 1

Right-click the database in which you want to import or export the tables of other databases. If you want to import the tables select the option **All Tasks → Import Data** and if you want to export the tables in another database select the option **All Tasks → Export Data**. Follow the necessary steps.

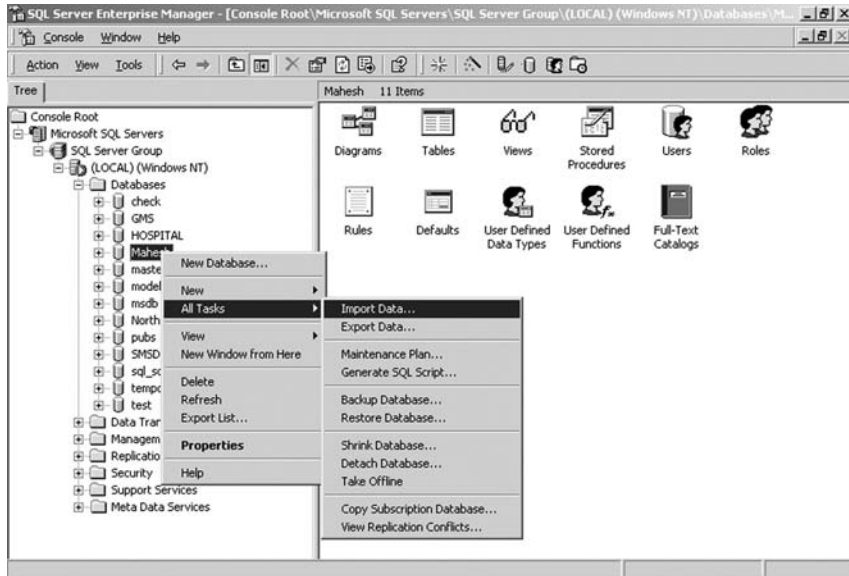


FIGURE 20.48

Step 2

Click the **Next** button.



FIGURE 20.49

Step 3

Select the data source from the data source list.

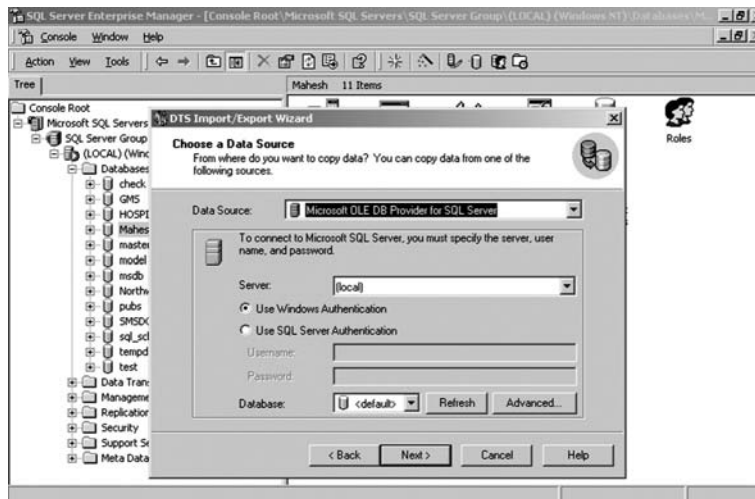


FIGURE 20.50

Step 4

If you want to import the tables of an MS Access database into the SQL Server database, select **Microsoft Access** from the data source list and give the path of the .mdb file in the filename box.

Click the **Next** button to continue.

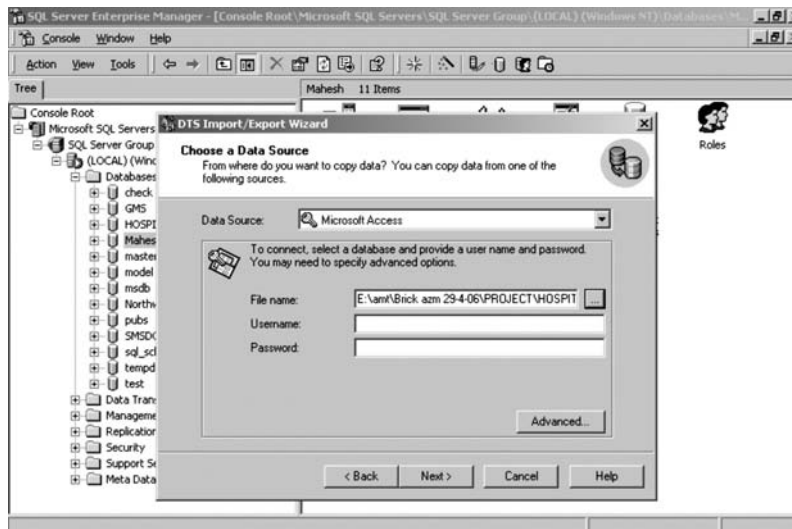


FIGURE 20.51

Step 5

Select where you want to import the tables.
 Select the data source from the destination list.
 Select the server name from the server list.
 Select the database from the database list.
 Click the **Next** button.

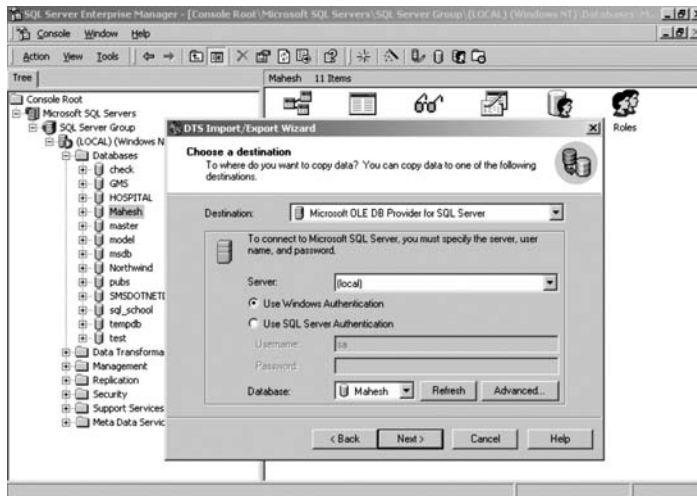


FIGURE 20.52

Step 6

Click the **Next** button to continue.

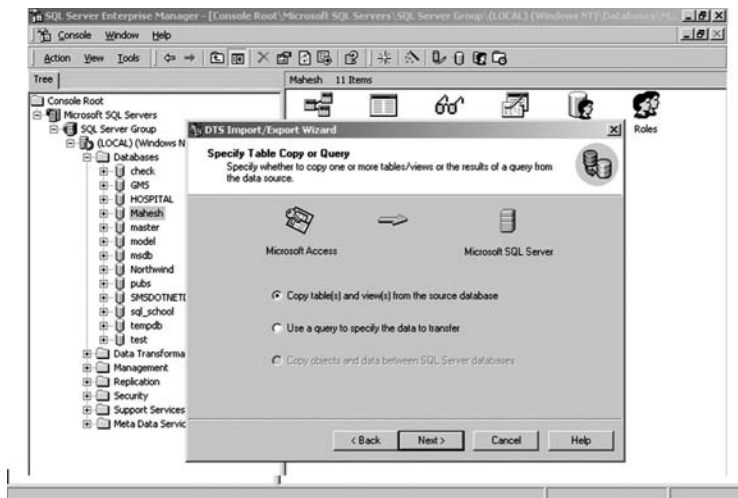


FIGURE 20.53

Step 7

Select table name in the Source column which you want to import. If you want to import all the tables, click the **Select All** button.

Click the **Next** button to continue.

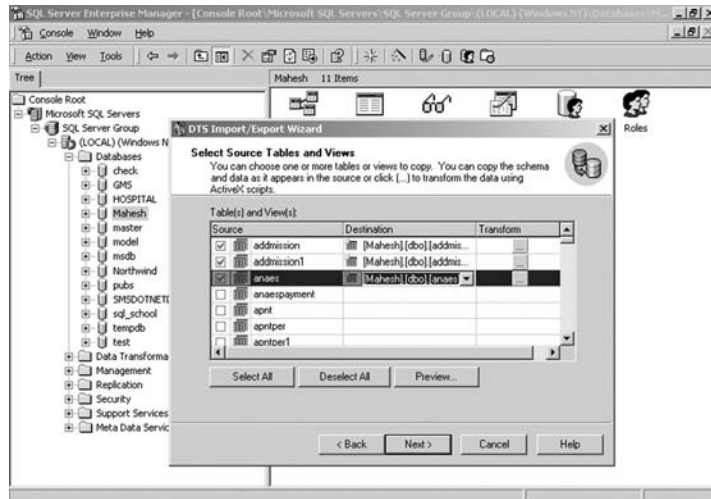


FIGURE 20.54

Step 8

Click the **Next** button to continue.

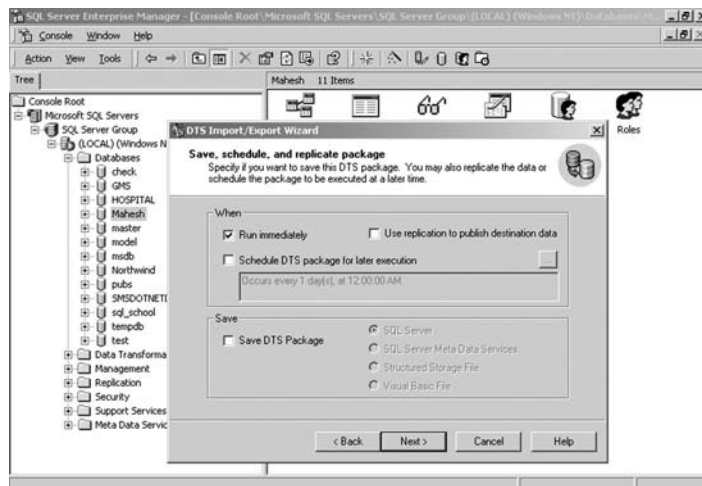


FIGURE 20.55

Step 9

Click the **Finish** button to finish the import process.

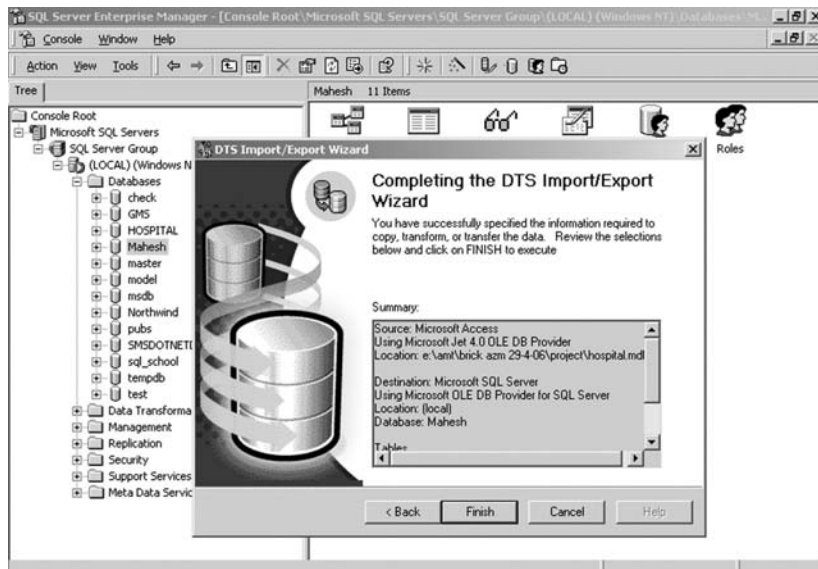


FIGURE 20.56

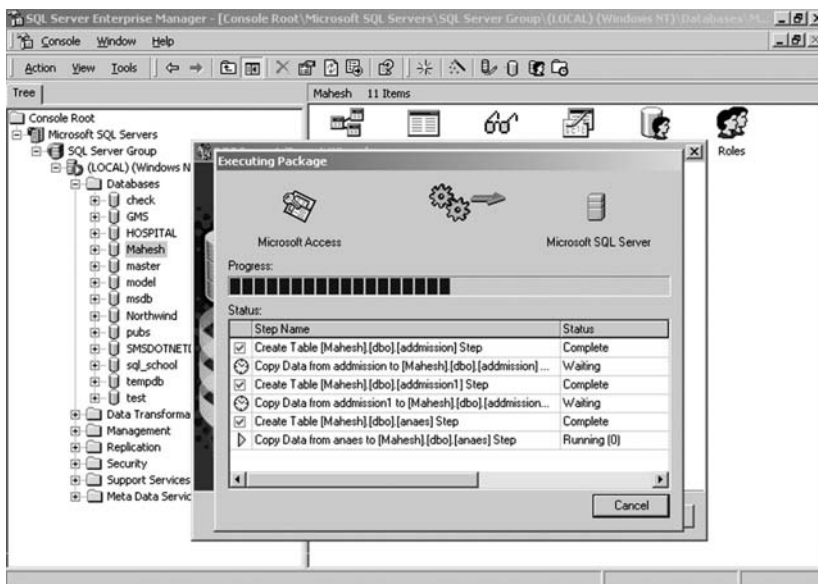


FIGURE 20.57

20.17 SQL SERVER SERVICE MANAGER

You can Start, Stop, or Pause the currently running database.

Go to **Programs → Microsoft SQL Server → Service Manager**.

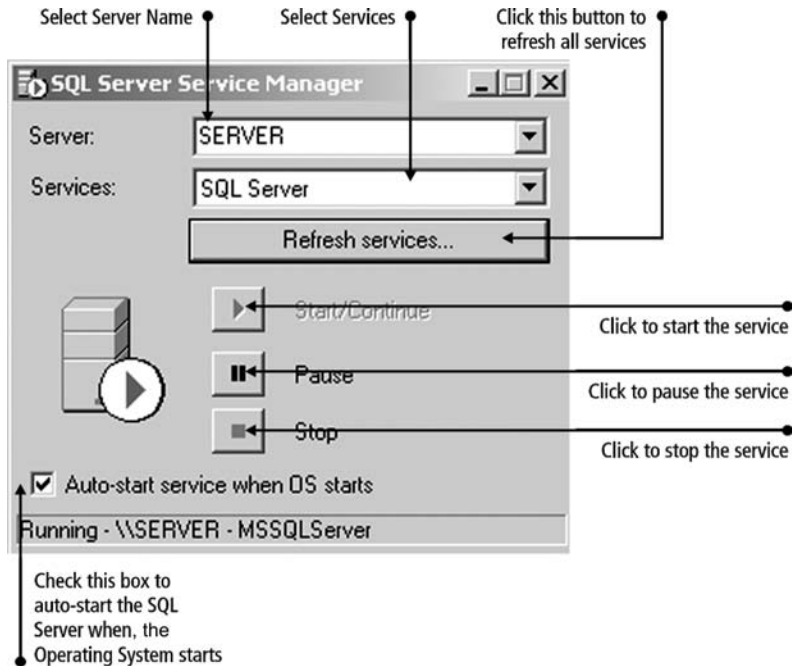


FIGURE 20.58

In the server list, a list of active SQL Server database servers on the network is displayed. This list only displays instances of SQL Servers on Windows NT. This functionality is not available on computers running Microsoft Windows 95 or Windows 98.

20.17.1 Authentication

Specify the type of authentication to use when connecting to the database server.

Windows NT Authentication	Specifies that the SQL Server will use the Windows NT user information to validate the user. This option is only available when connecting to an instance of SQL Server on Windows NT. The client needs to be part of a Windows NT domain or workgroup. The user needs to be validated as a Windows NT user before access is granted.
---------------------------	---

SQL Server Authentication	Specifies the use of standard SQL Server security validation. This is the default and only available option for an instance of SQL Server on the Windows 95 or Windows 98 operating system. It is optional for an instance of SQL Server on Windows NT. The log-in must be added to the SQL Server before a user can log in.
Login name	Specify a log-in name.
Password	Specify the password for the log-in name.

20.17.2 Creating a Connection with Visual Basic

Follow these steps to connect the MDI form of Visual Basic with the SQL Server 2000 database.

1. Open the Design View of the MDI form.
2. Add ADODC control in the Toolbox and draw it in MDI form.

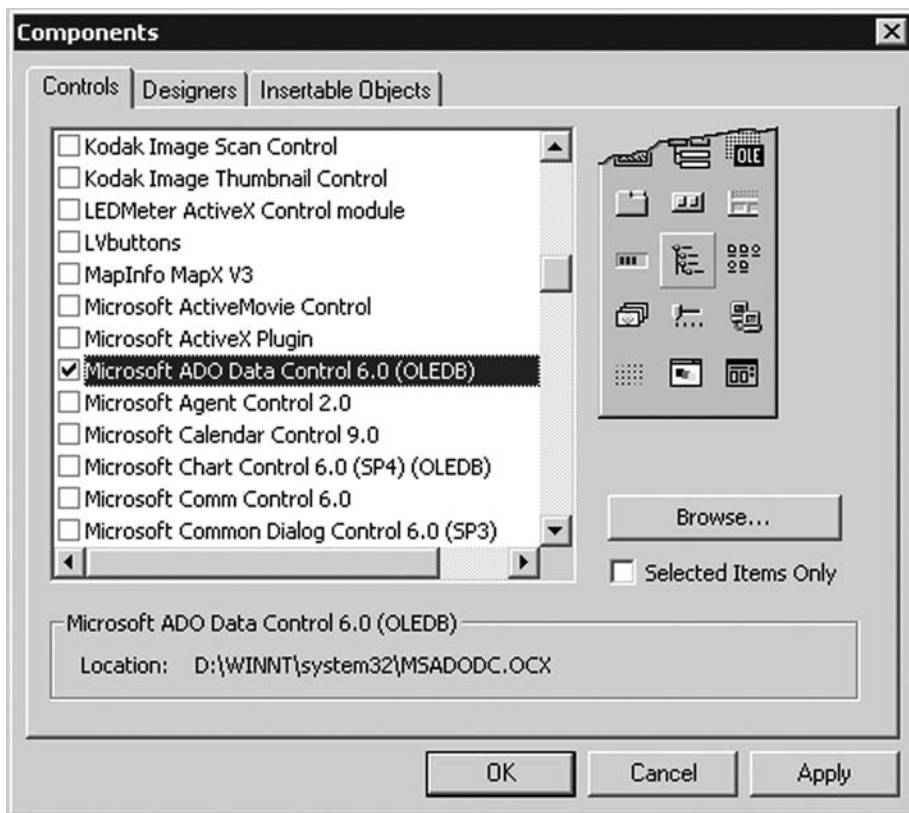


FIGURE 20.59

3. Right-click ADODC control and go to the properties option. Select the Use Connection String option and click the **Build** button.
4. From the list of OLEDB providers, select Microsoft OLEDB Provider for SQL Server and click the **Next** button.

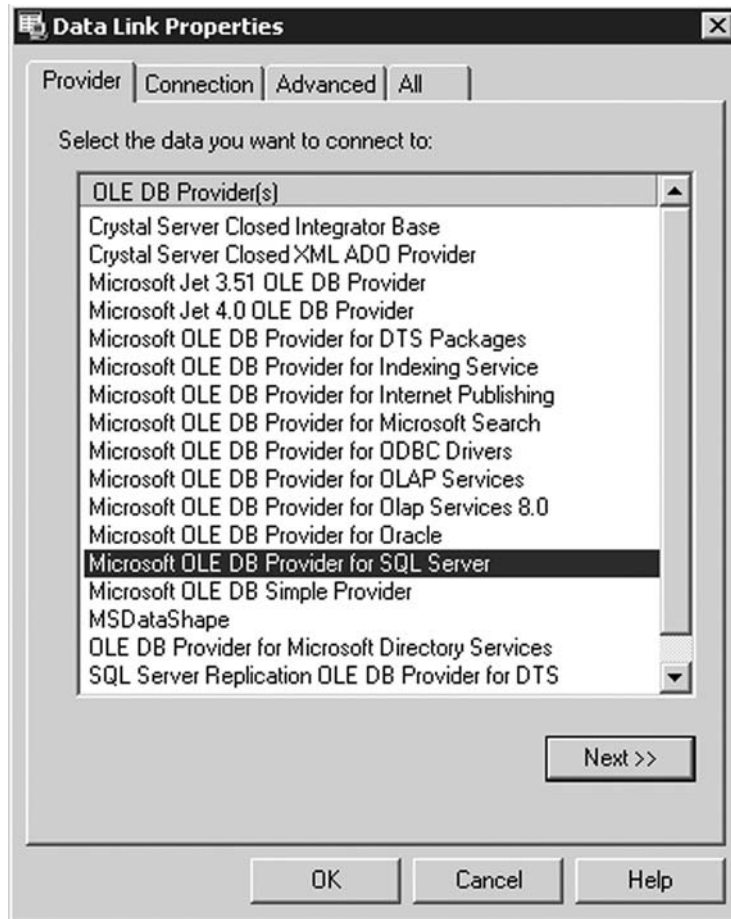


FIGURE 20.60

5. In the Connection window, select the server name from the server list. Select the option Windows NT Integrated security or give the username and password. Now select the database name and click the **Test Connection** button. A message box with the message 'Test connection succeeded' will appear. Click the **OK** button and then click the **OK** button again.

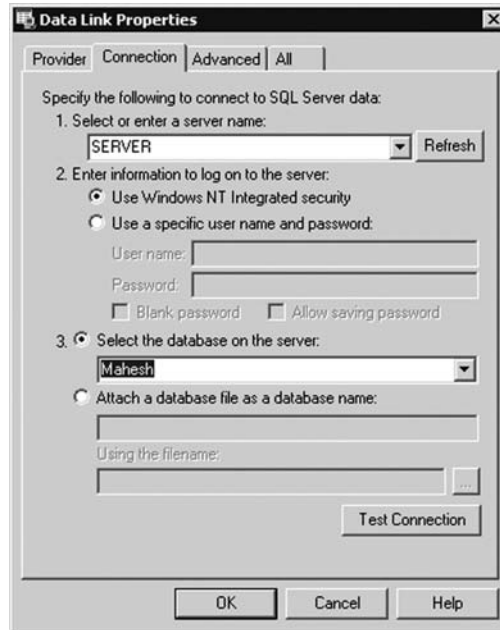


FIGURE 20.61

6. Copy the string that comes in the Use Connection String box, and click the OK button.

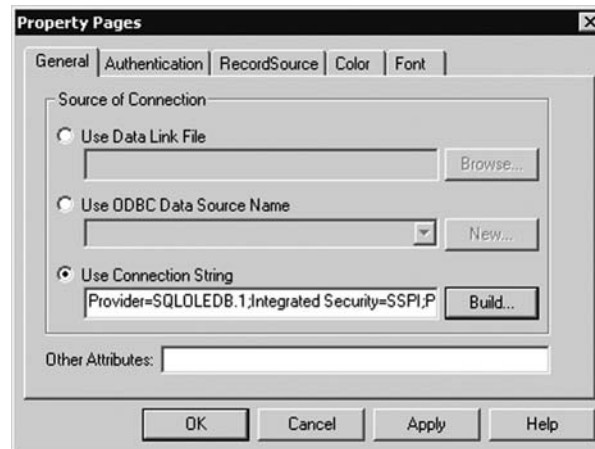


FIGURE 20.62

7. Go to the MDI form's Form load event and paste the string.
8. Further processes are just like MS Access or Oracle connectivity.

Chapter 21

PROGRAMMING IN VISUAL BASIC WITH MS ACCESS 2000

An application can be developed with a or without a database. If you have to develop an application that does not save the input data, you don't need to use a database, but if you have to save the input data, you must use a database. There are different types of databases. The database selection depends upon two things—first is the amount of input data and second is whether the application is single-user or multi-user, i.e., the application will run on a stand-alone system or it will run on a server with many clients. Each database is network compatible. If your application is small and only a few users will work on it, you should use MS Access 2000. If the data input and the number of users is much higher, use SQL Server 2000. But if the application requires a large amount of data flow and the number of users is also high, you should use Oracle 8/8i/9i/11i.

Now we are going to develop a new project in Visual Basic 6.0. The database we will use is MS Access. The details of the project are given below:

Name of Project: School

No. of forms: 3

The first form is the MDI form

The second form is for student record entry

The third form is for student report

MDI Form



FIGURE 21.1

To add a new MDI form in the project:

Go to the Menu bar and select **Project** → **Add MDI Form** option.

Set the property of the MDI form as given:

Caption: Student

WindowState: 2- Maximized

Use the Menu Editor to design the Menu bar on the MDI Form.

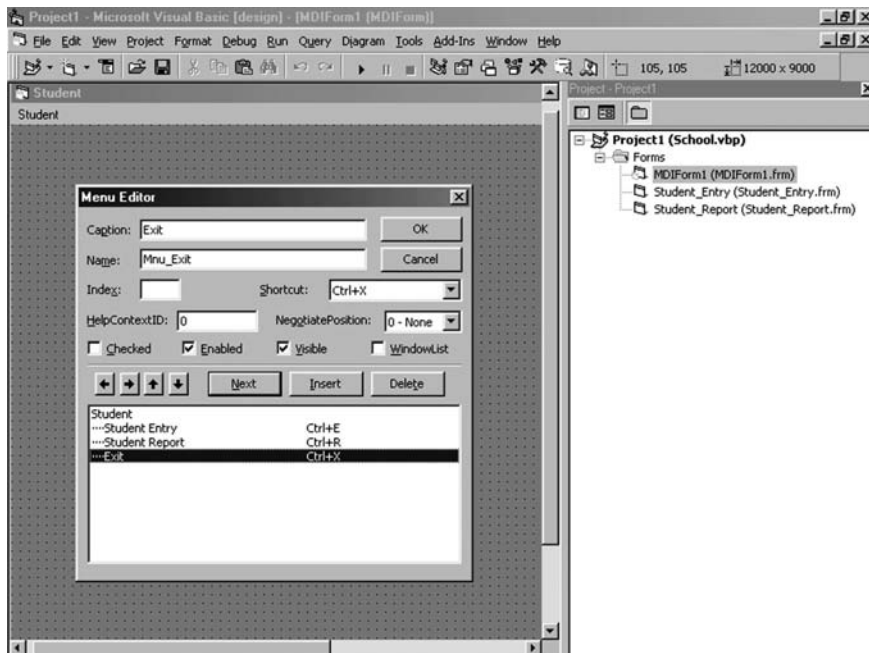


FIGURE 21.2

Go to the Menu bar **Project** → **Project1 Properties** and select the Startup Object as the MDI Form.

21.1 SAVING PROJECTS AND FORMS

Make a separate folder in the drive to save projects and forms. In this folder make sub-folders. Save different types of files in different sub-folders.

Sub-folder Name	Description
Form	To save form files with the .frm extension.
Project	To save project files with the .vbp extension. Some temporary files with the extension.tmp are created automatically in that folder where you have saved the project files. Keep removing these files to save disk space. Although these files are automatically removed from your folder when you close the project, in the case of an improper Shut Down or an illegal operation these files may remain on your folder.
Report	To save your Data Report or Crystal Report files.
OCX	If you have added some OCX controls in your project, save all OCX files in this folder.
Image	To save image files.
Module	To save the module file with the .bas extension.

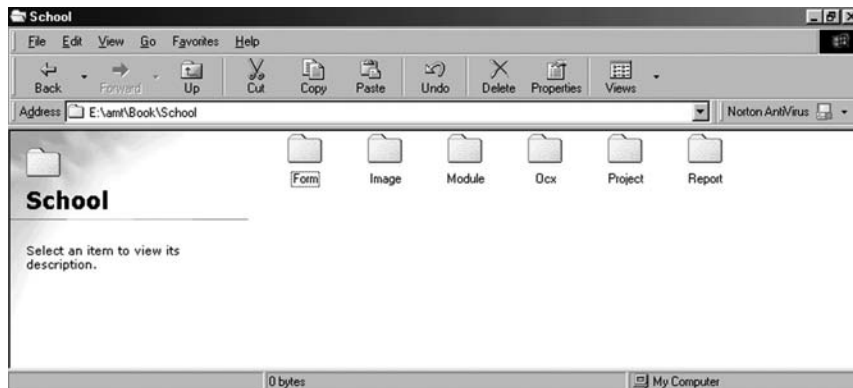


FIGURE 21.3

21.1.1 Design of Student Entry Form

Add a new form in the project and set the property of the newly added form, Form1, as given:

Caption: Student Entry
 BorderStyle: 1- Fixed Single
 MDIChild: True

Design the Student Entry form as shown in figure.

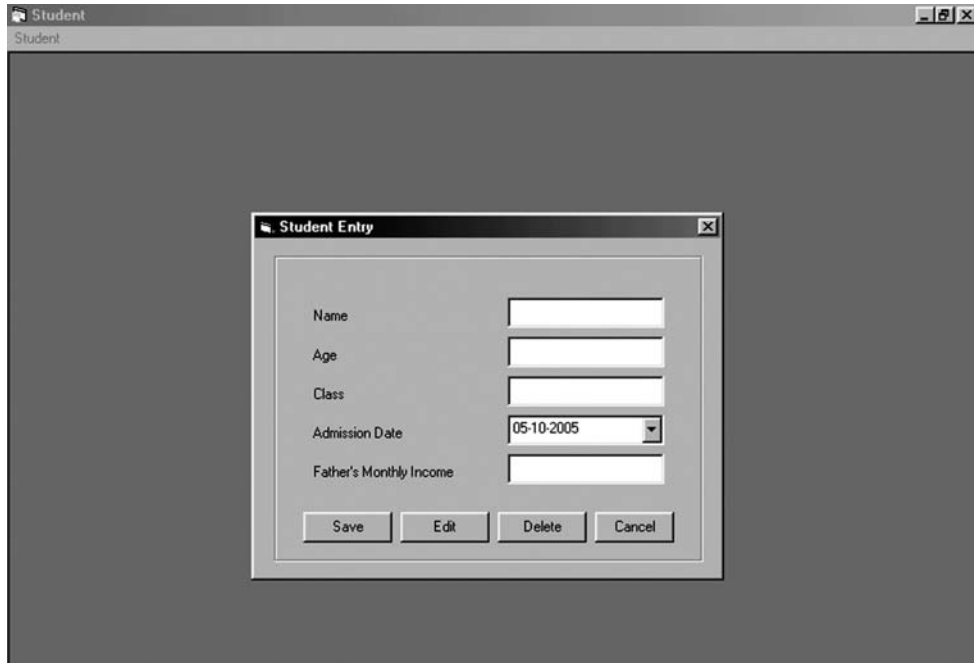


FIGURE 21.4

Add four text boxes in the form and change the name of text boxes as given:

Text Box	Name
Text1	Txtname
Text2	Txtage
Text3	Txtclass
Text4	Txtincome

Add five label controls in the form and change the caption properties of the labels as given:

Label	Caption
Label1	Name
Label2	Age
Label3	Class
Label4	Admission Date
Label5	Father's Monthly Income

Add a new control DTPicker in the project to select or enter the admission date. To enter the DTPicker control in the project, right-click Tool Box and go to components option. A list of controls will be populated. Scroll through the list and select Microsoft Windows Common Controls – 2 6.0 (SP4) and click **Apply** and then the **Close** button. You can see this in the figure. The control is added in the Tool Box.

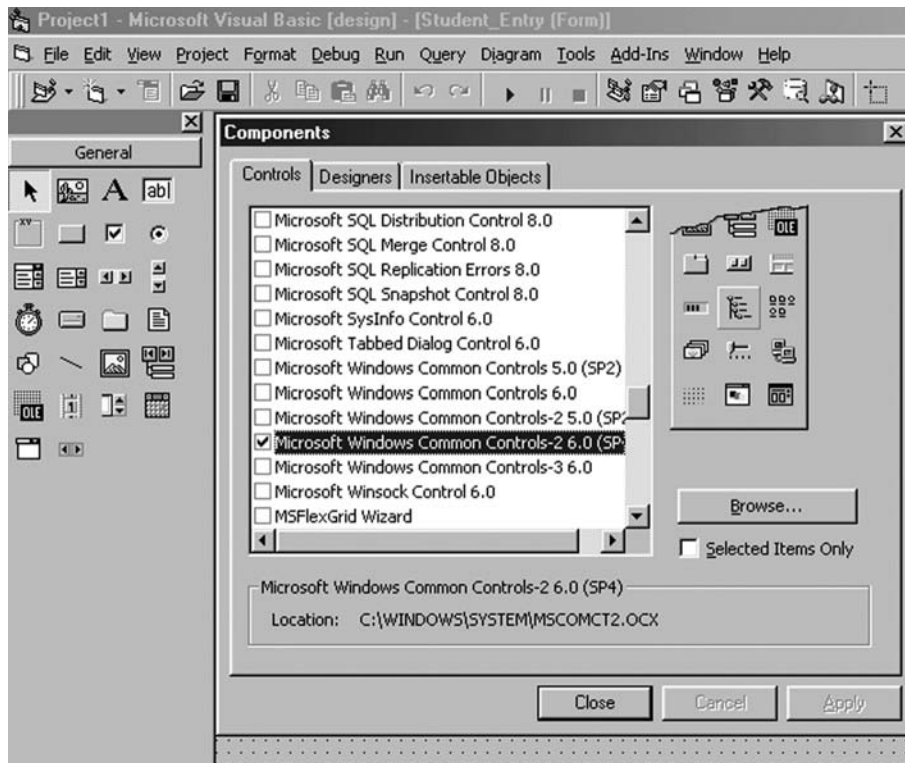


FIGURE 21.5

Now draw the control in the form.

Add five command buttons in the form. Change the Caption and Name property of these command buttons as follows:

Command Button	Name	Caption
Command 1	CmdNew	New
Command 2	CmdSave	Save [Change the Default property to True]
Command 3	CmdEdit	Edit
Command 4	CmdDelete	Delete
Command 5	CmdCancel	Cancel [Change the Cancel property to True]

If you set the Default property of a **Command** button to True, you can execute this **Command** button either with the mouse or by pressing the Enter Key, whereas if you set the Cancel property of a **Command** button to True, it can be executed by pressing the Esc key on your keyboard.

21.1.2 Design of Student Report Form

Add a new form in your project. To add a new form go to Menu Bar and select the option **Project** → **Add Form**.

FIGURE 21.6

Change the property of Form2 as given:

Name: Student_Report
Caption: Student Report
BorderStyle: 1- Fixed Single
MDIChild: True

Draw two DTPicker controls. First select Admission Date From and second select Admission Date To and rename them as DTPfrom and DTPto, respectively. Add two **Command** buttons. First to show all student records and second to show the admissions between two selected dates. Change the Caption of Command1 to All Record and rename it Cmdall. Change the Caption of Command2 to Admission Date wise and rename it Cmddatewise. Now Add a new control **MSFlexGrid** in the project. MSFlexGrid is used to display the data. To add this control right-click the Tool Box and select the components option. Select Microsoft FlexGrid Control 6.0 (SP3) and add it in the project.

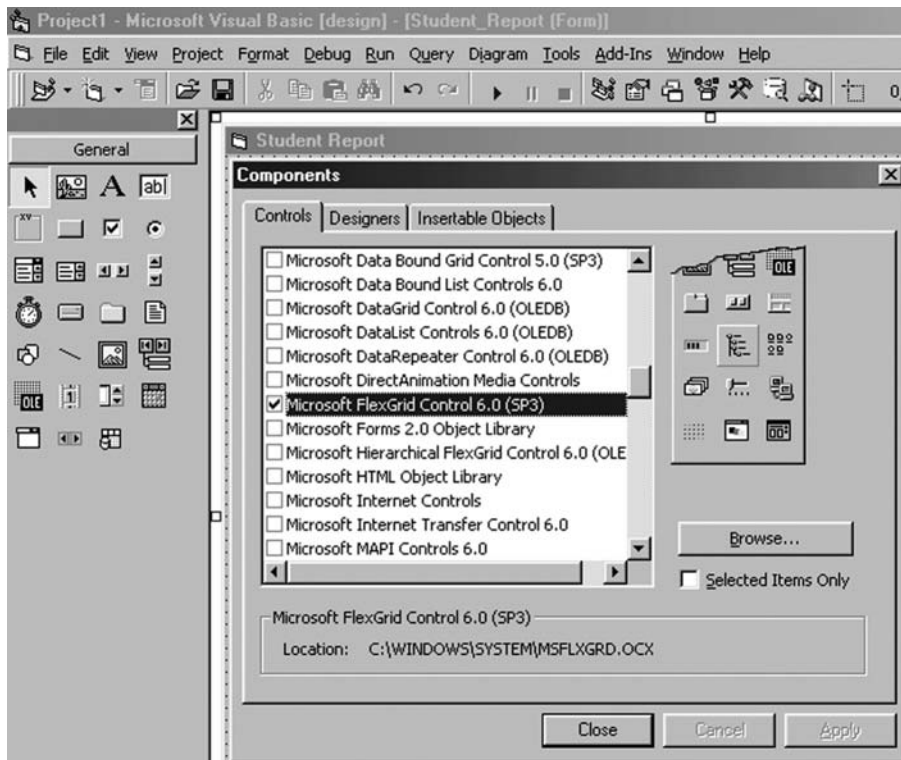


FIGURE 21.7

Draw the control in the form. To change the properties of the FlexGrid, right-click the control. A property page is displayed. In the General tab set the Cols property to 6 and the Allow User Resizing property to 1-Columns. By changing the Cols property to 6, we will have 6 columns in the FlexGrid and by changing the AllowUserResizing property to 1-Columns, we can resize the columns. There are 4 types of options for the AllowUserResizing property (see Figure 21.8).

Property Value	Description
0- None	Does not allow the user to resize columns or rows
1- Columns	Allows user to resize the columns
2- Rows	Allows user to resize the rows
3- Both	Allows user to resize the columns and rows

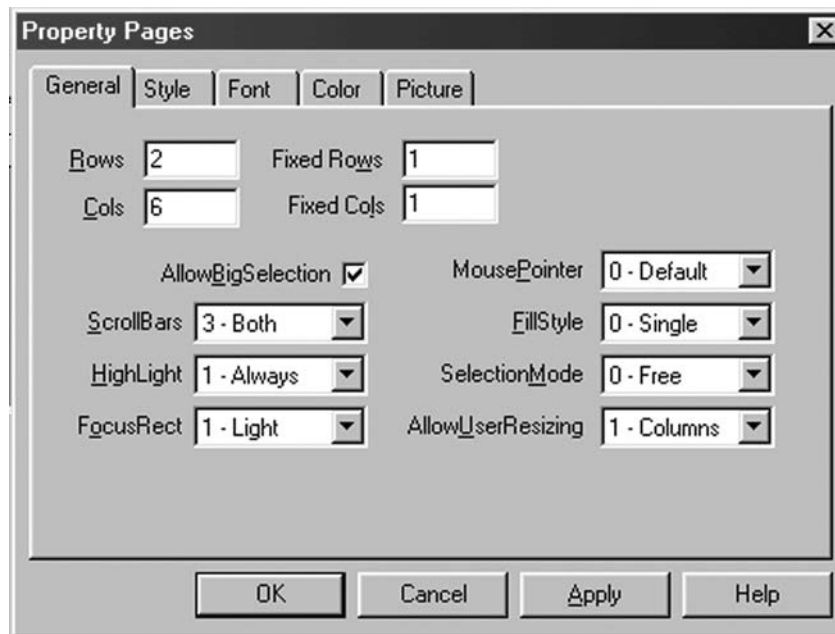


FIGURE 21.8

Now go to Style tab and in the Format box write the column names separated by the pipe (|) as shown.

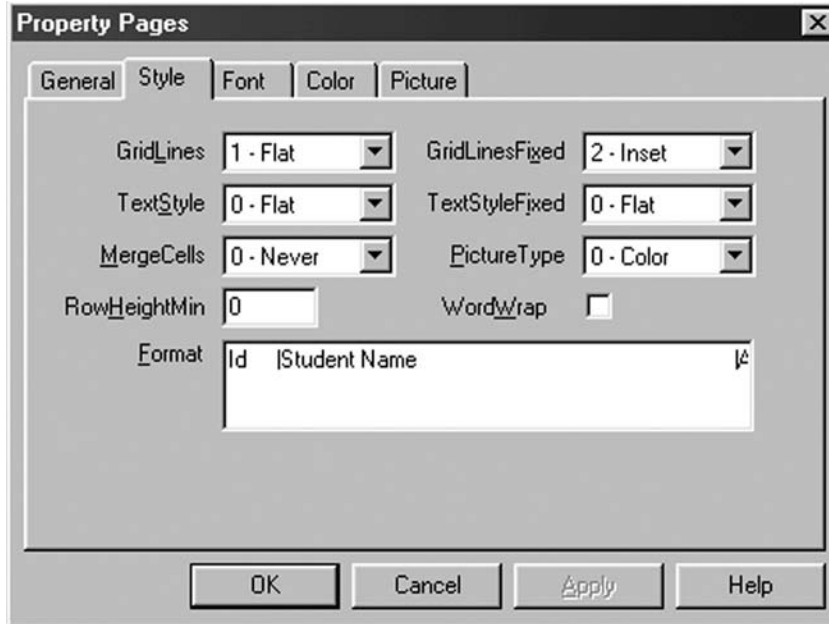


FIGURE 21.9

Example: Id | Student Name | Age | Class | Admission Date | Property page | Father's Monthly Income

21.2 DATABASE DESIGNING

Create a new database called SchoolData.mdb and save it in your project folder. Make a table called School.

Structure of the Table:

Field	Data type
ID	Number
NAME	Text
AGE	Number
STD_CLASS	Text
ADMISSION_DATE	Date/Time
FATHER_MONTHLY_INCOME	Currency

Set the field ID as the primary key. To set the primary key, right-click the ID column and select the Set as Primary Key option.

21.2.1 Modules

To add a module to the project, go to the Menu bar and select **Project → Add Module** option. Save this module in the module sub-folder which you have created already. The variables you declare or the functions you write in the module can be accessed globally in the entire project.

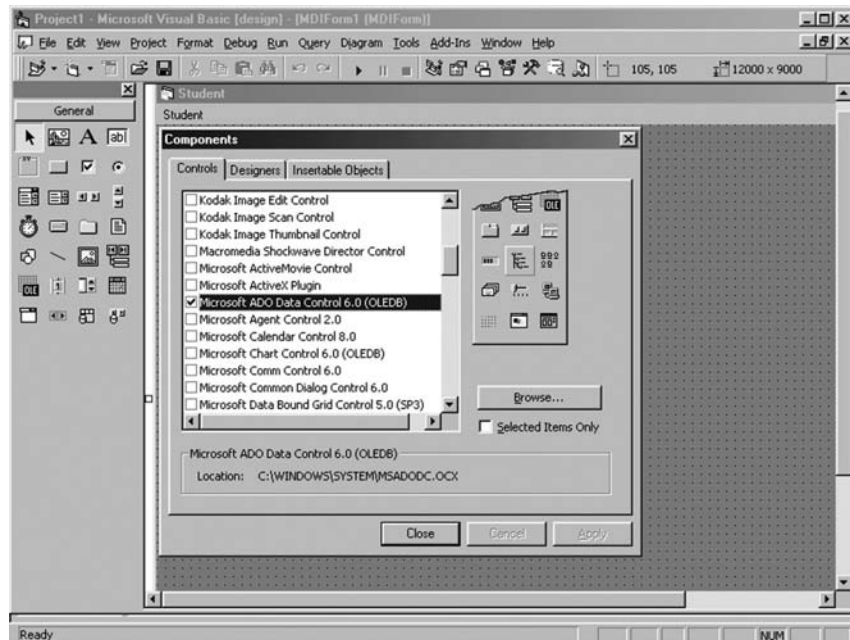


FIGURE 21.10

21.2.2 Database Connectivity

Follow the given steps to connect the MDI form with the MS Access 2000 database.

1. Open the Design View of the MDI form.
2. Right-click the Tool Box and select the components option.
3. From the list of the controls, select Microsoft ADO Data control 6.0 (OLEDB) and add it to your Project (see Figure 10.10).
4. Draw the **ADODC** control in the MDI form.
5. Right-click the ADODC control and select ADODC properties.

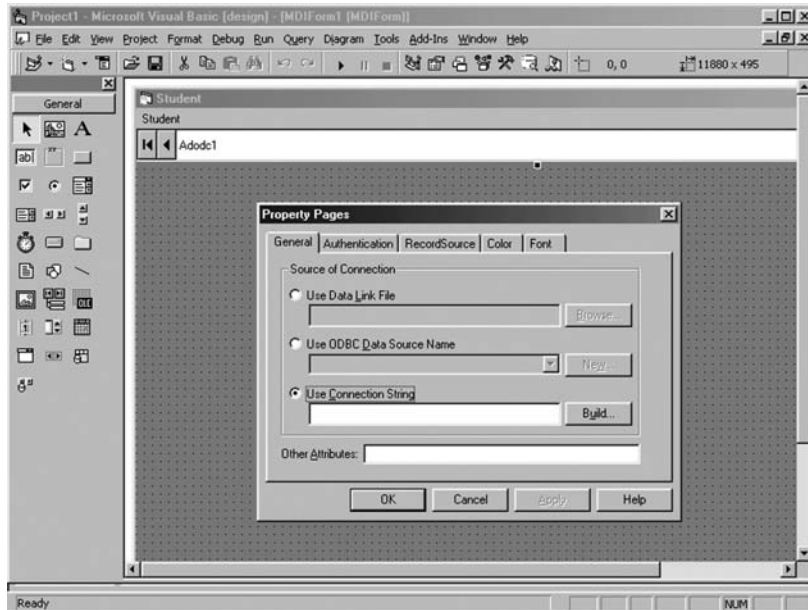


FIGURE 21.11

6. In the General properties page, select the last option Use Connection String, and click the **Build** button.
7. From the list of OLEDB providers, select Microsoft Jet 4.0 OLEDB provider and click the **Next** button.

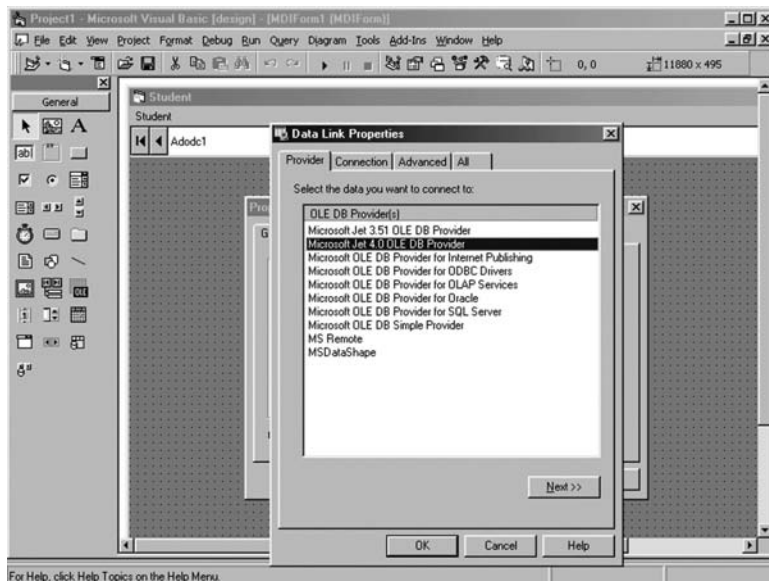


FIGURE 21.12

8. Enter the path of the database file (Student.mdb) or click the button to browse for the file.

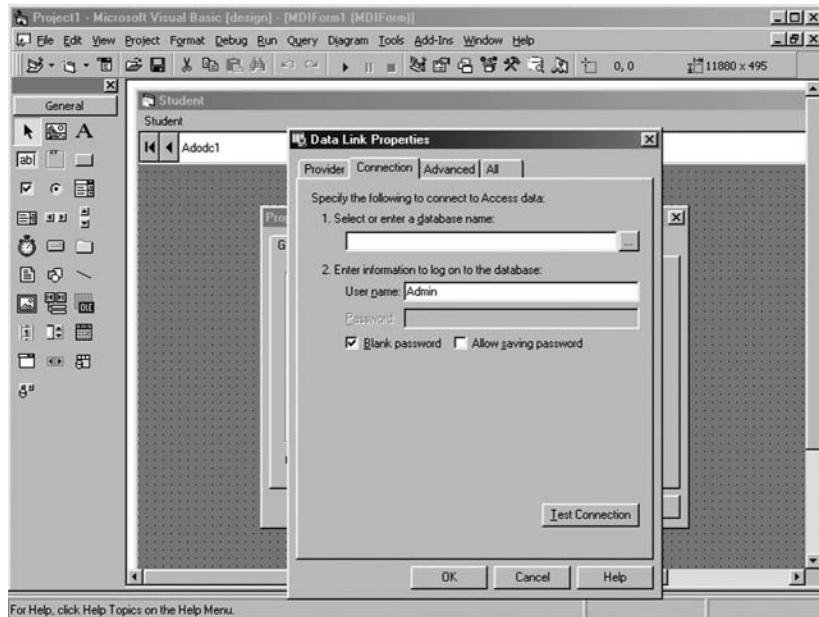


FIGURE 21.13

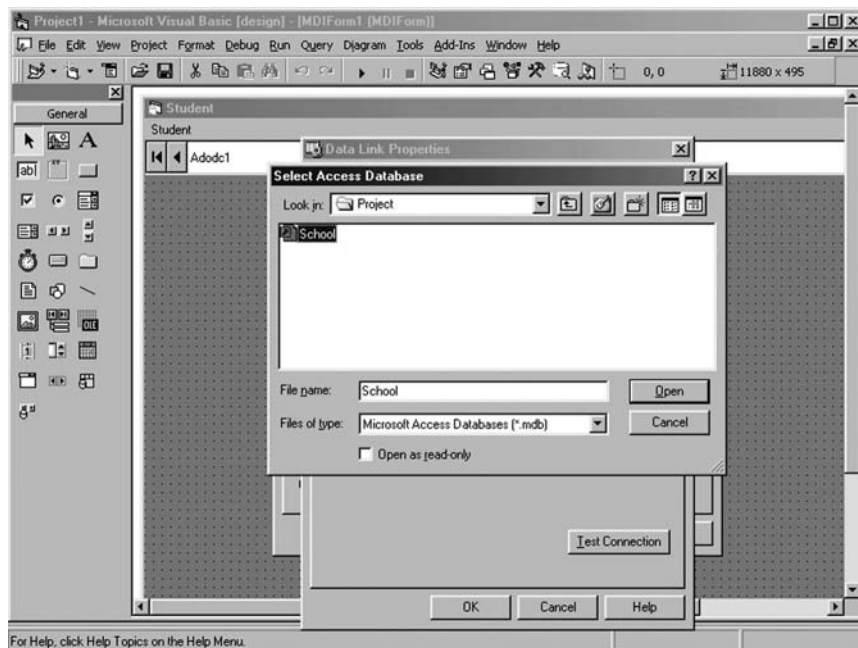


FIGURE 21.14

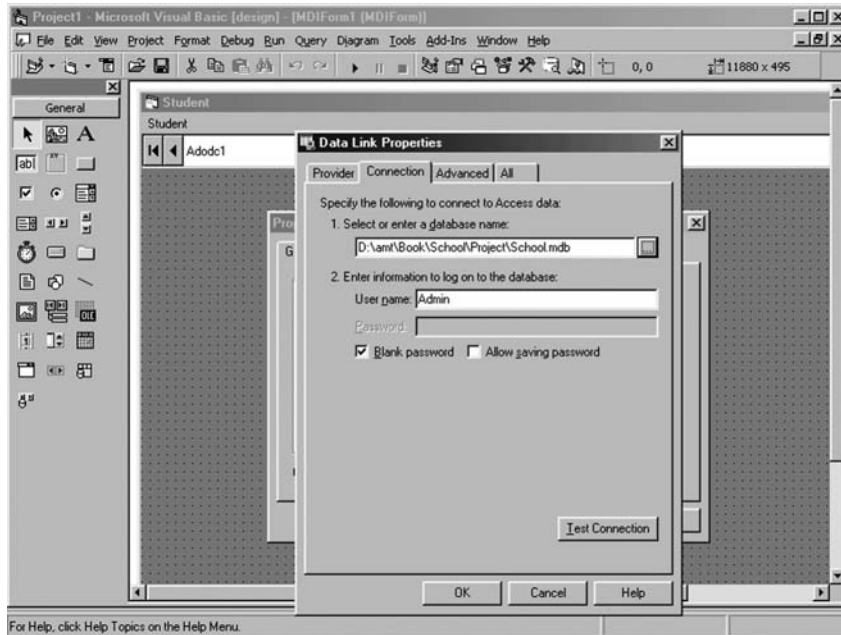


FIGURE 21.15

9. To test the connection, click the **Test Connection** button and click **OK**.

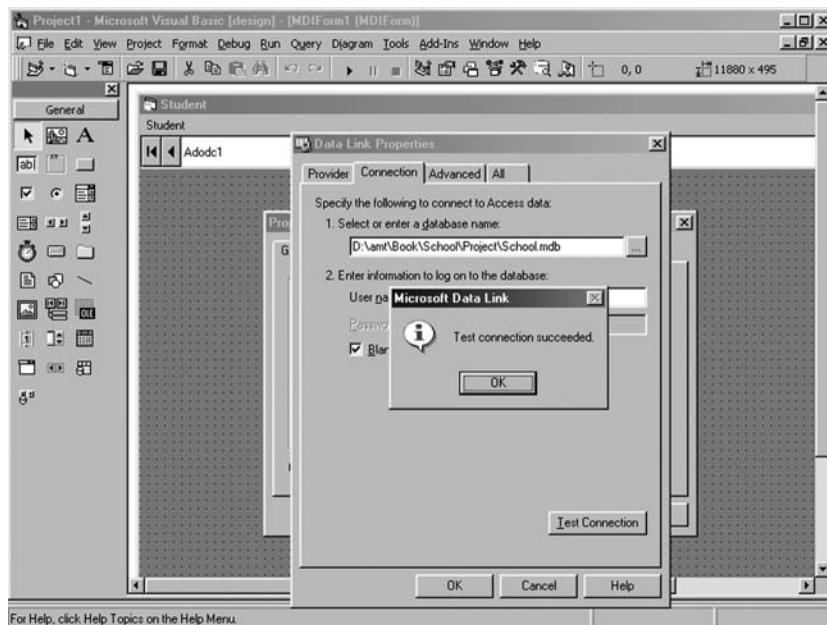


FIGURE 21.16

- Copy the string that comes in the Use Connection String box and click **OK**.

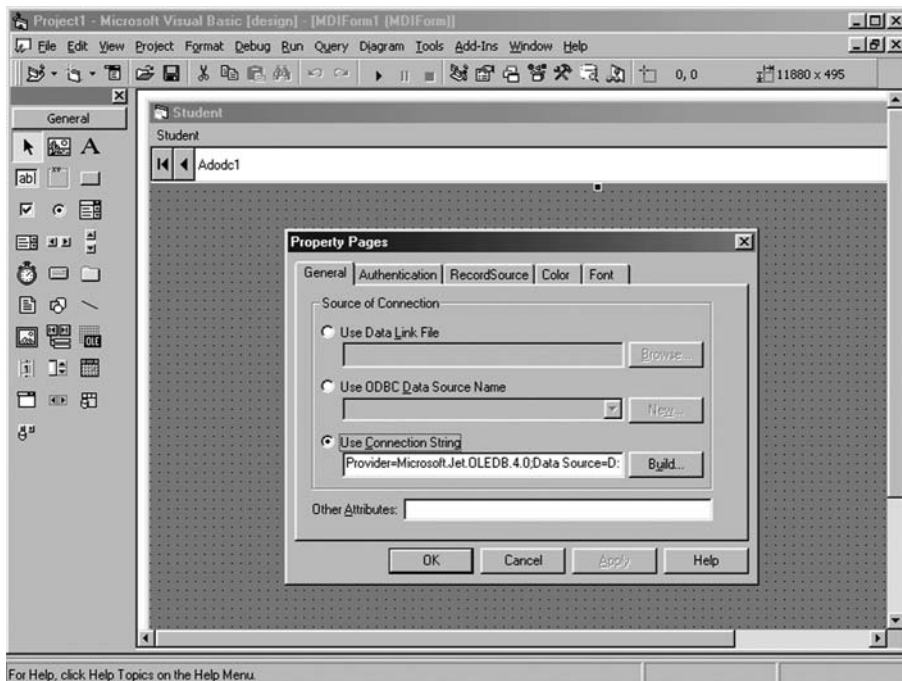


FIGURE 21.17

- Go to module and declare a Connection type public variable.

Syntax

Public conn as New ADODB.Connection

- Go to MDI Form's Form Load event and make the object of the variable conn and open the connection.

Syntax

Set conn as New ADODB.Connection

Conn. open " (Paste the Connection String copied earlier)"

- Delete the ADODC control from the MDI form.

You don't need to open the connection in each form. Make their MDIChild property True to connect with the MDI form's connection.

Code for Module

```
Public CONN As New ADODB.Connection
```

Code for MDI Form

```
Private Sub MDIForm_Load()
```

1. Set CONN = New ADODB.Connection
 2. CONN.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" &
App.Path & "\School.mdb;Persist
Security Info=False"
- ```
End Sub
```
- 

```
Private Sub Mnu_Student_Entry_Click()
```

1. Student\_Entry.Show
- ```
End Sub
```
-

```
Private Sub Mnu_Student_Report_Click()
```

1. Student_Report.Show
- ```
End Sub
```

**Code for Student Entry Form**

```
'General Declaration
```

1. Dim RST As New ADODB.Recordset 'DECLARATION OF RST VARIABLE AS RECORDSET
  2. Public IDNO As String 'Inputbox RETURNS A STRING
- 

```
Private Sub Cmddelete_Click()
```

1. IDNO = InputBox("Please Enter Student Id", "Delete")  
'EXIT FROM SUBROUTINE IF NO ID IS ENTERED
2. If IDNO = "" Then
3. Exit Sub
4. End If

```
'MAKE AN OBJECT OF THE VARIABLE TO STORE THE DATA
```

5. Set RST = New ADODB.Recordset

```
'CHECK WHETHER THE RECORD OF GIVEN ID EXIST
```

6. RST.Open "SELECT \* FROM STUDENT WHERE ID = " & IDNO & "", CONN,  
adOpenDynamic, adLockOptimistic

```

7. If RST.EOF = True Then ' i.e. NO RECORD FOUND
8. MsgBox "This Id does not exist", vbCritical, "Edit"
9. RST.Close ' CLOSE THE RECORDSET BEFORE EXIT
10. Exit Sub
11. End If

'ASK FOR CONFIRMATION TO DELETE THE RECORD
12. Dim X As Integer 'MsgBox RETURNS AN INTEGER
13. X = MsgBox("Are you sure to Delete the record", vbQuestion +
vbYesNo)
14. If X = 7 Then ' i.e. 'NO' BUTTON IS CLICKED
15. RST.Close ' CLOSE THE RECORDSET BEFORE EXIT
16. Exit Sub
17. Else
18. CONN.Execute "DELETE FROM STUDENT WHERE ID = " & IDNO & ""
19. MsgBox "Record Deleted", vbInformation, "Delete"
20. End If
21. RST.Close 'CLOSE THE RECORDSET BEFORE ENDING SUBROUTINE
End Sub

```

---

```

Private Sub Cmdedit_Click()
1. If Cmdedit.Caption = "Update" Then
2. Call Update 'CALLS UPDATE PROCEDURE TO SAVE THE RECORD AFTER
CHANGE
3. Exit Sub
4. End If
5. IDNO = InputBox("Please Enter Student Id", "Edit")
'EXIT FROM SUBROUTINE IF NO ID IS ENTERED
6. If IDNO = "" Then
7. Exit Sub
8. End If

```

```

'MAKE AN OBJECT OF THE VARIABLE TO STORE THE DATA
9. Set RST = New ADODB.Recordset

```

```

'CHECK WHETHER THE RECORD OF GIVEN ID EXIST
10. Set RST = New ADODB.Recordset
11. RST.Open "SELECT * FROM STUDENT WHERE ID = " & IDNO & "", CONN,
adOpenDynamic, adLockOptimistic

```

```

12. If RST.EOF = True Then ' i.e. NO RECORD FOUND
13. MsgBox "This Id does not exist", vbCritical, "Edit"
14. RST.Close ' CLOSE THE RECORDSET BEFORE EXIT
15. Exit Sub
16. End If

 'ASK FOR CONFIRMATION TO MODIFY THE RECORD
17. Dim X As Integer 'MsgBox RETURNS AN INTEGER
18. X = MsgBox("Are you sure to edit the record", vbQuestion +
 vbYesNo)
19. If X = 7 Then ' i.e. 'NO' BUTTON IS CLICKED
20. RST.Close ' CLOSE THE RECORDSET BEFORE EXIT
21. Exit Sub
22. Else

 'DISPLAY THE DATA IF 'YES' BUTTON IS CLICKED
23. Txtname = RST.Fields("NAME")
24. Txtage = RST.Fields("AGE")
25. Txtclass = RST.Fields("STD_CLASS")
26. DtpAdmDate = RST.Fields("ADMISSION_DATE")
27. Txtincome = RST.Fields("FATHER_MONTHLY_INCOME")
28. End If
29. RST.Close 'CLOSE THE RECORDSET BEFORE ENDING SUBROUTINE
30. Cmdedit.Caption = "Update" 'CHANGE THE CAPTION TO CALL
 'Update'
PROCEDURE
End Sub

```

---

```

Private Sub Cmdsave_Click()
 'CHECK NULL ENTRIES
1. If Txtname = "" Then
2. MsgBox "Please Enter Student Name", vbExclamation, "Incomplete
 Record"
3. Txtname.SetFocus
4. Exit Sub
5. End If
6. If Txtage = "" Then
7. MsgBox "Please Enter Age", vbExclamation, "Incomplete
 Record"

```



```

8. Txtage.SetFocus
9. Exit Sub
10. End If
11. If Txtclass = "" Then
12. MsgBox "Please Enter Class", vbExclamation, "Incomplete
 Record"
13. Txtclass.SetFocus
14. Exit Sub
15. End If
16. If Txtincome = "" Then
17. MsgBox "Please Enter Father's Monthly Income", vbExclamation,
 "Incomplete Record"
18. Txtincome.SetFocus
19. Exit Sub
20. End If
 'MAKE AN OBJECT OF THE VARIABLE TO STORE THE DATA
21. Set RST = New ADODB.Recordset
 'INCREMENT THE ID
22. Dim i As Integer
23. RST.Open "SELECT MAX(ID) FROM STUDENT", CONN, adOpenDynamic,
 adLockOptimistic
24. If IsNull(RST.Fields(0)) = False Then
25. i = RST.Fields(0) + 1
26. Else
27. i = 1
28. End If
29. RST.Close

 'RECORD INSERTION IN THE TABLE
30. CONN.Execute "INSERT INTO STUDENT VALUES(" & i & "," & Txtname
 & "," & Txtage & "," & Txtclass & "," & DtpAdmDate & ","
 & Txtincome & ")"
31. MsgBox "Student Record Saved", vbInformation, "Save"

 'CLEAR THE TEXT BOX FOR NEW ENTRY
32. Call Cmdnew_Click
 End Sub

```

---

```

Private Sub Cmdcancel_Click()
1. Unload Me

```

End Sub

---

```
Private Sub Cmdnew_Click()
1. Txtname = ""
2. Txtage = ""
3. Txtclass = ""
4. Txtincome = ""
5. DtpAdmDate = Date
6. Txtname.SetFocus
End Sub
```

---

```
Private Sub Form_Load()
'DISPLAY THE FORM AT THE CENTER OF MDIFORM
1. Me.Left = MDIFORM1.Width \ 2 - Me.Width \ 2
2. Me.Top = MDIFORM1.ScaleHeight \ 2 - Me.Height \ 2
'DISPLAY CURRENT DATE IN DTPICKER
3. DtpAdmDate = Date
End Sub
```

---

```
Private Sub Update()
'TO CHECK NULL ENTRIES
1. If Txtname = "" Then
2. MsgBox "Please Enter Student Name", vbExclamation, "Incomplete
Record"
3. Txtname.SetFocus
4. Exit Sub
5. End If
6. If Txtage = "" Then
7. MsgBox "Please Enter Age", vbExclamation, "Incomplete
Record"
8. Txtage.SetFocus
9. Exit Sub
10. End If
11. If Txtclass = "" Then
12. MsgBox "Please Enter Class", vbExclamation, "Incomplete
Record"
13. Txtclass.SetFocus
14. Exit Sub
```

```

15. End If
16. If Txtincome = "" Then
17. MsgBox "Please Enter Father's Monthly Income", vbExclamation,
 "Incomplete Record"
18. Txtincome.SetFocus
19. Exit Sub
20. End If

'ReCORD UPDATION IN THE TABLE
21. CONN.Execute "UPDATE STUDENT SET NAME = '" & Txtname & "',AGE
 = '" & Txtage & "',STD_CLASS = '" & Txtclass & "',ADMISSION_DATE
 = '" & DtpAdmDate & "',FATHER_MONTHLY_INCOME = '" & Txtincome
 & " WHERE ID = '" & IDNO & "'"
22. MsgBox "Student Record Updated", vbInformation, "Edit"

'CLEAR THE TEXT BOX FOR NEW ENTRY
23. Call Cmdnew_Click
 End Sub

```

### Code for Student Report Form

```

'General Declaration
1. Dim RST As New ADODB.Recordset ' DECLARATION OF RST VARIABLE
 AS RECORDSET

```

---

```

 Private Sub Command1_Click()
1. MSFlexGrid1.Clear
2. MSFlexGrid1.Rows = 1
3. MSFlexGrid1.Cols = 6
4. MSFlexGrid1.FormatString = "Id |Student Name |Age |Class |
 Admission Date |Father Monthly Income"
5. Dim i As Integer
 'MAKE AN OBJECT OF THE VARIABLE TO STORE THE DATA
6. Set RST = New ADODB.Recordset

 'SELECT THE RECORD
7. RST.Open "SELECT * FROM STUDENT ORDER BY ADMISSION_DATE,ID",
 CONN,
 adOpenDynamic, adLockOptimistic
8. If RST.EOF = True Then ' NO RECORD FOUND

```

```

9. MsgBox "No Record Found", vbInformation, "Student"
10. RST.Close
11. Exit Sub ' EXIT FROM SUBROUTINE
12. End If
13. Do While Not RST.EOF = True
14. MSFlexGrid1.AddItem RST!Id & Chr(9) & RST!Name & Chr(9) &
 RST!AGE & Chr(9) & RST!STD_CLASS & Chr(9) & RST!ADMISSION_DATE
 & Chr(9) & RST!FATHER_MONTHLY_INCOME
15. RST.MoveNext
16. Loop
17. RST.Close
 End Sub

```

---

```

 Private Sub Command2_Click()
1. MSFlexGrid1.Clear
2. MSFlexGrid1.Rows = 1
3. MSFlexGrid1.Cols = 6
4. MSFlexGrid1.FormatString = "Id |Student Name |Age |Class
 |Admission Date |Father Monthly Income"

 'MAKE AN OBJECT OF THE VARIABLE TO STORE THE DATA
5. Set RST = New ADODB.Recordset

 'SELECT THE RECORD BETWEEN TWO DATES
6. RST.Open "SELECT * FROM STUDENT WHERE ADMISSION_DATE >= #" &
 Format(DTPfrom, "dd-mmm-yy") & "# and ADMISSION_DATE <= #" &
 Format(DTPto, "dd-mmm-yy") & "# ORDER BY ADMISSION_DATE, ID",
 CONN, adOpenDynamic, adLockOptimistic
7. If RST.EOF = True Then ' NO RECORD FOUND
8. MsgBox "No Record Found", vbInformation, "Student"
9. RST.Close
10. Exit Sub ' EXIT FROM SUBROUTINE
11. End If
12. Do While Not RST.EOF = True
13. MSFlexGrid1.AddItem RST!Id & Chr(9) & RST!Name & Chr(9) &
 RST!AGE & Chr(9) & RST!STD_CLASS & Chr(9) & RST!ADMISSION_DATE
 & Chr(9) & RST!FATHER_MONTHLY_INCOME
14. RST.MoveNext
15. Loop

```

```
16. RST.Close
 End Sub
```

---

```
 Private Sub Form_Load()
1. DTPfrom = Date
2. DTPto = Date
 End Sub
```

## 21.3 USE OF APP.PATH

---

You can use App.Path to access a file if you are not sure about its location. App.Path brings you to the path where your project or executable file is residing. You can use this path to locate other files that you are using in your project.

Look at the example given below to access the school.mdb file to open the connection.

If school.mdb is present in the same folder where your project or executable file is residing, use the following code:

### Syntax

```
1. CONN.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" &
 App.Path & "\School.mdb;Persist Security Info = False"
```

If school.mdb is present in the folder DATA and this folder is present where your project or executable file is residing, use the following code:

### Syntax

```
1. CONN.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" &
 App.Path & "\.DATA\School.mdb;Persist Security Info=False"
```

If school.mdb is present in the folder DATA and this folder is present one level up from the folder where your project or executable file is residing, use the following code:

### Syntax

```
1. CONN.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" &
 App.Path & "..\DATA\School.mdb;Persist Security Info=False"
```

# Chapter 22

## PROGRAMMING WITH ORACLE AND SQL SERVER 2000

In the last chapter you learned how a project is developed in Visual Basic with an MS Access database. Now we are going to work with an Oracle 8 database. The project is the same that we developed with Access. We will discuss only the changes that we have to make in the project while working with Oracle.

### 22.1 TABLE CREATION

---

Make a new user school with the password student and under this user create a table student. The structure of the table is given below:

| Field name            | Data type            |
|-----------------------|----------------------|
| ID                    | Number (Primary key) |
| Name                  | Varchar              |
| AGE                   | Number               |
| STD_CLASS             | Varchar              |
| ADMISSION_DATE        | Date                 |
| FATHER_MONTHLY_INCOME | Number               |

## 22.2 DATA LINKS

You can add a data link in the project to create, rename, delete, modify, and open the tables.

### Steps to create data link:

1. Click the Data View window (a yellow color box adjacent to **Toolbox** button in the Toolbar).



FIGURE 22.1



Color figures available on CD.

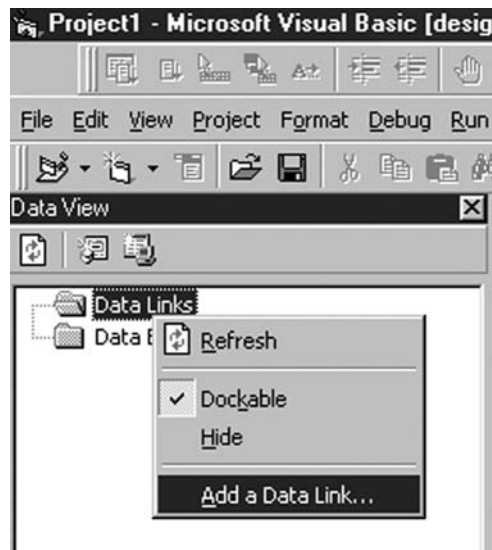


FIGURE 22.2. A New Data View Window Appears

2. Right-click Data Links and select the Add a Data Link option.
3. Select the option Microsoft OLE DB Provider for Oracle from the provider list and click the **Next**.

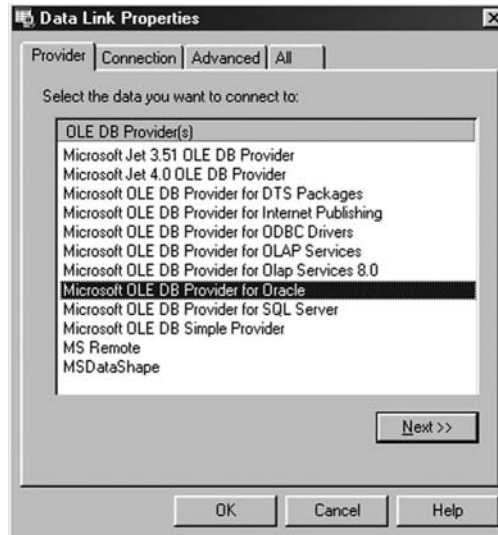


FIGURE 22.3

4. In the Connection tab give the username SCHOOL in and the password student. Click the Allow saving password check box and click the **Test Connection** button.

A message box with the message Test connection succeeded will appear. Click **OK** and then click the **OK** button at the bottom.

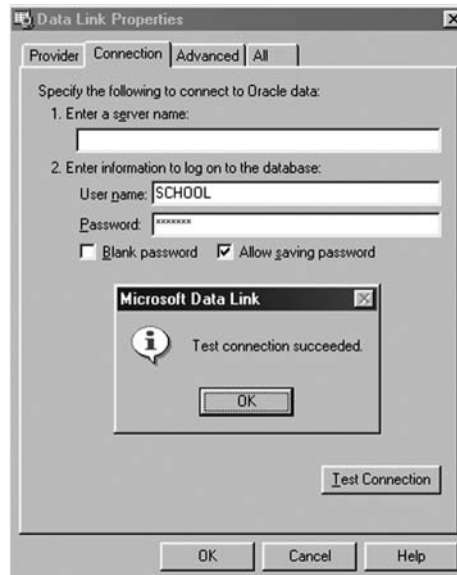


FIGURE 22.4



5. A data link is created with the name DataLink1. You can rename it to SCHOOL.
6. Expand the SCHOOL data link and then expand the Tables option below the data link. Right-click the Tables option and select the New Table option to create a new table. Create a new table called Student if you have not created one already. After designing the table click the cross button of the Table Design window. It will ask to save the table before closing the window. Click **Yes** to save the table (see Figure 22.5).

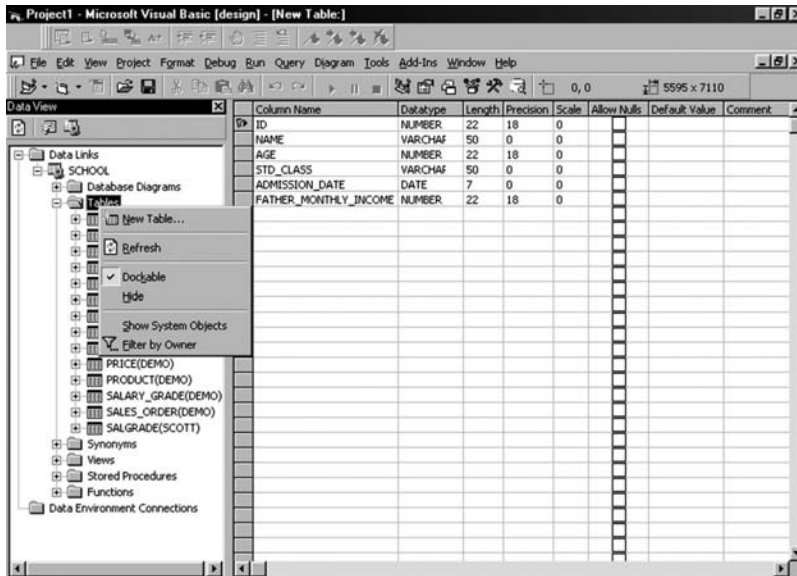


FIGURE 22.5

7. When you expand the Tables option of the data link, it displays all the tables of all the users. You can choose to display the tables of only those users to which you are currently connected. To do this right-click the Tables option and select the Filter by Owner option. A screen will appear to enter the user name. Give the user name in uppercase letters.



FIGURE 22.6

8. Right-click the table and select the option to open, design, or delete the table as needed.

### 22.2.1 Creating a Connection

Follow these steps to connect your MDI form with the Oracle database.

1. Open the design view of the MDI form.
2. Add the ADODC control in the Tool Box and draw it in the MDI form.
3. Go to the ADODC properties and select the Use Connection String option and click the **Build** button.
4. From the list of OLEDB providers, select Microsoft OLEDB Provider for Oracle and click **Next**.

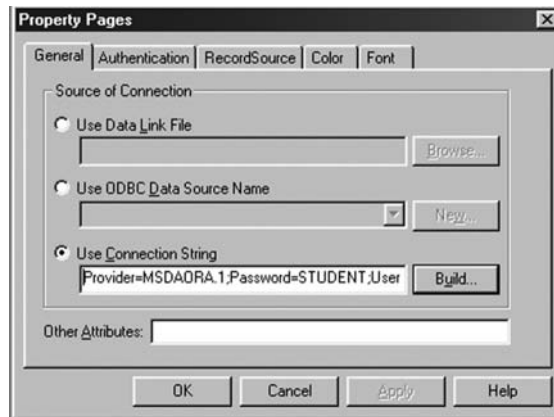


FIGURE 22.7

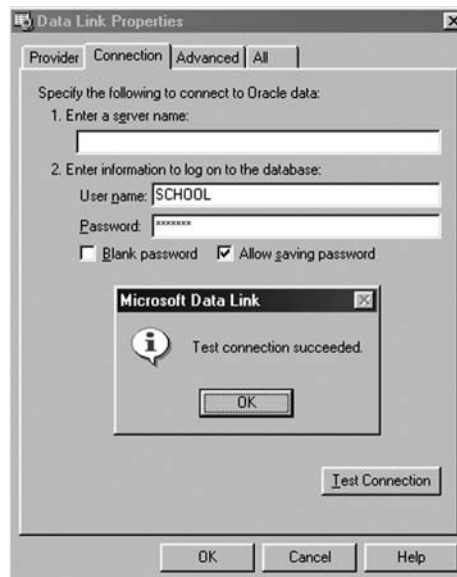


FIGURE 22.8

5. In the connection window give the user name SCHOOL in the username box and the password student. Check the Allow saving password check box and click the **Test Connection** button. A message box with the message Test connection succeeded will appear. Click **OK** and then click **OK** again at the bottom (see Figure 22.8).

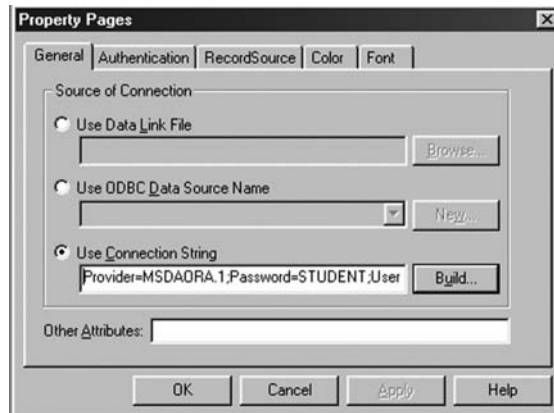


FIGURE 22.9

6. Copy the string that comes in the Use Connection String box and click OK.
7. Go to MDI Form's Form load event and paste the string.

### Syntax

```
Set conn as New ADODB.Connection
Conn.open "<Paste the string>"
```

8. Delete the ADODC Control from the MDI form and save the project.

## 22.3 WORKING WITH THE PROJECT

---

We will work in the same project that we developed using MS Access. You have to make some changes in the connectivity and programming to run the project with an Oracle 8 database.

### 22.3.1 Required Modifications

1. Change the database connection from MS Access to Oracle 8, as we have described previously.

2. Change the functions that do not support Oracle. For example, to convert the characters from lowercase to uppercase and from uppercase to lowercase the function Ucase and Lcase is used in MS Access whereas in Oracle the function Upper and Lower is used for the same purpose.
3. Oracle accepts dates in the format dd-mmm-yy, i.e., '01-Jan-05'. So you have to change the date to the format 'dd-mmm-yy' in the query wherever you are using dates.

You can use the conddate function given below to convert the date format. Write this function in the module and use it wherever you use dates, whether you are inserting dates in the table or are running a query.

### 22.3.2 Use of Conddate Function

To convert the format of current system date:

' ' & conddate(sysdate) & " - in Oracle sysdate function is used to retrieve system date.

To convert the format of a user-defined date:

' ' & conddate(DTPicker1) & " - DTPicker is used to retrieve user defined date.

#### Syntax for conddate function

```
Public Function conddate(I As Date) As String
 If Month(I) = 1 Then
 conddate = Day(I) & "-jan-" & Year(I)
 ElseIf Month(I) = 2 Then
 conddate = Day(I) & "-feb-" & Year(I)
 ElseIf Month(I) = 3 Then
 conddate = Day(I) & "-mar-" & Year(I)
 ElseIf Month(I) = 4 Then
 conddate = Day(I) & "-apr-" & Year(I)
 ElseIf Month(I) = 5 Then
 conddate = Day(I) & "-may-" & Year(I)
 ElseIf Month(I) = 6 Then
 conddate = Day(I) & "-jun-" & Year(I)
 ElseIf Month(I) = 7 Then
 conddate = Day(I) & "-jul-" & Year(I)
 ElseIf Month(I) = 8 Then
 conddate = Day(I) & "-aug-" & Year(I)
```

```

ElseIf Month(I) = 9 Then
 condate = Day(I) & "-sep-" & Year(I)
ElseIf Month(I) = 10 Then
 condate = Day(I) & "-oct-" & Year(I)
ElseIf Month(I) = 11 Then
 condate = Day(I) & "-nov-" & Year(I)
ElseIf Month(I) = 12 Then
 condate = Day(I) & "-dec-" & Year(I)
End If
End Function

```

## 22.4 DATA EXPORT AT RUNTIME

---

You can export all the tables of the user into a given file just by clicking a command button. The arguments that you have to include in the statement are:

1. Path of the file to be run
2. Username/password
3. Export file that is to be created
4. WindowState

The syntax is given below:

```
P = Shell("c:\orawin95\bin\exp80 SCHOOL/STUDENT file= c:\expdat.
dmp", vbMaximizedFocus)
```

## 22.5 WORKING IN A PROJECT WITH AN SQL SERVER 2000 DATABASE

---

You can use the same project that you used with Oracle 8 to run with an SQL Server 2000 database. The only change you have to make is to change the connectivity. Create a database with name SCHOOL and create a table Student with the same structure as was created in Oracle. Now change the connection from Microsoft OLE DB Provider for Oracle' to SQL Server.

For more details about SQL Server 2000 see the chapter SQL Server 2000.

# Chapter 23

## GRAPHS

Microsoft has included a chart control in Visual Basic 6.0. The chart control can be used to present your data in graphical format. The graphical format of data gives the user a quick overview of the data and information. In this chapter we will learn how to generate the graph of the number of students present in each class.

Add a new control, Microsoft Chart Control 6.0 (SP4) (OLEDB), in the School project.

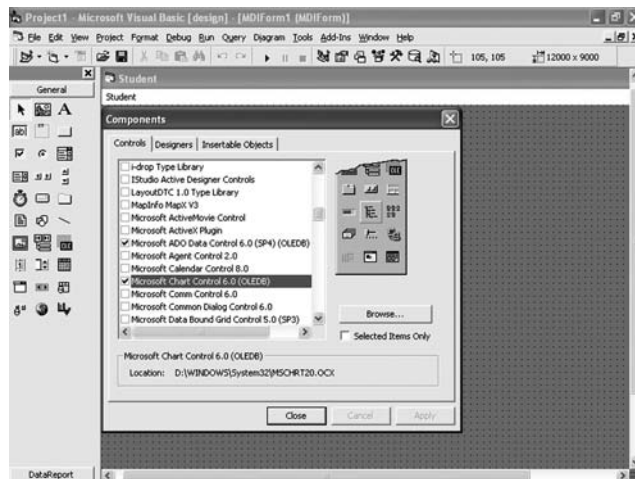


FIGURE 23.1

Add a new form in the project. Add Chart Control in the form and design the form as follows:

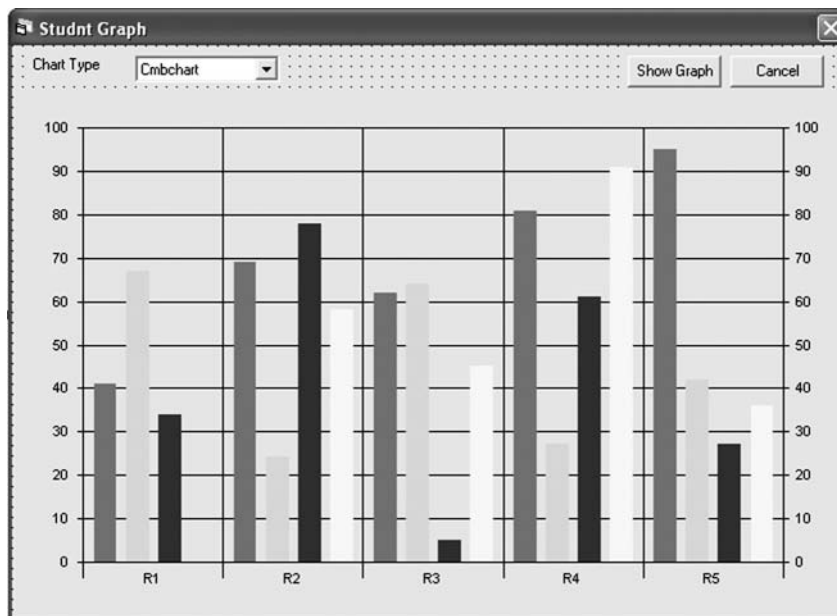


FIGURE 23.2

Add a Combo Box in the form and set its property as follows:

Name: Cmbchart  
 Style: 2- Dropdown list  
 List:  
   2D Area  
   2D Bar  
   2D Line  
   2D Pie  
   2D Step  
   3D Area  
   3D Bar  
   3D Line  
   3D Step

Add two command buttons: Command1 and Command2.

Change the property of Command 1:

Name: Cmdshow

Caption: ShowGraph

Change the property of Command 2:

Name: Cmdcancel

Caption: Cancel

Cancel: True

### Write the Given Code

General Declaration section

'DECLARATION OF RECORDSET

1. Dim rstch As New ADODB.Recordset

---

Private Sub Cmdcancel\_Click()

1. Unload Me

End Sub

---

Private Sub Cmdshow\_Click()

1. Set rstch = New ADODB.Recordset

'SHOW THE TITLE OF THE GRAPH

2. MSChart1.TitleText = "Class wise No. of Students"

'OPEN THE RECORDSET TO SELECT DATA

3. rstch.Open "select std\_class,count(\*) from student group  
by std\_class order by std\_class", CONN, adOpenStatic,  
adLockOptimistic

4. If rstch.EOF = False Then

'SET THE DATASOURCE FOR MSCHART1

5. Set MSChart1.DataSource = rstch

'SHOW THE LEGEND AT RIGHT SIDE OF THE GRAPH

6. MSChart1.ShowLegend = True

'GIVE NAME OF THE LEGEND

7. MSChart1.ColumnLabel = "No. of Students"



```

'SELECT CHART TYPE
8. If Cmbchart = "2D Bar" Then
9. MSChart1.chartType = VtChChartType2dBar
10. ElseIf Cmbchart = "3D Bar" Then
11. MSChart1.chartType = VtChChartType3dBar
12. ElseIf Cmbchart = "2D Area" Then
13. MSChart1.chartType = VtChChartType2dArea
14. ElseIf Cmbchart = "3D Area" Then
15. MSChart1.chartType = VtChChartType3dArea
16. ElseIf Cmbchart = "2D Pie" Then
17. MSChart1.chartType = VtChChartType2dPie
18. ElseIf Cmbchart = "2D Line" Then
19. MSChart1.chartType = VtChChartType2dLine
20. ElseIf Cmbchart = "2D Step" Then
21. MSChart1.chartType = VtChChartType2dStep
22. ElseIf Cmbchart = "3D Line" Then
23. MSChart1.chartType = VtChChartType3dLine
24. ElseIf Cmbchart = "3D Step" Then
25. MSChart1.chartType = VtChChartType3dStep
26. End If
27. End If
28. rstch.Close
End Sub

```

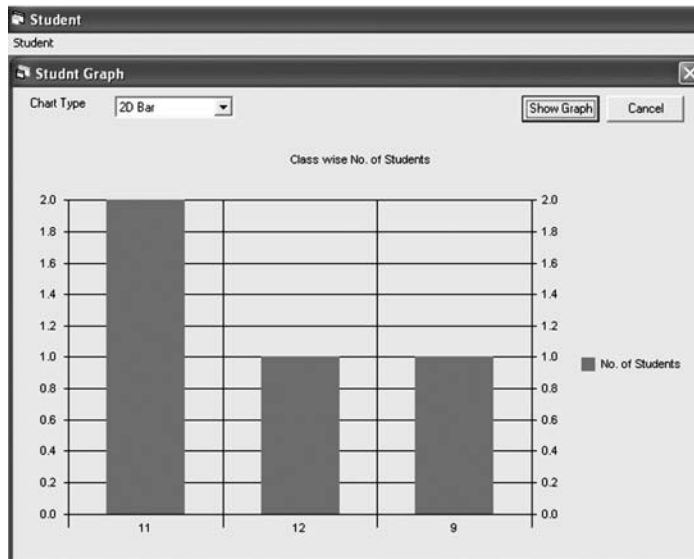


FIGURE 23.3 Display of the Graph at Runtime

# Chapter 24

## DATA REPORTS

**D**ata reporting is a method used to display data in printed format. You can print this report by attaching a printer. The data report can be saved as HTML or as a text file.

### **24.1 DATA REPORT CREATION**

---

There are three steps to create a data report:

1. Adding a data environment in the project and connecting to the database.
2. Adding a report, binding it with the data environment, and designing the data report.
3. Calling the data report.

### **24.2 DATA ENVIRONMENT AND THE CONNECTION**

---

Open the project SCHOOL in the Access database in which you have already worked.

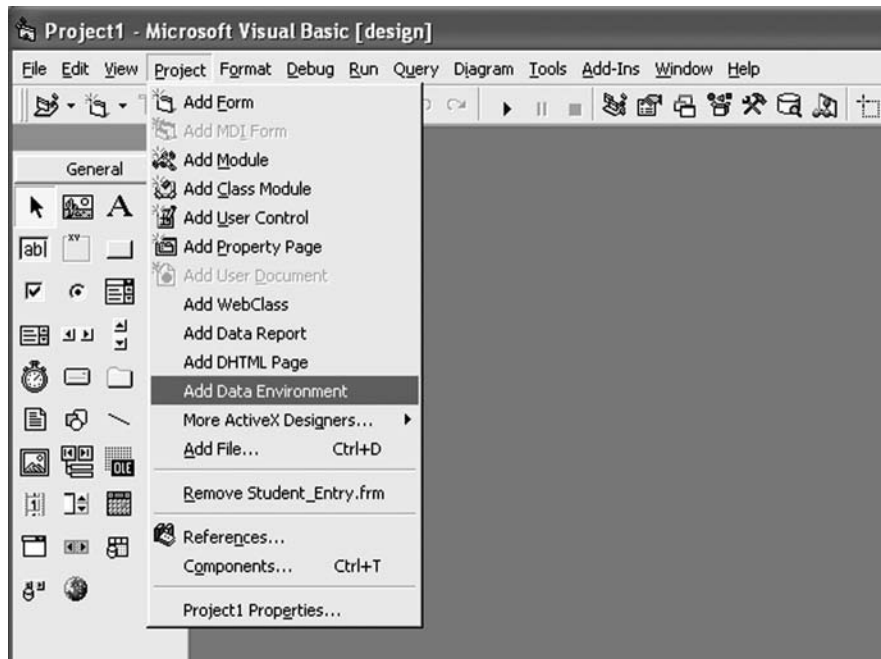


FIGURE 24.1

Add a data environment in your project by selecting the **project** → **Add Data Environment** menu (see Figure 24.1). Add only one data environment to connect with one database.

Open the data environment by double-clicking the Data Environment Designer in the Project Explorer window. Go to Connection1, right-click it, and select the properties option (see the figure).

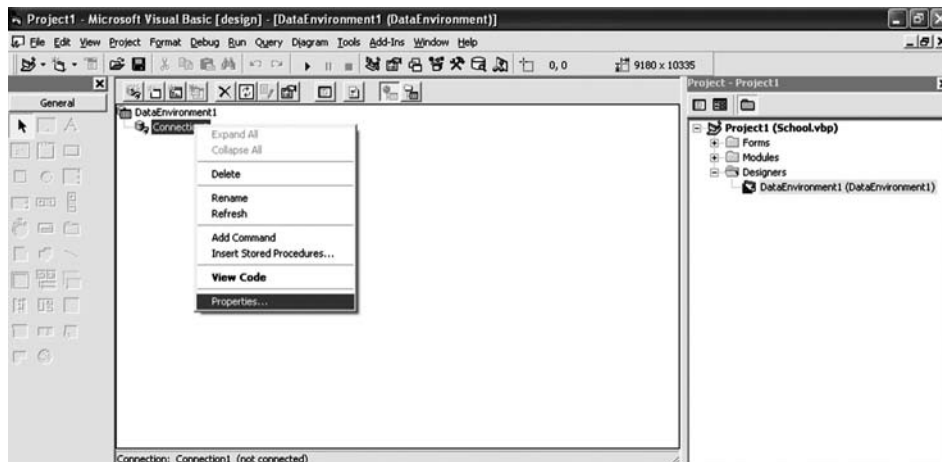


FIGURE 24.2

Select the provider Microsoft Jet 4.0 OLEDB Provider from the list of providers and click **Next**. Browse for the database file with which you want to make the connection and click the **Test connection** button to check the connection. Save the Data Environment1 in the report folder, which you have already created in the main folder School.

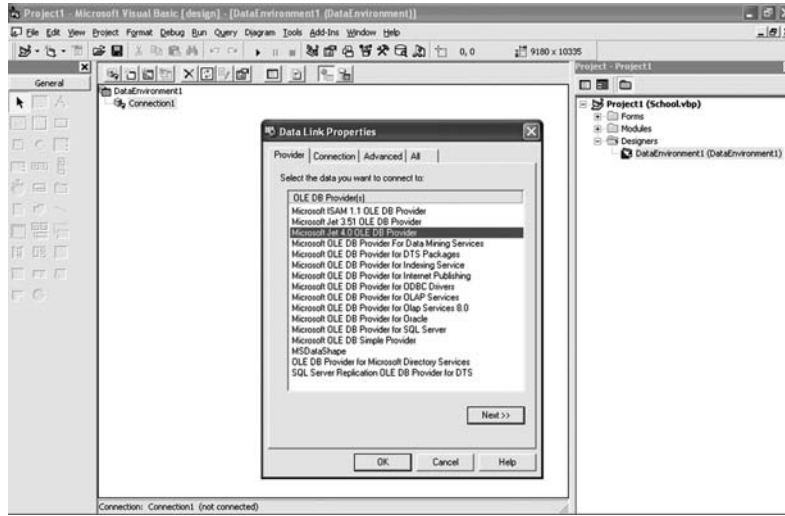


FIGURE 24.3

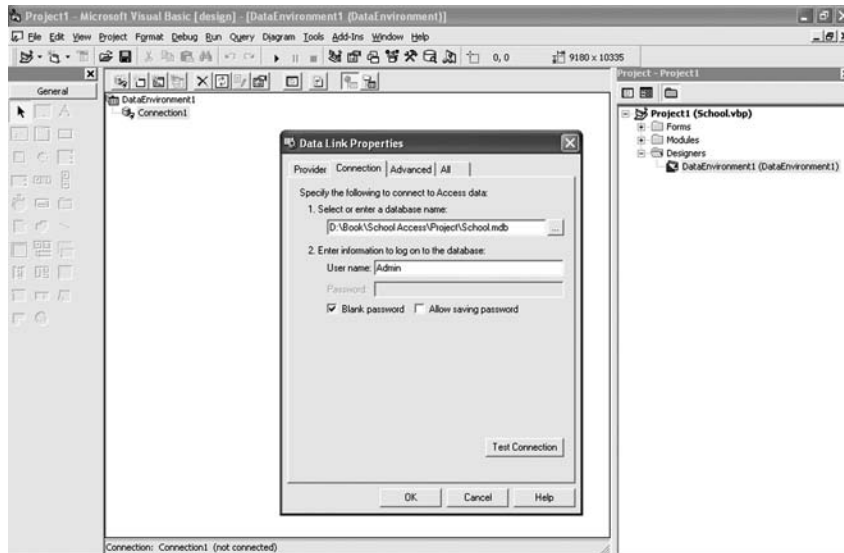


FIGURE 24.4

After establishing the data environment connection, right-click connection1 and select the Add command option (see the figure). The command stores the connection to the table and the fields that we have to show on the data report.

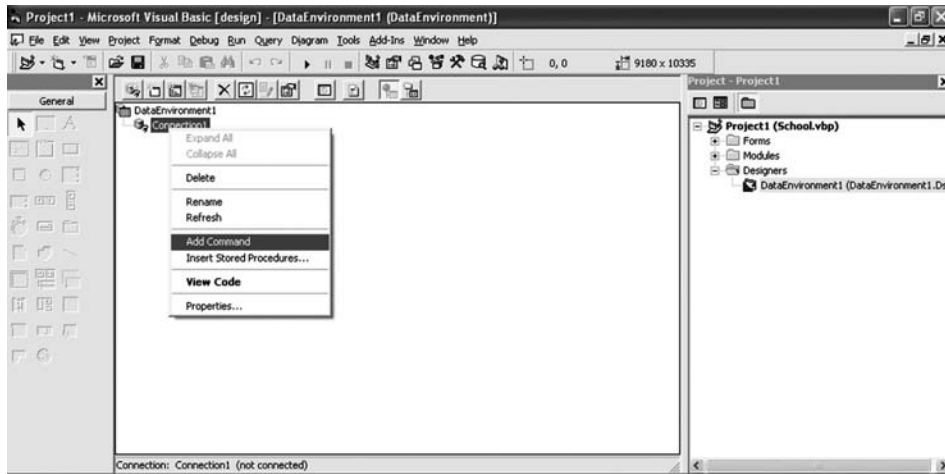


FIGURE 24.5

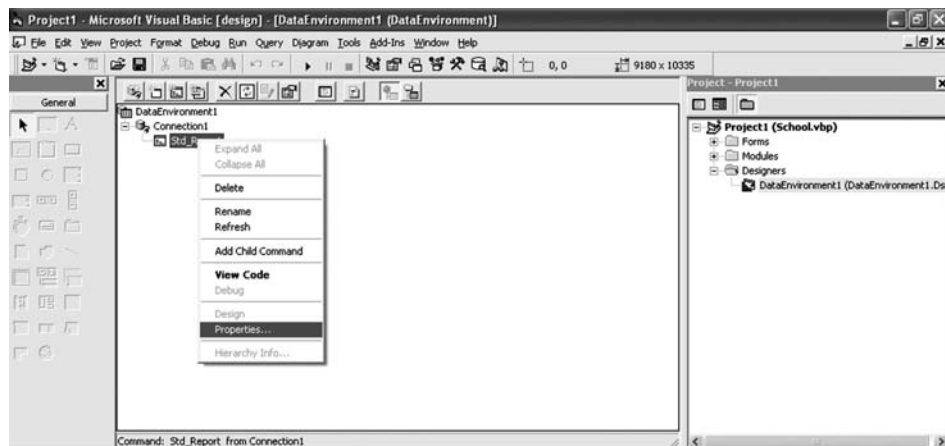


FIGURE 24.6

After selecting the Add Command option, a new command, Command1, is added under the group connection1. Rename Command 1 as Std\_Report. Right-click the command Std\_Report and select the Properties option (see Figure 24.5).

Click the **SQL Statement** option button and write the SQL Statement "Select\* From Student" in the given text box and click Apply and then **OK** (see Figure 24.6).

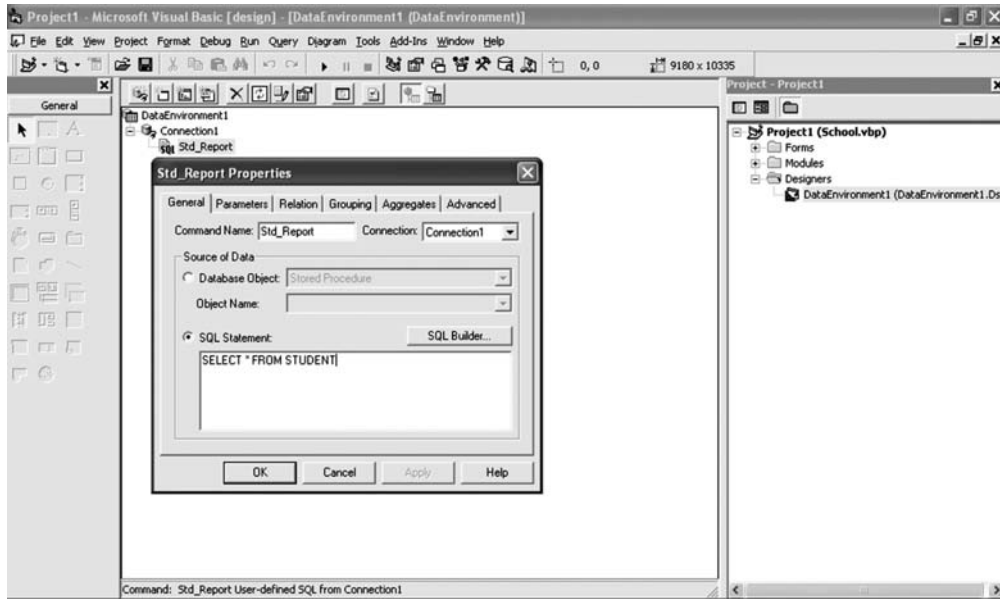


FIGURE 24.7

Expand the command Std\_Report. Now you can see all the fields of the table Student under the command (see the figure).

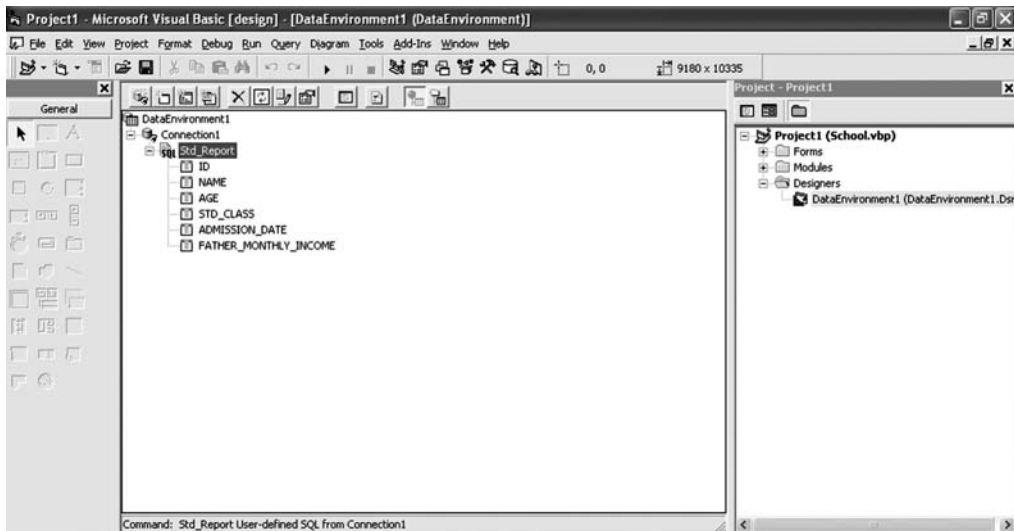


FIGURE 24.8

## 24.3 DATA REPORT DESIGNING

Add a data report to the project.

To do this go to the menu **Project** → **Add Data Report**. A new data report called Data Report1 will be added in the Designer. Double-click the Data Report1 object in the Explorer Window. This will display the data report screen. Open the Property Window and set the property of the Data Report 1 as follows:

| Name        | DataReport_student |
|-------------|--------------------|
| Caption     | Student Report     |
| DataSource  | DataEnvironment1   |
| DataMember  | Std_Report         |
| WindowState | 2-vbMaximized      |

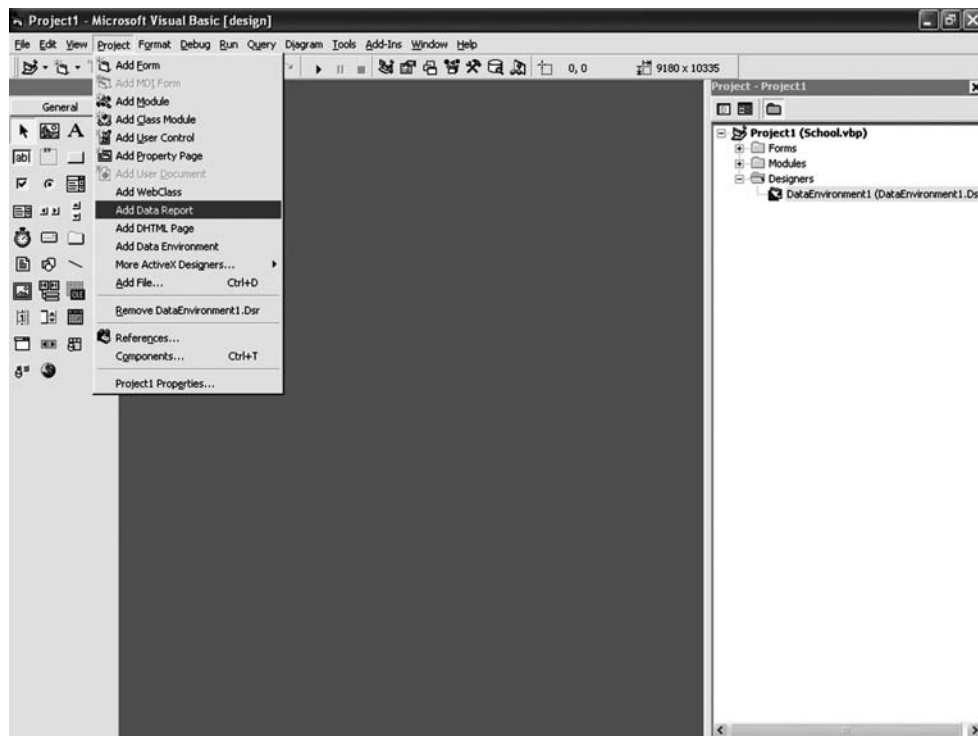


FIGURE 24.9

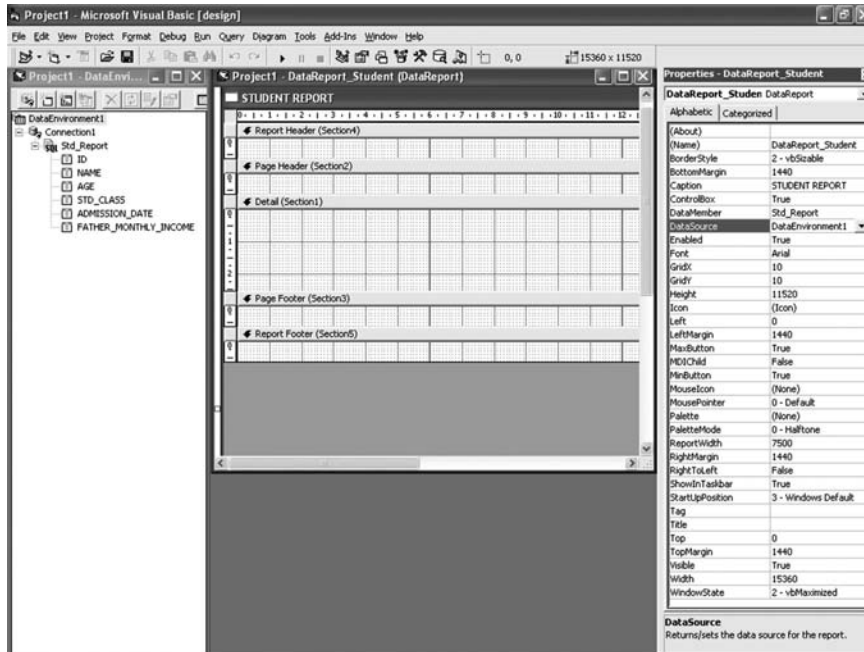


FIGURE 24.10

Now drag the fields from the command `Std_Report` of `Data Environment1` and drop the fields in the detail section of the data report (see the figure).

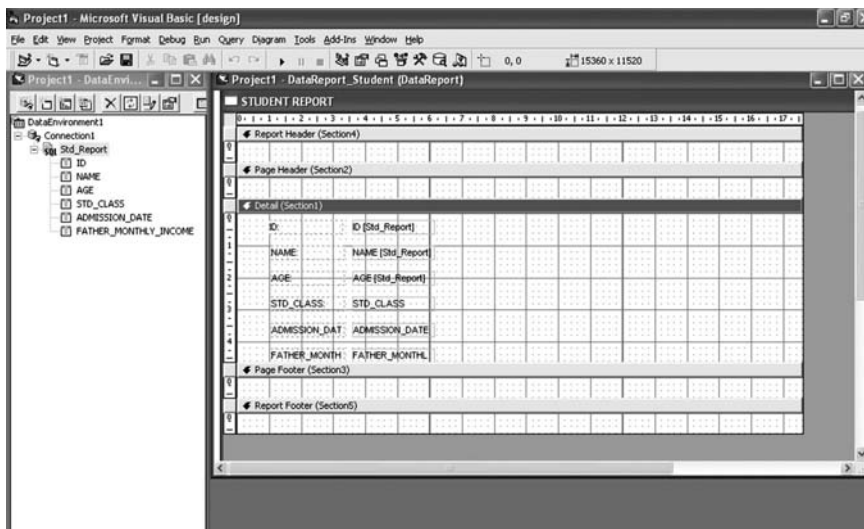


FIGURE 24.11



Each field has two components. First, is a label and second is a text box. The caption of the label is editable. You can change the caption of the label or you can use it as a column heading. In the text box, data of that particular field is displayed to which it is bound. If you place the label in the detail section, the caption of the label will be displayed with each record. To avoid the repetitive display of the label, you can place it in the Page Header section (see the figure).

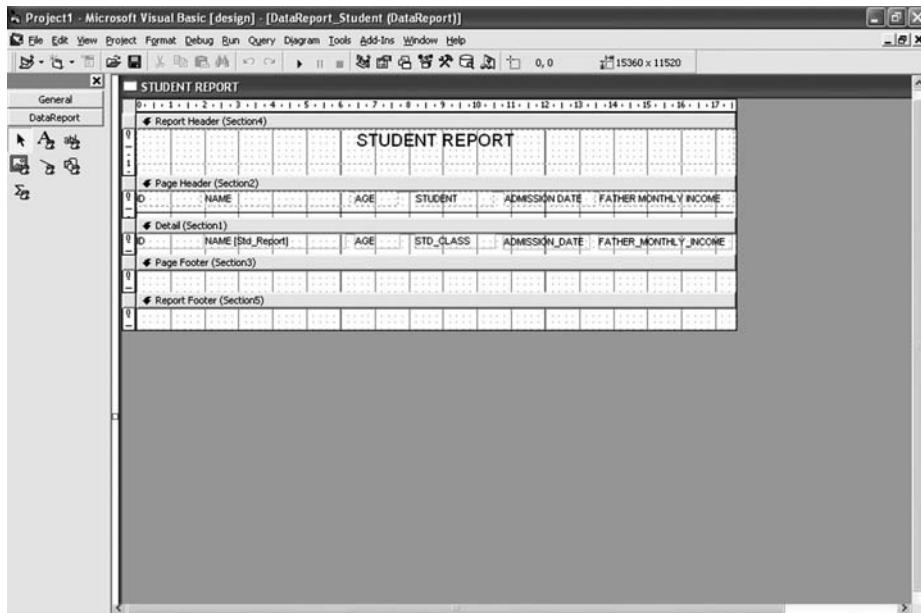


FIGURE 24.12

Draw a label from the Report Tool Box in the Report Header of the data report and set the properties of the label as given below:

|           |                     |
|-----------|---------------------|
| Alignment | 2- rptJustifyCenter |
| Caption   | Student Report      |
| Font      | Arial, 14, Regular  |

There are five sections in the data report:

1. **Report Header:** Used to display the title of the report. The control placed in this section is displayed one time in the report.
2. **Page Header:** Used to display the controls at the top of each page of the report.
3. **Detail Section:** The controls placed in this section are displayed with each record. This section is mainly used to display the data.

4. **Page Footer:** The controls placed in the page footer are displayed at the bottom of each page.
5. **Report Footer:** This section is used to display the controls at the end of the report.

## 24.4 DATA REPORT CONTROLS

---

When you add a report, the data report controls are automatically added in the control box. There are two types of controls in the Control Box – **General** and **DataReport**. General controls are used in form designing, whereas Data Report controls are used in data report designing. There are six types of Data Report controls:

| Control       | Properties                                                    | Description                                                                                                                                                                                                                                                      |
|---------------|---------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. RptLabel   | Caption<br>Font                                               | To display the caption.<br>To change the font of the caption.                                                                                                                                                                                                    |
| 2. RptTextBox | CanGrow<br><br>DataField<br><br>DataFormat<br>DataMember      | If property is set to True, it increases the width of the text box to fit all data.<br>Name of the field to which the text box is attached.<br>To change the display format of data.<br>Command name of the data environment in which the data field is present. |
| 3. Rpt Image  | Picture<br>PictureAlignment<br>SizeMode                       | Stores the path of image files.<br>Sets the alignment of the image.<br>Sets the size of the image.                                                                                                                                                               |
| 4. Rpt Line   | BorderColor<br>BorderStyle                                    | Changes the color of the line.<br>Changes the style of the line.                                                                                                                                                                                                 |
| 5. Rpt Shape  | BackColor<br>BackStyle<br>BorderColor<br>BorderStyle<br>Shape | Changes the back color of the shape.<br>Changes the back style of the shape.<br>Changes the border color of the shape.<br>Changes the border style of the shape.<br>Changes the shape to rectangle, square, oval, circle, rounded rectangle, and rounded square. |

|                |              |                                                                                                                                                                                                                                                                                                                                                       |
|----------------|--------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 6. RptFunction | DataField    | Name of the field to which the function is performed.                                                                                                                                                                                                                                                                                                 |
|                | DataFormat   | Changes the display format of data.                                                                                                                                                                                                                                                                                                                   |
|                | DataMember   | Command name of the data environment in which the data field is present.                                                                                                                                                                                                                                                                              |
|                | FunctionType | Performs different types of functions. For example:<br>0- rptFuncSum - gives the sum of the numeric field<br>1- rptFuncAve - gives the average value of the field<br>2- rptFuncMin - gives the minimum value of the field<br>3- rptFuncMax - gives the maximum value of the field<br>4- rptFuncRcnt - gives the total number of records in the report |

**NOTE** The RptTextBox can be used only in the detail section and Rpt Function can be used only in the report footer.

There are some extra controls in the data report that you can use to display the current date, time, and page number. To add these extra controls in the data report right-click data report and select the **Insert Control** option. Now you can add the following controls:

1. Current page number
2. Total number of pages
3. Current date (short format)
4. Current date (long format)
5. Current time (short format)
6. Current time (long format)
7. Report title

Add current date control and current time control in the report header and current page number in the page footer. Add the rptFunction in the report footer and set the DataMember property Std\_Report and bind data field property to Father\_monthly\_income. Add a RptLabel in the report footer and change its caption to Total.

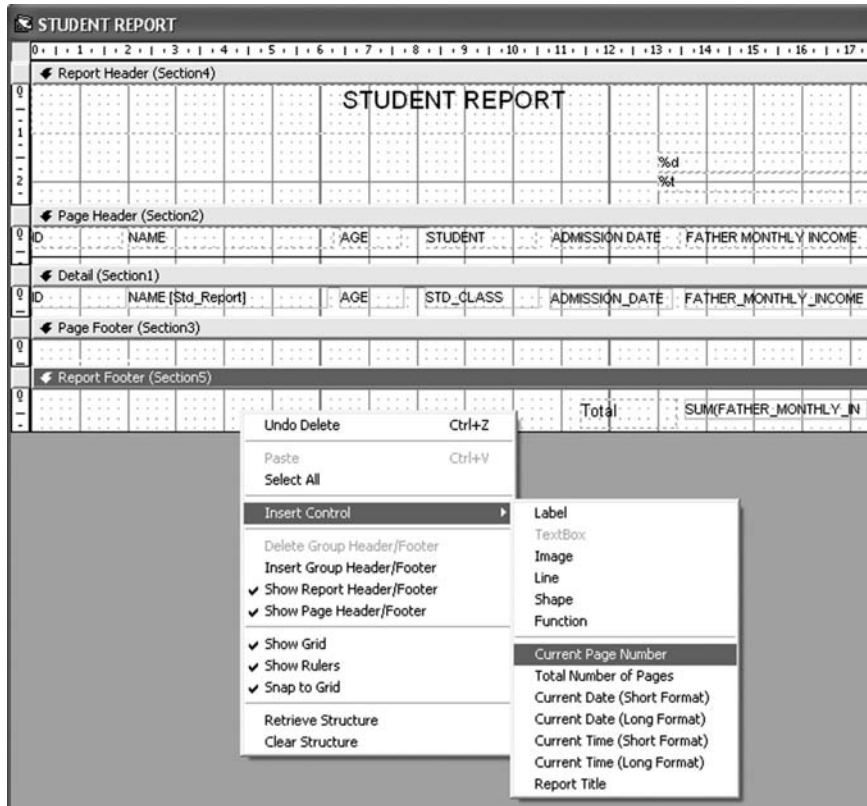


FIGURE 24.13

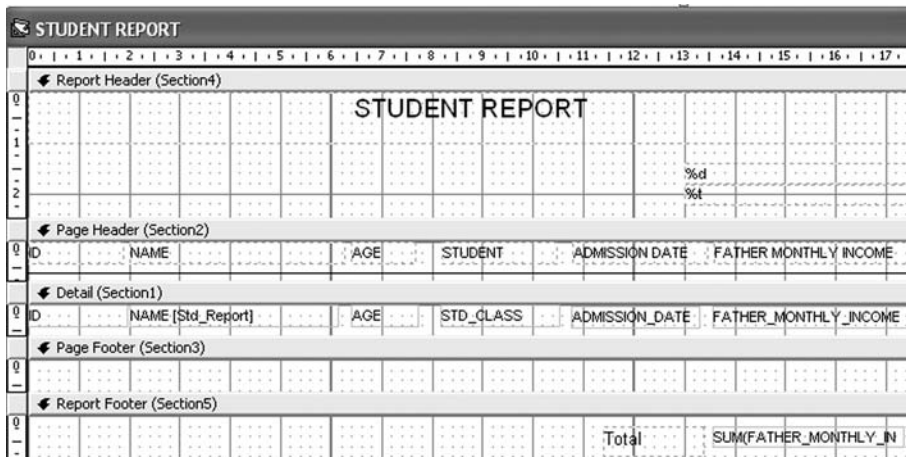


FIGURE 24.14

## 24.5 CALLING A REPORT

A data report can be called on screen using the Show method.

### Syntax

```
DataReport_Student.Show
```

01-01-02  
1:30

| ID | NAME       | AGE | STUDENT CLASS | ADMISSION DATE | FATHER MONTHLY |
|----|------------|-----|---------------|----------------|----------------|
| 1  | ARVIND     | 20  | 11            | 02-03-01       | 20000          |
| 2  | BHARAT     | 21  | 12            | 09-04-02       | 23000          |
| 4  | DHARMENDRA | 19  | 11            | 04-03-05       | 27000          |
| 3  | CHANDAR    | 22  | 9             | 04-04-05       | 12000          |
|    |            |     |               | Total          | 82000          |

Pages: [Navigation Icons]

FIGURE 24.15

At the top-left corner of the Report the Print icon is given. Click it to print the report. You can send the report directly to the printer by using Print Report method.

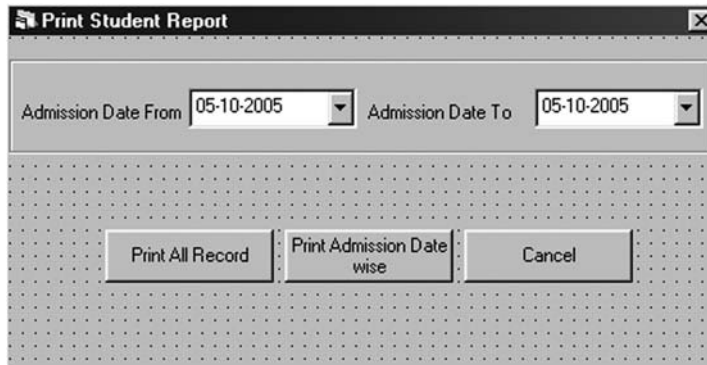
### Syntax

```
DataReport_Student.PrintReport
```

## 24.6 RETRIEVAL OF SELECTED DATA IN THE DATA REPORT

Add a new form in the project and design the form and set the properties of the form as given:

|              |                      |
|--------------|----------------------|
| Name         | Print_Student_Report |
| Border Style | 1- Fixed Single      |
| Caption      | Print Student Report |
| MDIChild     | True                 |

**FIGURE 24.16**

Make a submenu Print Student Report in the MDI form's menu and show the form.

Change the property of Command1

Name: Cmdall  
Caption: Print All Record

Change the property of Command2

Name: Cmddatewise  
Caption: Print Admission Date wise

Change the property of Command3

Name: Cmdcancel  
Caption: Cancel  
Cancel: True

Write the code in the Cmdall command button to display the records of all students.

### Syntax

```
Private Sub Cmdall_Click()
```

```

'OPEN THE DATAENVIRONMENT TO SELECT ALL RECORDS FROM THE TABLES
1. DataEnvironment1.Recordsets.Item("Std_Report").Open "SELECT *
FROM STUDENT ORDER BY ADMISSION_DATE,ID", CONN, adOpenDynamic,
adLockOptimistic

'TO CALL THE REPORT
2. DataReport_Student.Show
End Sub

```

Write the code in the Cmddatewise command button to display the records between two selected dates.

### Syntax

```

Private Sub Cmddatewise_Click()
'OPEN THE DATAENVIRONMENT TO SELECT ALL RECORDS FROM THE TABLES
1. DataEnvironment1.Recordsets.Item("Std_Report").Open "SELECT *
FROM STUDENT WHERE ADMISSION_DATE >= #" & Format(DTPfrom, "dd-
mmm-yy") & "# and ADMISSION_DATE <= #" & Format(DTPto, "dd-
mmm-yy") & "# ORDER BY ADMISSION_DATE,ID", CONN, adOpenDynamic,
adLockOptimistic 'TO DISPLAY THE DATEFROM AND DATETO IN THE
LABEL OF DATAREPORT
2. DataReport_Student.Sections(1).Controls("labelfrom").Caption
= DTPfrom
3. DataReport_Student.Sections(1).Controls("labelto").Caption =
DTPto

'TO CALL THE REPORT
4. DataReport_Student.Show
End Sub

```

Write the code in the Cmdcancel command button to unload the form.

### Syntax

```

Private Sub Cmdcancel_Click()
1. Unload Me
End Sub

```

Write the code in Form- Load() event to display the current date in DTPicker

### Syntax

```

Private Sub Form_Load()
1. DTPfrom = Date

```

```
2. DTPto = Date
End Sub
```

Go to the code window of the DataReport\_Student and write the code to close the data environment at the terminate event of DataReport\_Student.

#### Syntax

```
Private Sub DataReport_Terminate()
1. DataEnvironment1.Recordsets.Item("Std_Report").Close
End Sub
```

## 24.7 INDEX NUMBER OF DATA REPORT SECTION

---

By using the index number of the different sections of the data report you can change the property of the control at runtime. The index number of the data report section without the group header and footer is given below:

| Data Report Section | Index Number |
|---------------------|--------------|
| Report Header       | 1            |
| Page Header         | 2            |
| Detail              | 3            |
| Page Footer         | 4            |
| Report Footer       | 5            |

The index number of the data report section with the group header and is as follows:

| Data Report Section | Index Number |
|---------------------|--------------|
| Report Header       | 1            |
| Page Header         | 2            |
| Group Header        | 3            |
| Detail              | 4            |
| Group Footer        | 5            |
| Page Footer         | 6            |
| Report Footer       | 7            |



## 24.8 GROUPING IN DATA REPORTS

A data report allows you to display data in groups. Grouping can be done in one or more fields with aggregates on numeric field at each group footer.

To create grouping on a data report add a new data report in the project and save it in the Report folder with the name Data Report\_Studentgroup.Dsr. Add a new command in the Data Environment1 and rename it Groupreport. Go to the Groupreport command properties and write the SQL statement in the given box (see Figure 24.17).

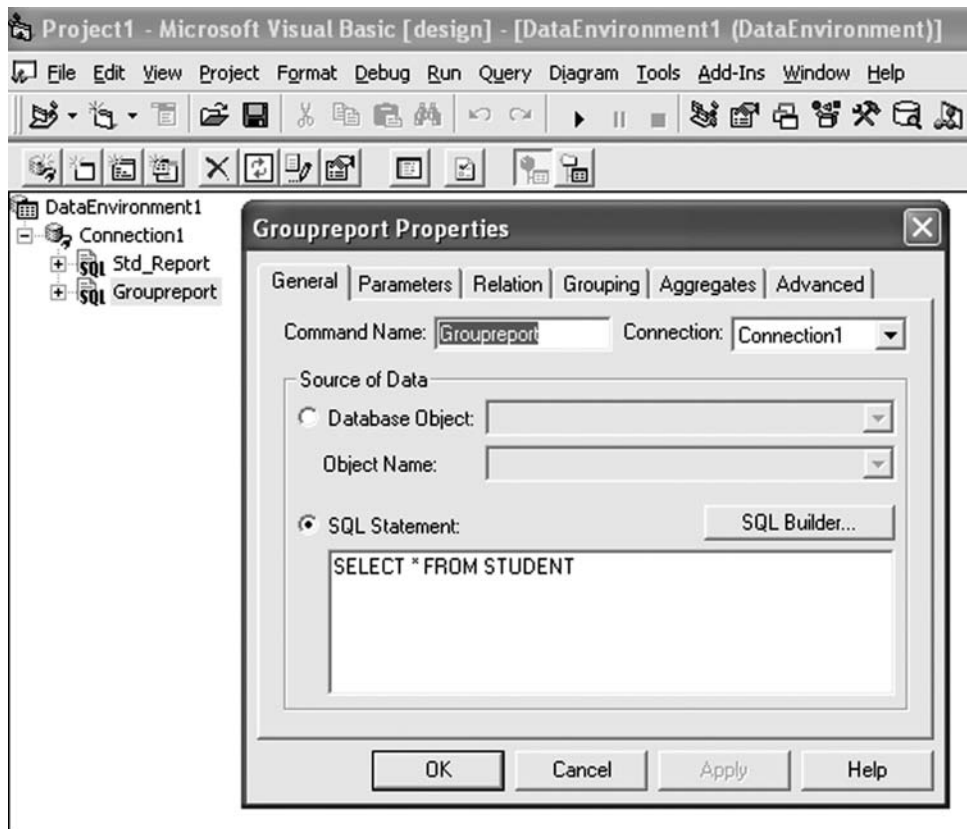


FIGURE 24.17

### SQL Statement:

```
select * From Student
```

Go to the Grouping tab and click the Group Command Object checkbox. Select the field STD\_ CLASS from the left list box and add it into the right list box (see Figure 24.18).

Click **Apply** and go to the Aggregates tab.

Click the Add button to add the aggregate function. Now set the properties for the new added function Aggregate. The first setting is to change the name of the function. The second setting is to set the type of function that is to be performed on the Aggregate1 function. In the third setting you have to select whether you want to perform the function on the grouping or on the Grand Total. The fourth and last setting is to select the field on which you want to perform the function. Here, select the field father\_monthly\_income and click **Apply** and then **OK** (see the figure).

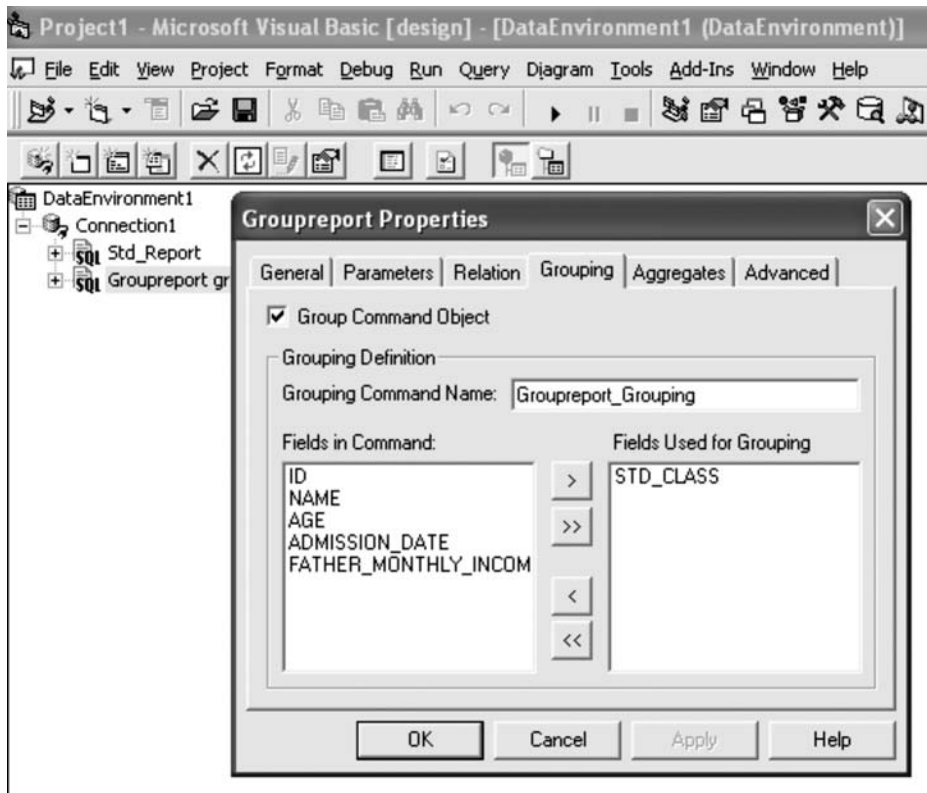


FIGURE 24.18

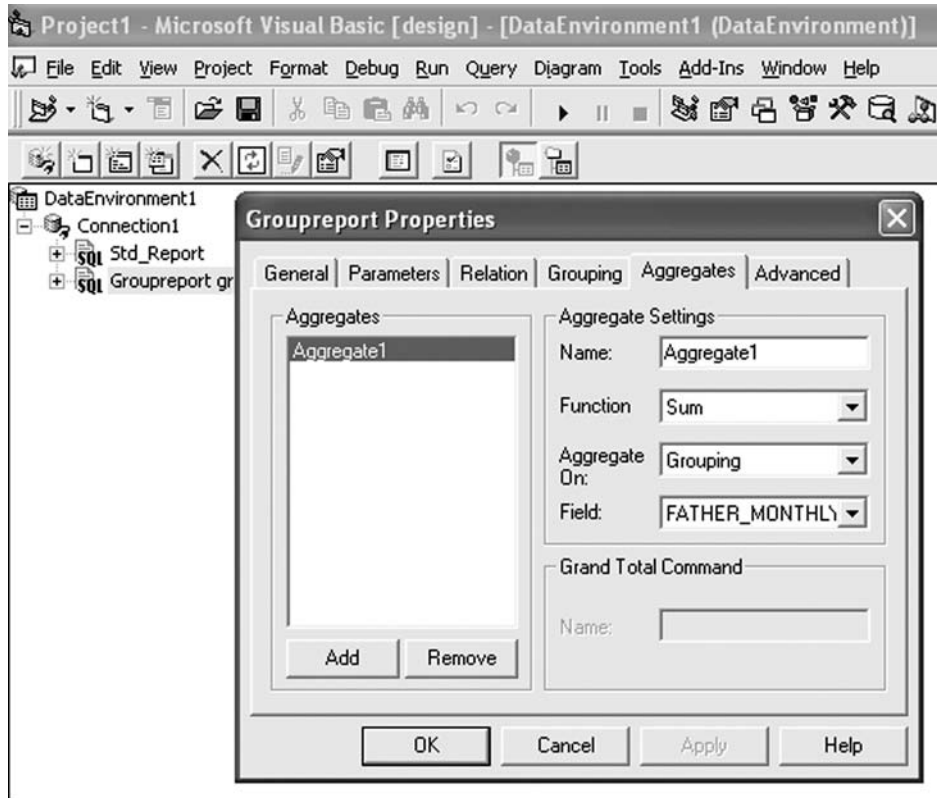


FIGURE 24.19

Now look at the Data Environment1 groupreport command. The command is divided into two different types of fields (see Figure 24.20).

The first is a summary field and the second is a detail field. Summary fields are used in the group header and footer sections of the data report, whereas detail fields are used in the detail section of the data report.

Set the properties of the DataReport\_Studentgroup as follows:

| Name        | Data Report_Student group |
|-------------|---------------------------|
| Caption     | Student Group Report      |
| DataMember  | Groupreport_Grouping      |
| DataSource  | DataEnvironment1          |
| WindowState | 2- vbMaximized            |

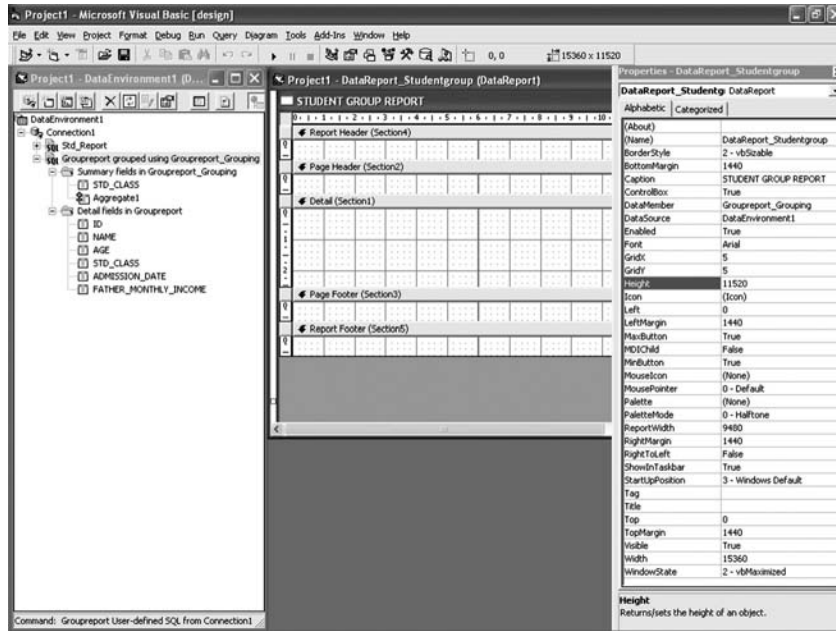


FIGURE 24.20

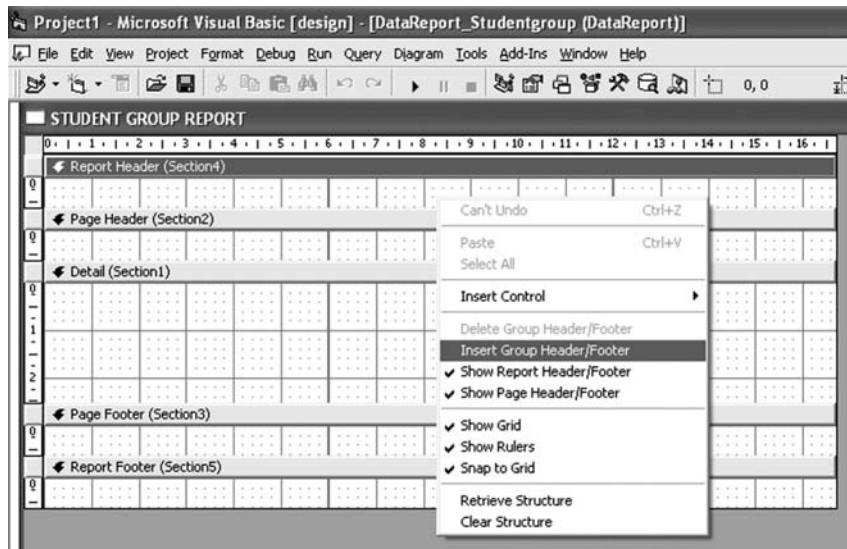


FIGURE 24.21

Now right-click the DataReport\_Student group and select the option Insert Group Header/Footer (see Figure 24.21).

The group header and group footer will be added in the data report (see the figure).

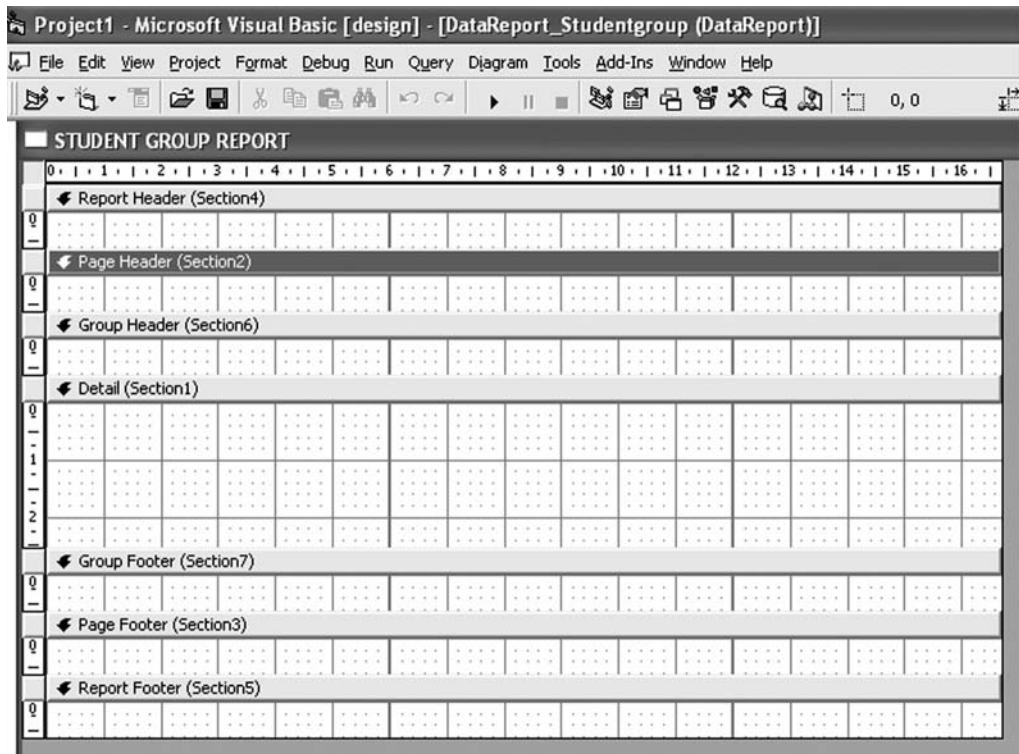


FIGURE 24.22

Now drag the STD\_CLASS field from the summary fields of the Groupreport command into the group header section of the data report. Drag the Aggregate1 field from the summary fields into the group footer section. Drag all fields except STD\_CLASS from the details fields of the Groupreport command into the detail section of the data report (see Figure 24.23).

Now design the report as given in the figure (see Figure 24.24).

To call the report at runtime, write the given syntax in the click event of the command button.

### Syntax

```
DataReport_Studentgroup.Show
```

The report at runtime will be displayed as given in Figure 24.25.

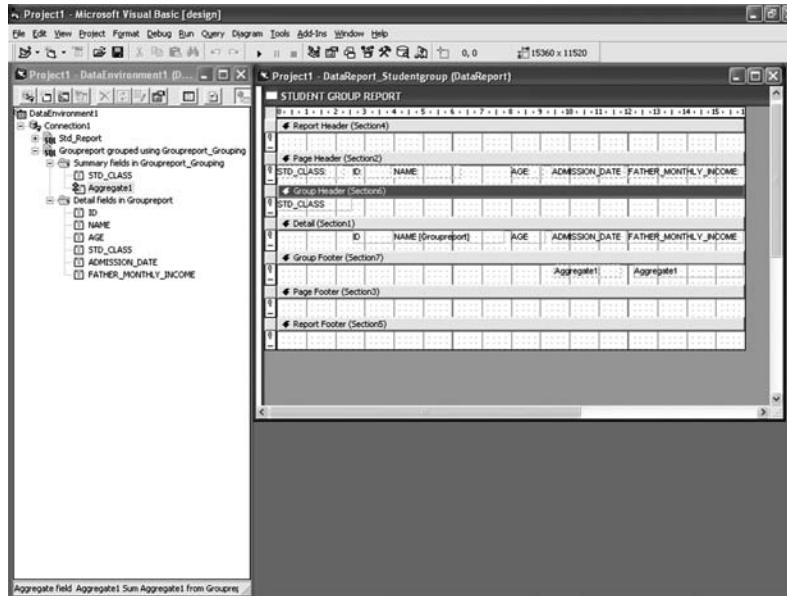


FIGURE 24.23

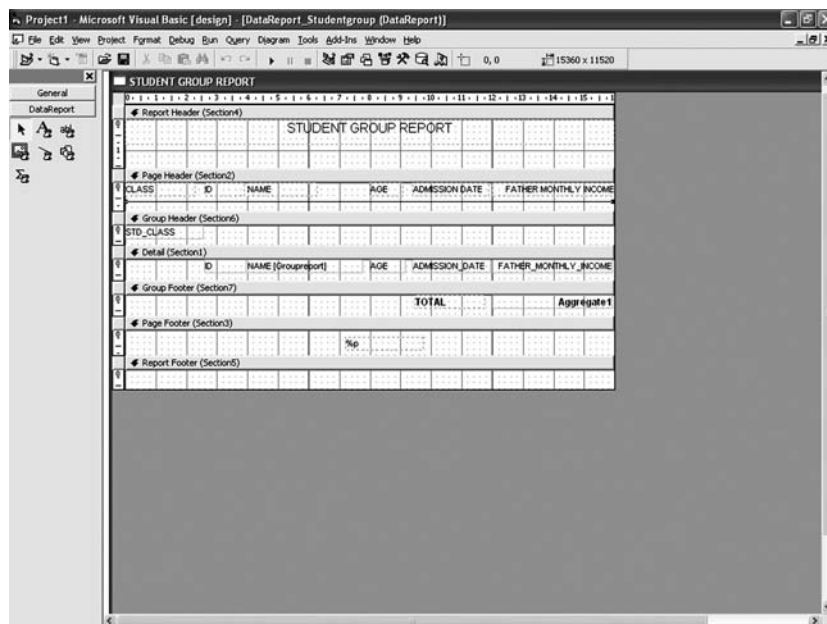


FIGURE 24.24

STUDENT GROUP REPORT

| CLASS | ID | NAME         | AGE | ADMISSION DATE | FATHER MONTHLY INCOME |
|-------|----|--------------|-----|----------------|-----------------------|
| 9     | 3  | CHANDAR      | 22  | 04-04-05       | 12000                 |
|       |    | <b>TOTAL</b> |     |                | <b>12000</b>          |
| 11    | 1  | ARVIND       | 20  | 02-03-01       | 20000                 |
|       | 4  | DHARMENDRA   | 19  | 04-03-05       | 27000                 |
|       |    | <b>TOTAL</b> |     |                | <b>47000</b>          |
| 12    | 2  | BHARAT       | 21  | 09-04-02       | 23000                 |
|       |    | <b>TOTAL</b> |     |                | <b>23000</b>          |

Pages: 1

FIGURE 24.25

# Chapter 25

## CRYSTAL REPORTS

**C**rystal Reports are specialized reports. The flexibility of Crystal Reports doesn't end with creating reports. Your report is designed to work with your database to help you analyze and interpret important information. Crystal Reports make it easy to create simple reports and also have the comprehensive tools you need to produce complex reports that can be published in a variety of formats including Microsoft Word, Excel, email, and even over the Web. Built-in **Report Experts** guide you step-by-step through building reports and completing common reporting tasks. Formulas, cross-tabs, sub-reports, and conditional formatting help make sense of data and uncover important relationships that might otherwise be hidden. Geographic maps and graphs communicate information visually when words and numbers are simply not enough.

Application and web developers can save time and meet their needs by integrating the report processing power of Crystal Reports into their database applications. Support for most popular development languages makes it easy to add reporting to any application.



## 25.1 ADVANTAGES OVER VISUAL BASIC DATA REPORTS

- Flexible
- Rich Functions
- Formula Creation
- Multiple Report Generation
- Duplicate Record Suppression
- Graph Generation
- Easily Updateable
- Query Passing
- Supports Web Reporting

## 25.2 STARTING WITH CRYSTAL REPORT 8.5

Create a table, Student.mdb, with the following fields:

| ID | NAME       | AGE | STD_CLASS | ADMISSION_DATE | FATHER_MONTHLY |
|----|------------|-----|-----------|----------------|----------------|
| 1  | ARVIND     | 20  | 11        | 02-08-04       | 20000          |
| 2  | BHARAT     | 21  | 12        | 09-09-03       | 23000          |
| 3  | CHANDAR    | 22  | 9         | 04-07-02       | 12000          |
| 4  | DHARMENDRA | 19  | 11        | 04-09-05       | 27000          |
| 5  | FARHAN     | 20  | 12        | 05-09-02       | 23000          |
| 6  | HASAN      | 21  | 12        | 09-07-01       | 12500          |
| 7  | INDRAJEET  | 20  | 11        | 08-09-02       | 27000          |
| 8  | JASNEET    | 19  | 11        | 09-03-05       | 28000          |
| 9  | KAMALJEET  | 21  | 12        | 09-02-03       | 12000          |
| 10 | MANISH     | 19  | 10        | 05-02-03       | 16000          |
| 11 | NAGENDRA   | 20  | 11        | 04-09-05       | 12900          |

FIGURE 25.1

Go to **Start** → **Programs** → **Crystal Report**.

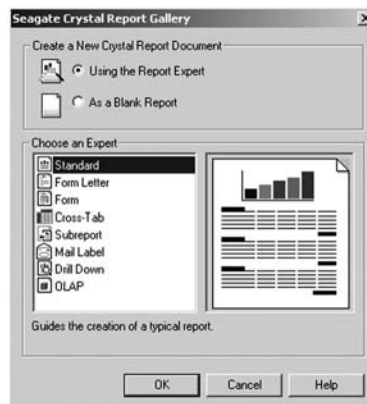


FIGURE 25.2

The screen **Crystal Report Gallery** will be displayed. Now select document type **Using the Report Expert**. Choose the **Standard** report format and click **OK**.

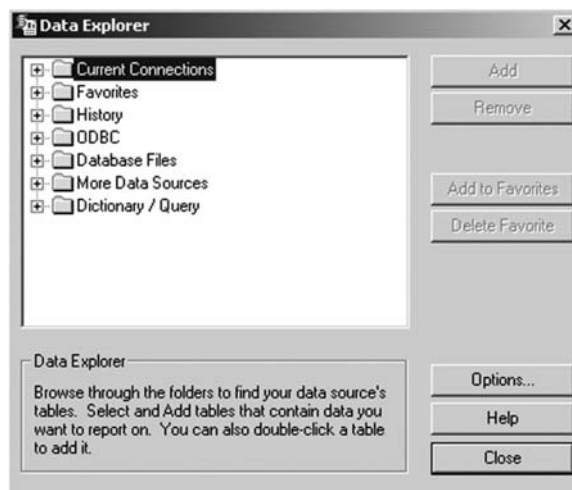


FIGURE 25.3

The above **Data Explorer** will be displayed. Now select **Database Files > Find Database File**.

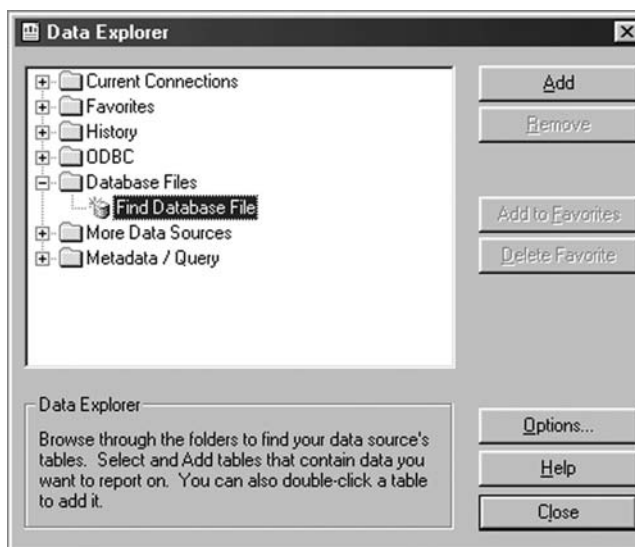


FIGURE 25.4

The open dialogue box will be displayed. Select the required database file.

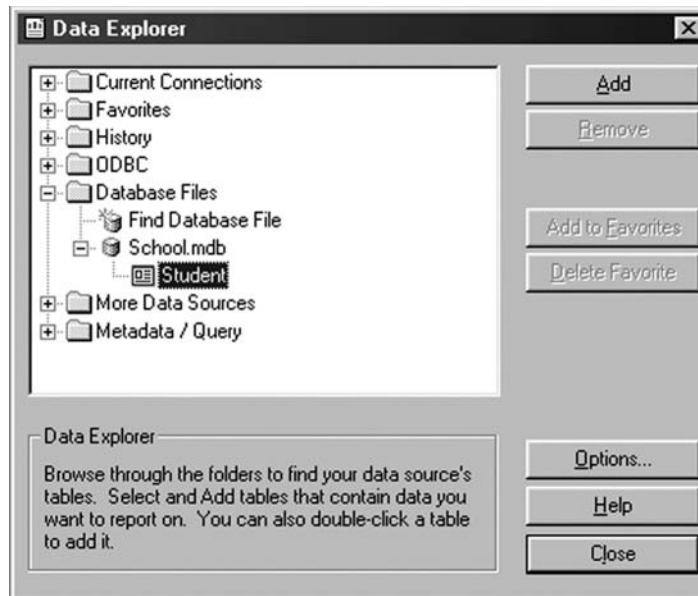


FIGURE 25.5

The open Dialog Box will be displayed. Select the required database. Now click **Add** and **Close**.

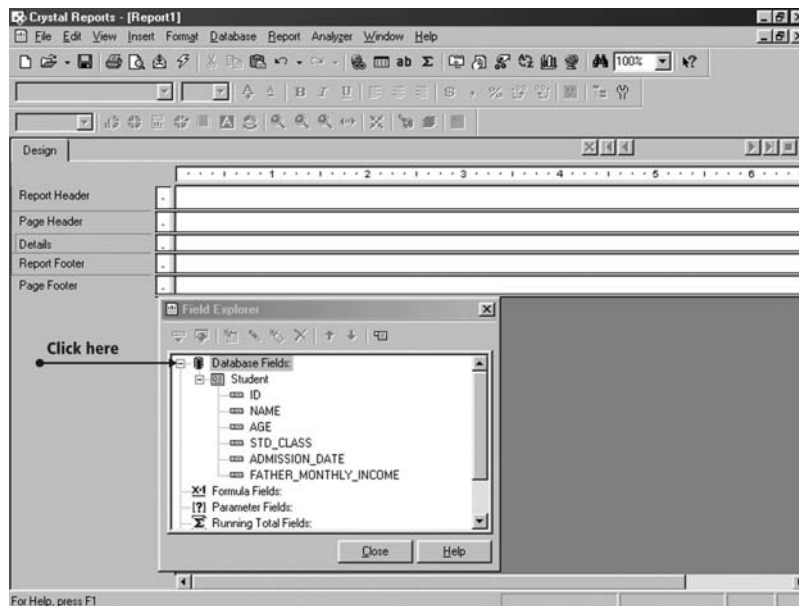


FIGURE 25.6

The above window will be displayed. Now expand the database field **Student**.

Drag the fields from the **Student** database into the detail section of the report and save the report.

You can preview the report with data by clicking the **Preview** button.

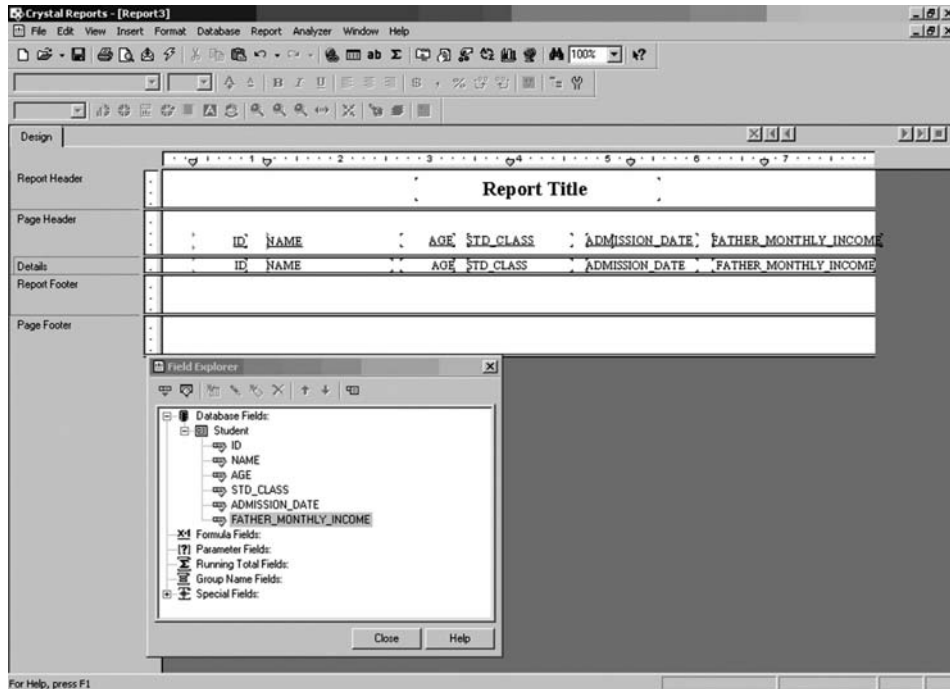


FIGURE 25.7

## 25.3 CREATING REPORTS USING DSN OF THE SQL SERVER 2000 DATABASE

The SQL Server 2000 database is used for Windows 2000, XP, and higher versions.

### 25.3.1 How to Make a DSN

Click Start → Settings → Control Panel → Administrative Tools → ODBC.

The window (Figure 25.9) will be displayed.

Now click the **Add** button.

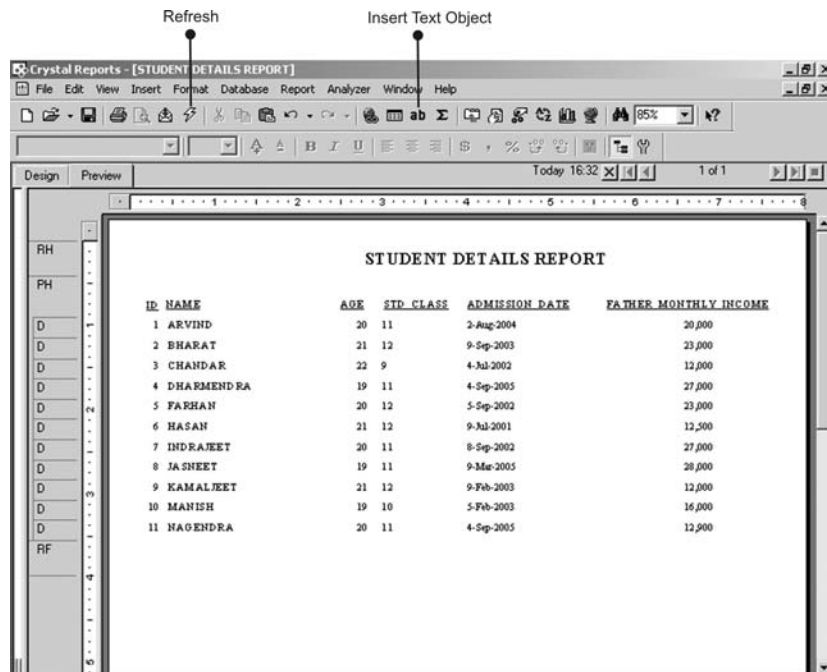


FIGURE 25.8

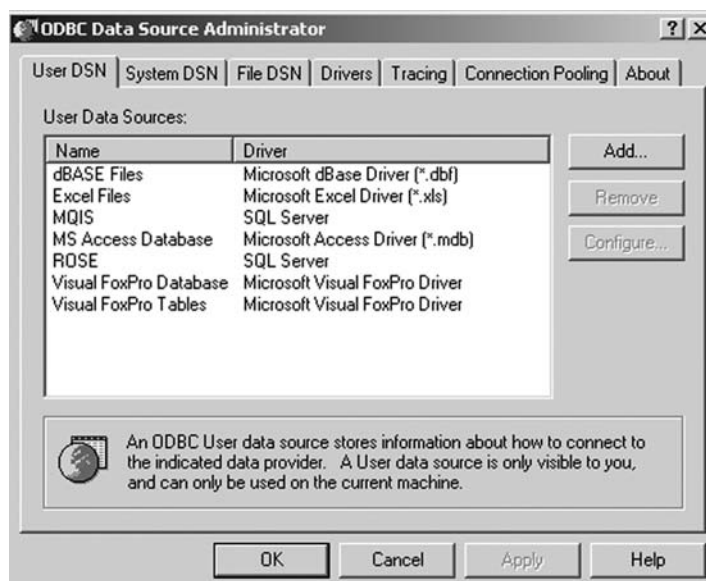


FIGURE 25.9

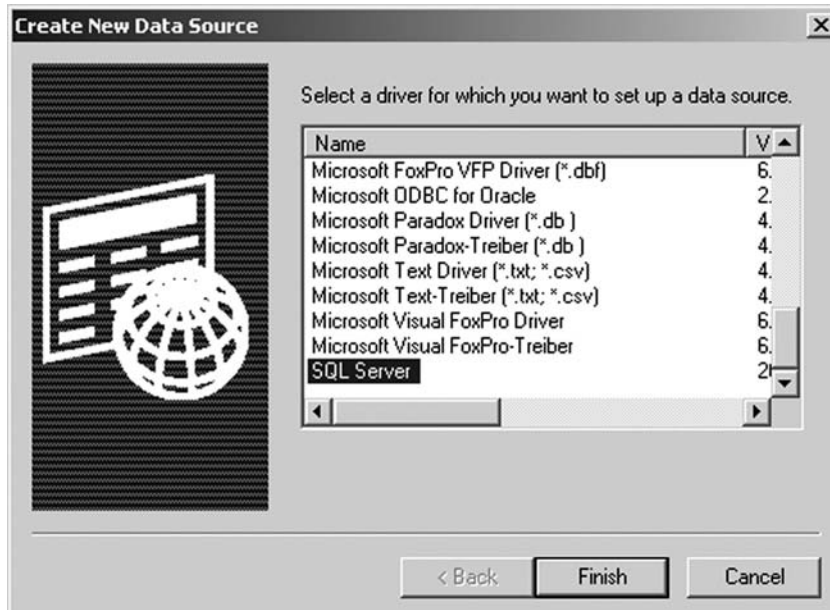


FIGURE 25.10

In the above displayed window select SQL Server from the list and click **Finish**.

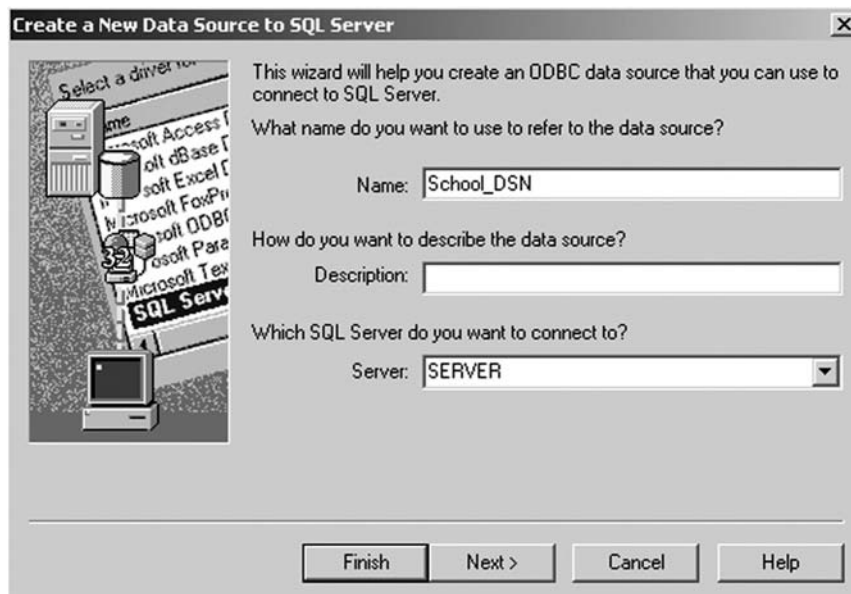


FIGURE 25.11

Enter the DSN name in the **Name** text box.

Select Server name from **Server** list.

Click the **Next** button.

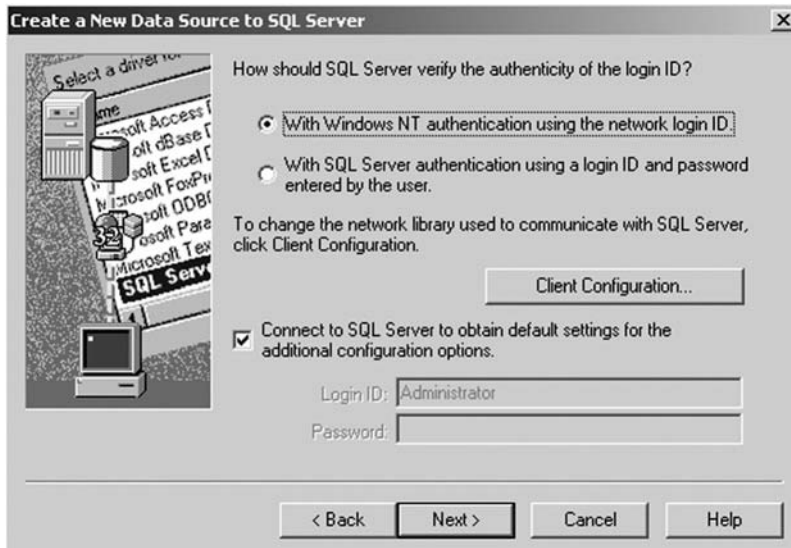


FIGURE 25.12

Select login type which may be Windows NT authentications or SQL Server authentication. Click **Next**.

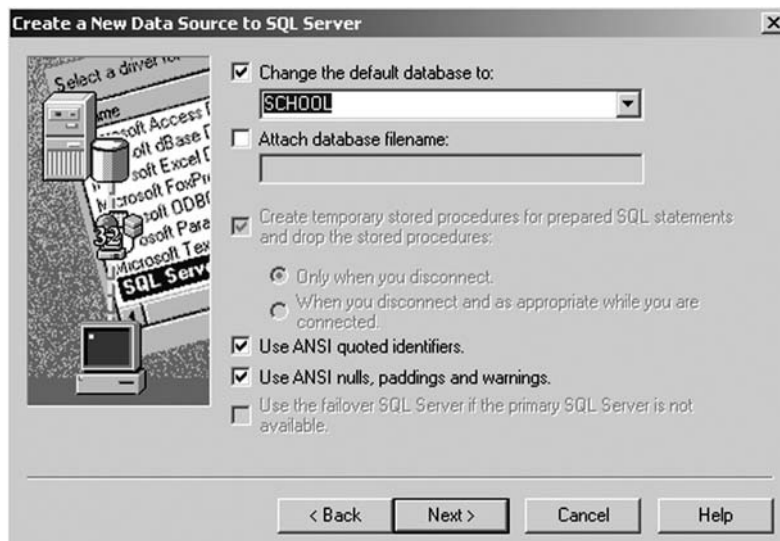


FIGURE 25.13

Select source database.

Click **Next**.

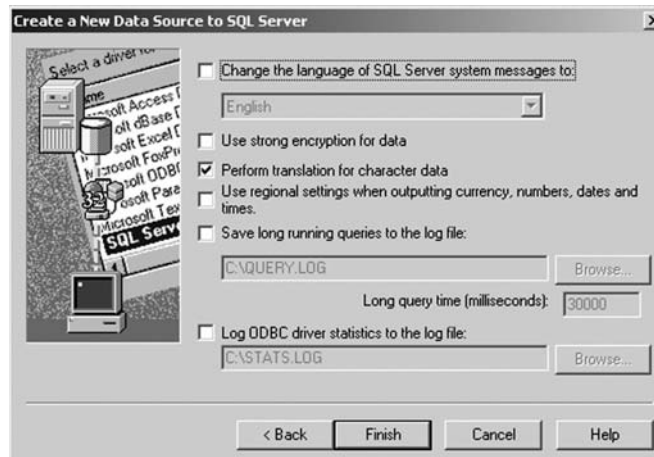


FIGURE 25.14

In this final window click **Finish**.

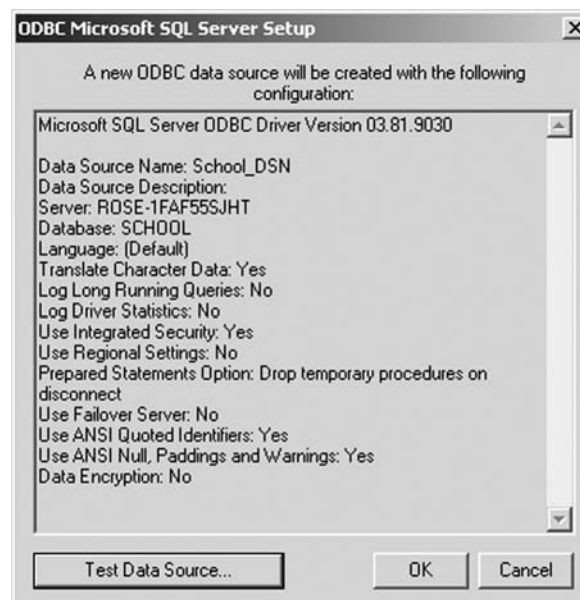


FIGURE 25.15

Click **OK**.



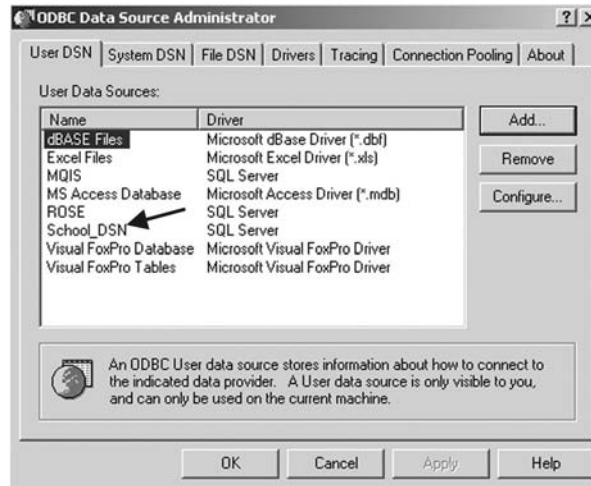


FIGURE 25.16

Now your defined DSN name is displayed in the list.

## 25.4 CREATING CONNECTION USING DSN

Go to File Menu and select Crystal Report and select

**File→New→As Blank Report.**

Select ODBC.

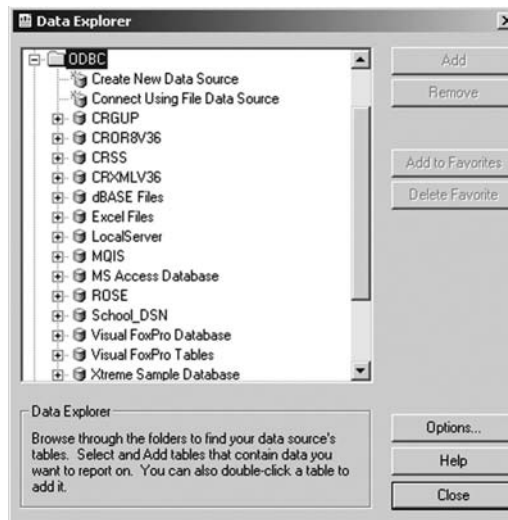


FIGURE 25.17

Select **School\_DSN** in the ODBC list and click **Add**.

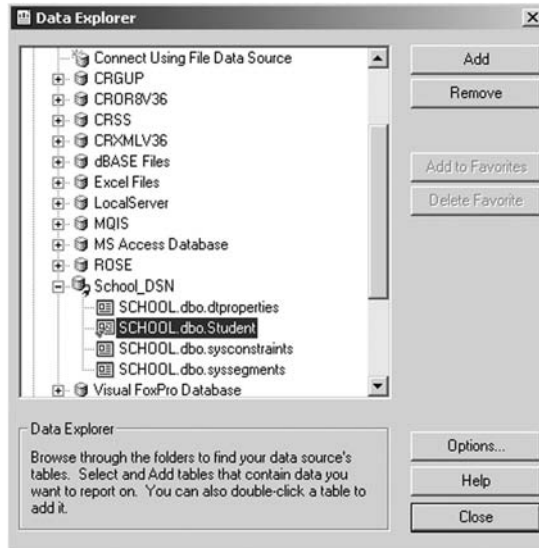


FIGURE 25.18

### 25.4.1 Creation of Reports Using an SQL Server 2000 Database

Launch Crystal Reports.

To create a new report select the option **As a Blank Report** and click **OK**.

In the Data Explorer Window double-click **More Data Sources**→**OLE DB**→**Make New Connection**.

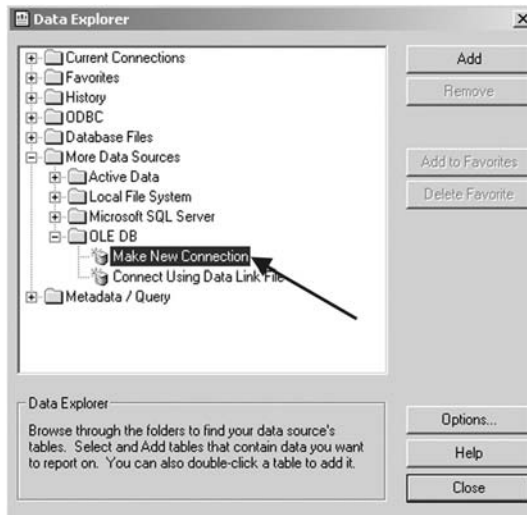


FIGURE 25.19

Now the Data Link Properties window will be displayed.

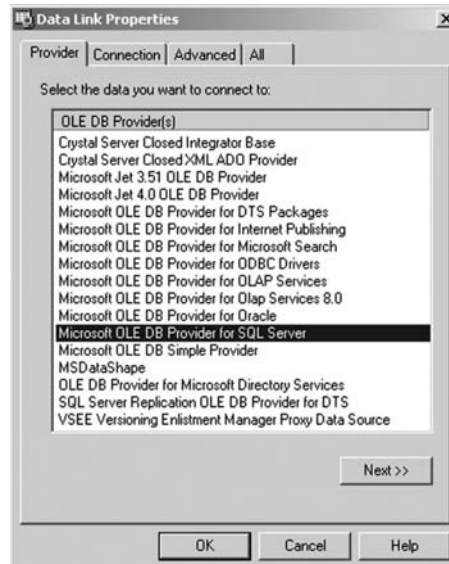


FIGURE 25.20

From Provider tab select **Microsoft OLE DB Provider for SQL Server**. Click **Next**.

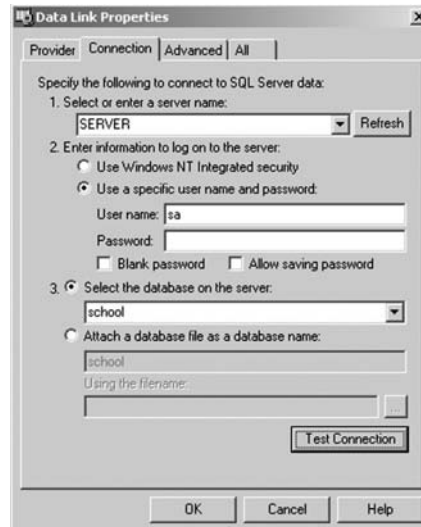


FIGURE 25.21

Go to the Connection tab:

1. Select server name
2. Select login type
3. Select the database name on the Server (in our case **School**)

Click the **Test connection** button. The **Test Connection Succeeded** Message Box will be displayed.

Click **OK**.

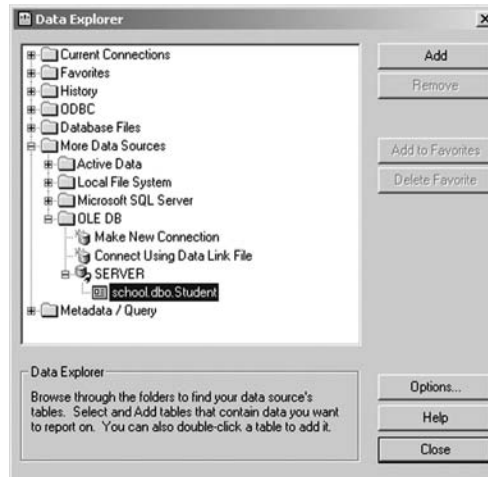


FIGURE 25.22

In the above displayed window click **Add** and then click **Close**.

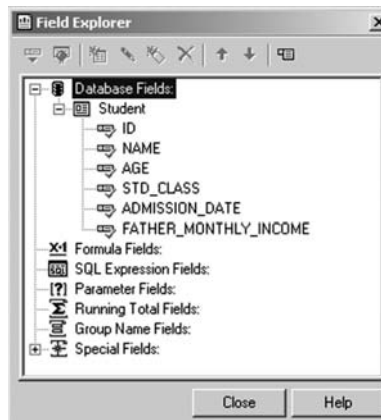


FIGURE 25.23

The Field Explorer window will be displayed.

Now drag and drop the student field values on the report form.

Select report title, page number, and other required items from the special field and drop them in the report.

Click **Refresh** to view the report with data.

| ID | NAME       | AGE | STD. CLASS | ADMISSION DATE | FATHER MONTHLY INCOME |
|----|------------|-----|------------|----------------|-----------------------|
| 1  | ARVIND     | 20  | 11         | 2-Aug-2004     | 20,000                |
| 2  | BHARAT     | 21  | 12         | 9-Sep-2003     | 23,000                |
| 3  | CHANDAR    | 22  | 9          | 4-Jul-2002     | 12,000                |
| 4  | DHARMENDRA | 19  | 11         | 4-Sep-2005     | 27,000                |
| 5  | FARHAN     | 20  | 12         | 5-Sep-2002     | 23,000                |
| 6  | HASAN      | 21  | 12         | 9-Jul-2001     | 12,500                |
| 7  | INDRAJEET  | 20  | 11         | 8-Sep-2002     | 27,000                |
| 8  | JA SNEET   | 19  | 11         | 9-Mar-2005     | 28,000                |
| 9  | KAMALJEET  | 21  | 12         | 9-Feb-2003     | 12,000                |
| 10 | MANISH     | 19  | 10         | 5-Feb-2003     | 16,000                |
| 11 | NAGENDRA   | 20  | 11         | 4-Sep-2005     | 12,900                |

FIGURE 25.24

### 25.4.2 Inserting Formulas in the Report

Right-click the Formula Field in the Field Explorer window:

Click the **New** Option.

Enter the formula name.

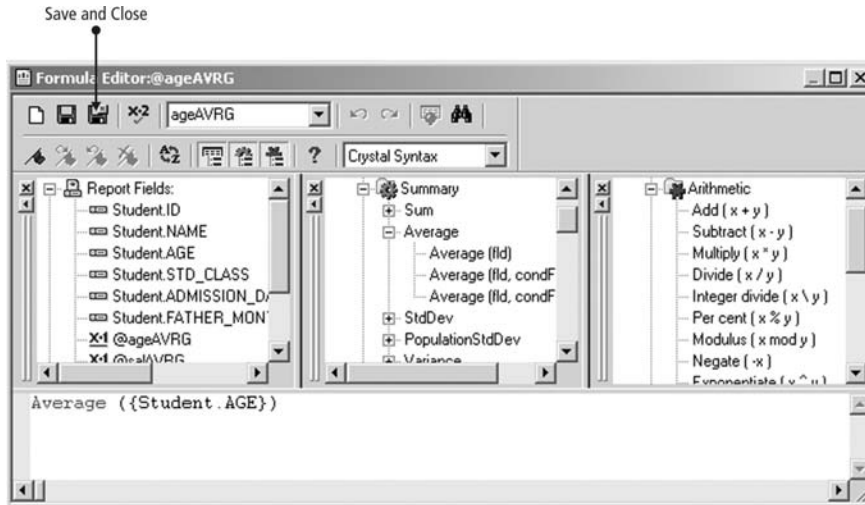


FIGURE 25.25

Insert the desired formula. We have used the Average of Student's Age. Click **Save and Close**.

Now the Field Explorer window will be displayed with your formula. Drag and drop this formula on the desired place in your report.

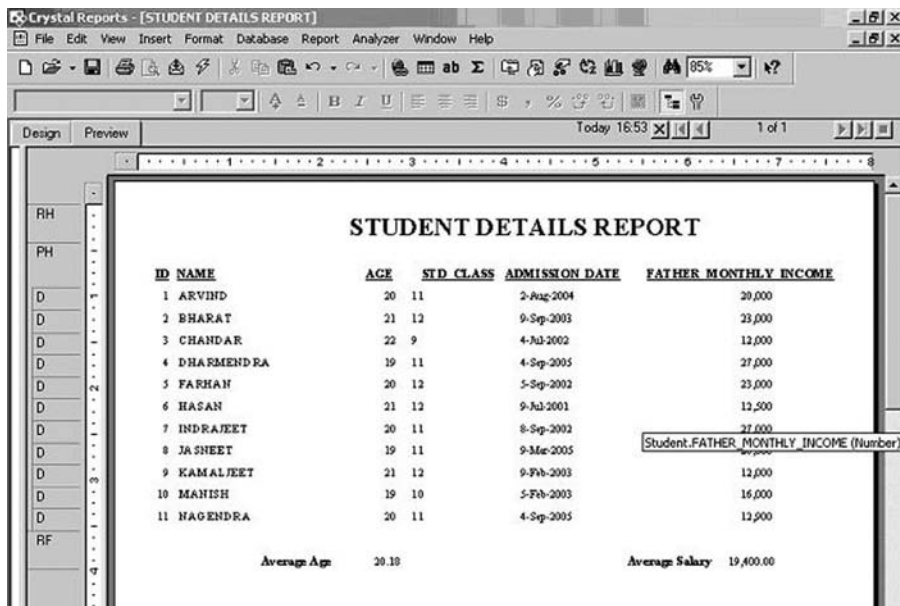


FIGURE 25.26

### 25.4.3 Grouping of Fields

Select the desired report.

Select Insert→Summary.

Insert the formula in the first combo box.

Select grouping field in the second combo box and click OK.

Drag and drop this formula in the desired place in your report. (See Figure 25.27.)

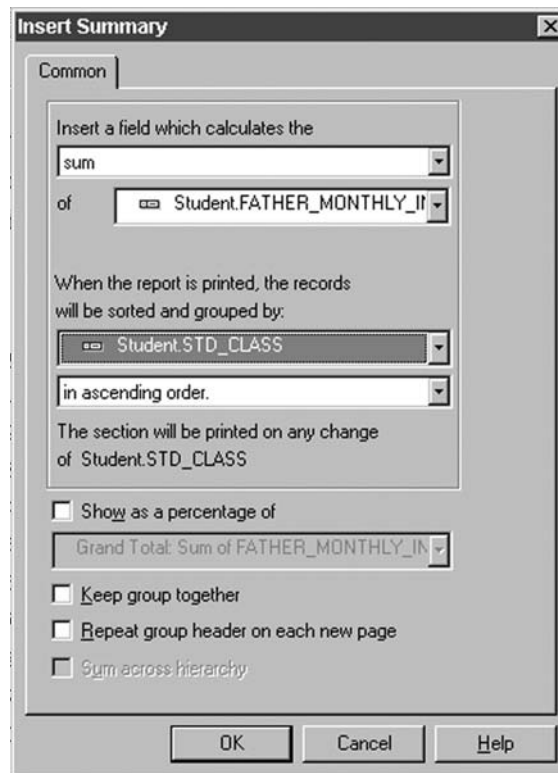


FIGURE 25.27

### 25.4.4 Connecting Crystal Reports with Visual Basic

Start Visual Basic

Select New→Standard Exe→Open.

Go to Project→Component.

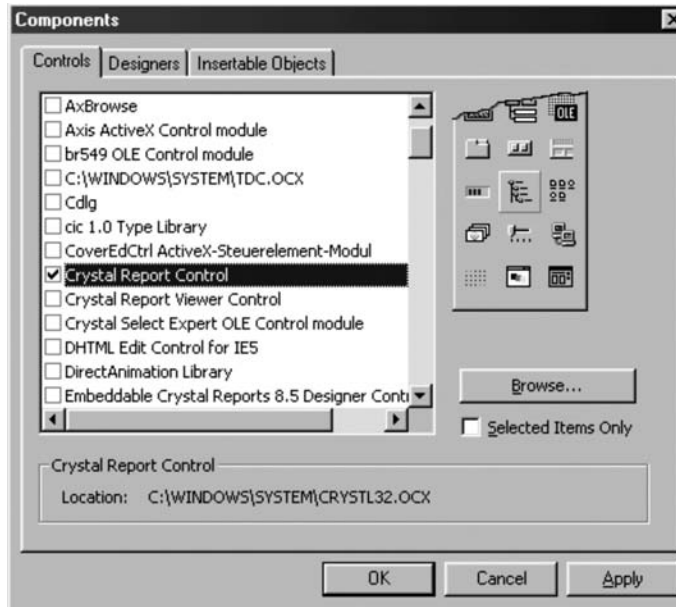


FIGURE 25.28

In the Components tab select the **Crystal Report Control** component and then click **Apply** and **OK**. The Crystal Report component will be displayed in the Tool Box. Double-click this component to add it in your form.

Write the following code in the command button click event:

```
Private Sub Command1_Click ()
 'PHYSICAL PATH OF THE REPORT
 1. CrystalReport1.ReportFileName=App.Path &
 "\STUDENTDETAILSREPORT.rpt"

 'DESIRED FORMULA TO FILTER INFORMATION FROM DATABASE
 2. CrystalReport1.SelectionFormula = "{STUDENT.ADMISSION_DATE}
 >=#" & DTPicker1 & " # and {STUDENT.ADMISSION_DATE}<=#" &
 DTPicker2 & "#"

 'RETRIEVE FILTERED INFORMATION
 3. CrystalReport1.RetrieveDataFiles

 'ASSIGNING REPORT POSITION ON WINDOW
 4. CrystalReport1.WindowState = crptMaximized

 'ASSIGNING REPORT DESTINATION OR WINDOW TO DISPLAY THE
 REPORT
```



5. CrystalReport1.Destination = crptTowindow  
'SHOWING REPORT
6. CrystalReport1.Action = 1  
End Sub

| ID | NAME        | AGE   | STD. CLASS | ADMISSION DATE | FATHER MONTHLY INCOME |
|----|-------------|-------|------------|----------------|-----------------------|
| 10 | MANISH      | 19    | 10         | 5-Feb-2003     | 16,000                |
|    |             |       |            | SUM            | 16,000.00             |
|    |             |       |            | AVERAGE        | 16,000.00             |
| 11 | NAGENDRA    | 20    | 11         | 4-Sep-2005     | 12,900                |
| 1  | ARVIND      | 20    | 11         | 2-Aug-2004     | 20,000                |
| 4  | DHARMENDRA  | 19    | 11         | 4-Sep-2005     | 27,000                |
| 7  | INDRAJEET   | 20    | 11         | 8-Sep-2002     | 27,000                |
| 8  | JASNEET     | 19    | 11         | 9-Mar-2005     | 28,000                |
|    |             |       |            | SUM            | 114,900.00            |
|    |             |       |            | AVERAGE        | 22,980.00             |
| 9  | KAMALJEET   | 21    | 12         | 9-Feb-2003     | 12,000                |
| 5  | FARHAN      | 20    | 12         | 5-Sep-2002     | 23,000                |
| 2  | BHARAT      | 21    | 12         | 9-Sep-2003     | 23,000                |
|    |             |       |            | SUM            | 58,000.00             |
|    |             |       |            | AVERAGE        | 19,333.33             |
| 3  | CHANDAR     | 22    | 9          | 4-Jul-2002     | 12,000                |
|    |             |       |            | SUM            | 12,000.00             |
|    |             |       |            | AVERAGE        | 12,000.00             |
|    | Average Age | 20.10 |            | Average Salary | 20,090.00             |

FIGURE 25.29 Report View Generated by Crystal Reports

# Chapter 26

## ERROR HANDLING

Error handling is one of the most important parts of software development. An application with bugs may produce the incorrect output, termination of an application, or a user's irritation. We will discuss some methods and precautions in this chapter that will help you to develop an error-free application.

### 26.1 KEY HANDLING

---

To ensure that proper data is being inserted, lock other keys that may cause errors.

For example, if you do not want to allow the user to enter any alphanumeric characters in the text box where you want only numeric characters, you can lock the character keys. To lock the keys of the keyboard you must know the ASCII value of each key.

| Key | ASCII Value | VB Constant |
|-----|-------------|-------------|
| A   | 65          | VbKeyA      |
| B   | 66          | VbKeyB      |
| C   | 67          | VbKeyC      |

|   |     |        |
|---|-----|--------|
| D | 68  | VbKeyD |
| E | 69  | VbKeyE |
| F | 70  | VbKeyF |
| G | 71  | VbKeyG |
| H | 72  | VbKeyH |
| I | 73  | VbKeyI |
| J | 74  | VbKeyJ |
| K | 75  | VbKeyK |
| L | 76  | VbKeyL |
| M | 77  | VbKeyM |
| N | 78  | VbKeyN |
| O | 79  | VbKeyO |
| P | 80  | VbKeyP |
| Q | 81  | VbKeyQ |
| R | 82  | VbKeyR |
| S | 83  | VbKeyS |
| T | 84  | VbKeyT |
| U | 85  | VbKeyU |
| V | 86  | VbKeyV |
| W | 87  | VbKeyW |
| X | 88  | VbKeyX |
| Y | 89  | VbKeyY |
| Z | 90  | VbKeyZ |
| a | 97  |        |
| b | 98  |        |
| c | 99  |        |
| d | 100 |        |
| e | 101 |        |

|   |     |        |
|---|-----|--------|
| f | 102 |        |
| g | 103 |        |
| h | 104 |        |
| i | 105 |        |
| j | 106 |        |
| k | 107 |        |
| l | 108 |        |
| m | 109 |        |
| n | 110 |        |
| o | 111 |        |
| p | 112 |        |
| q | 113 |        |
| r | 114 |        |
| s | 115 |        |
| t | 116 |        |
| u | 117 |        |
| v | 118 |        |
| w | 119 |        |
| x | 120 |        |
| y | 121 |        |
| z | 122 |        |
| 0 | 48  | VbKey0 |
| 1 | 49  | VbKey1 |
| 2 | 50  | VbKey2 |
| 3 | 51  | VbKey3 |
| 4 | 52  | VbKey4 |
| 5 | 53  | VbKey5 |
| 6 | 54  | VbKey6 |

|                 |     |              |
|-----------------|-----|--------------|
| 7               | 55  | VbKey7       |
| 8               | 56  | VbKey8       |
| 9               | 57  | VbKey9       |
| ←               | 37  | VbKeyLeft    |
| ↑               | 38  | VbKeyUp      |
| →               | 39  | VbKeyRight   |
| ↓               | 40  | VbKeyDown    |
| Back            | 8   | VbKeyBack    |
| Ctrl            | 17  | VbKeyControl |
| . (Decimal/Dot) | 46  |              |
| /               | 47  | VbKeyDivide  |
| #               | 35  |              |
| Esc             | 27  | VbKeyEscape  |
| ‘               | 96  |              |
| ~               | 126 |              |
| !               | 33  |              |
| @               | 64  |              |
| \$              | 36  |              |
| %               | 37  |              |
| ^               | 94  |              |
| &               | 38  |              |
| *               | 42  |              |
| (               | 40  |              |
| )               | 41  |              |
| -               | 45  |              |
| _ (Under Score) | 95  |              |
| =               | 61  |              |
| +               | 43  |              |

|             |     |              |
|-------------|-----|--------------|
| \           | 92  |              |
|             | 124 |              |
| ,           | 44  |              |
| <           | 60  |              |
| >           | 62  |              |
| ?           | 63  |              |
| <Enter>     | 13  | VbKeyReturn  |
| Num Lock    | 144 | VbKeyNumLock |
| “           | 34  |              |
| ,           | 39  |              |
| :           | 58  |              |
| ;           | 59  |              |
| Pause       | 19  | VbKeyPause   |
| Scroll Lock | 145 | VbScrollLock |
| Shift       | 16  | VbKeyShift   |
| <Space>     | 32  | VbKeySpace   |
| Tab         | 9   | VbKeyTab     |

## 26.2 KEY LOCKING AT KEY PRESS EVENT

---

Write the following code for different types of validation at the key press event of a text box or combo box.

### 1. Validation for Integers

#### Syntax

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
1. If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or (KeyAscii =
 vbKeyBack) Then
2. Exit Sub
3. Else
4. KeyAscii = 0
```

```

5. End If
 End Sub

```

## 2. Validation for Characters

### Syntax

```

Private Sub Text1_KeyPress(KeyAscii As Integer)
1. If (KeyAscii >= 65 And KeyAscii <= 90) Or (KeyAscii >= 97 And
 KeyAscii <= 122) Or (KeyAscii = vbKeyBack) Or (KeyAscii =
 vbKeySpace) Then
2. Exit Sub
3. Else
4. KeyAscii = 0
5. End If
 End Sub

```

## 3. Validation for Float Numbers

### Syntax

```

Private Sub Text1_KeyPress(KeyAscii As Integer)
1. If (KeyAscii >= vbKey0 And KeyAscii <= vbKey9) Or (KeyAscii =
 vbKeyBack) Or (KeyAscii = 46)
2. Then
3. Exit Sub
4. Else
5. KeyAscii = 0
6. End If
 End Sub

```

## 26.3 OTHER ERROR-HANDLING METHODS

---

### On Error GoTo

When an error occurs in an application, the application may terminate. To overcome this problem use the 'On Error GoTo —' statement and display a proper error message with an error number. At the start of the procedure write 'On Error GoTo <label name>' and at the end of the procedure define the label.

### Syntax

```

Private Sub Command1_Click()
1. On Error GoTo err1:
 -

```

- (Code)
- 
- 
- 2. err1:  
    'If any error occurs, the procedure returns an error number greater than zero otherwise less than zero
- 3. If Err.Number <> 0 Then  
    X = MsgBox("Error number : " & Err.Number & vbCrLf & vbCrLf & " " & Err.Description, vbExclamation)
- 4. Exit Sub
- 5. End If
- End Sub

### On Error Resume Next

You may use the On Error Resume Next statement at the start of the procedure to ignore the error, if it does not produce an incorrect result.

#### Syntax

- ```
Private Sub Command1_Click()
1. On Error Resume Next
-
- (Code)
-
-
End Sub
```

26.4 SOME COMMON ERRORS

1. Not a valid path

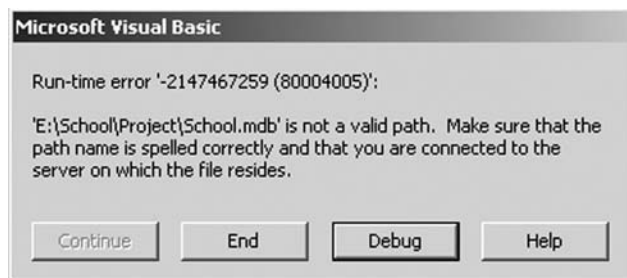


FIGURE 26.1

The path given to locate the file is incorrect or does not exist.

2. Object required

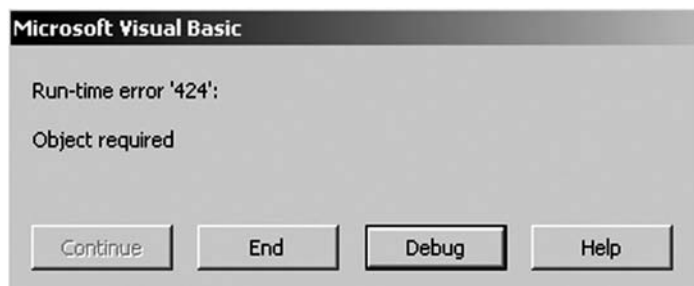


FIGURE 26.2

Object or control is missing or the object name is misspelled.

3. Expected: end of statement



FIGURE 26.3

String is not properly terminated or there is some required character missing.

4. Could not find file

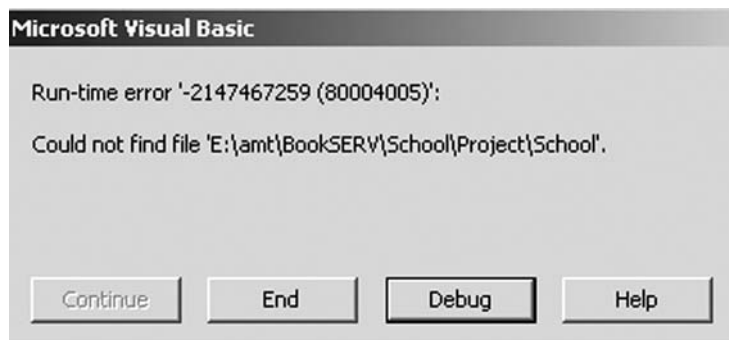


FIGURE 26.4

File is missing in the given path or the filename is misspelled.

5. Provider cannot be found

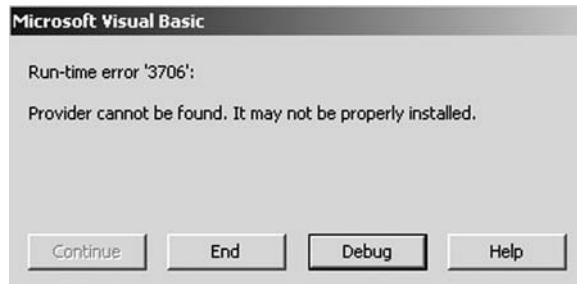


FIGURE 26.5

Provider is specified incorrectly in the connection string or you have not installed the database.

6. Syntax error



FIGURE 26.6

There is some syntax error in the statement. Check the spellings of all the commands in the given line.

7. Data type mismatch in criteria expression



FIGURE 26.7

The data type of the value entered is not the same as the field type of that column in the table.

8. Invalid property value

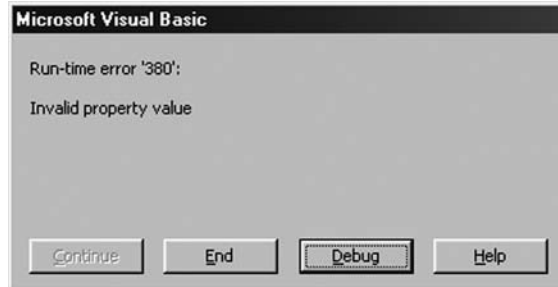


FIGURE 26.8

An incorrect property has been set for a control or object.

9. Cannot find input table or query

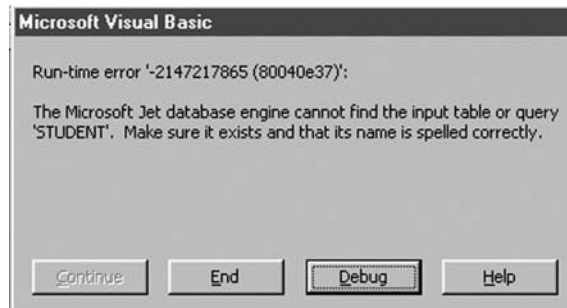


FIGURE 26.9

Table does not exist in the database or you have misspelled the table name.

10. Operation is not allowed when the object is open



FIGURE 26.10

You are trying to reopen the connection, recordset, or any other object before closing it.

11. Operation is not allowed when the object is closed



FIGURE 26.11

You are trying to reclose the connection, recordset, or any other object which is already closed.

12. Item cannot be found in the collection



FIGURE 26.12

The field name associated with the recordset does not exist or the field name is misspelled.

13. Number of query values and destination fields are not the same

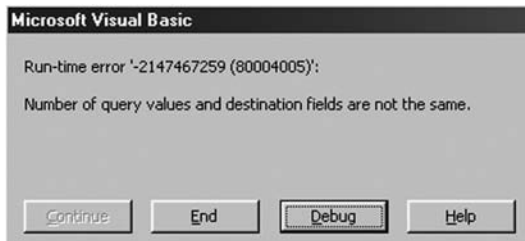


FIGURE 26.13

The number of columns that are inserted in the table are more or less than the number of columns in the table.

14. Syntax error (missing operator) in query expression

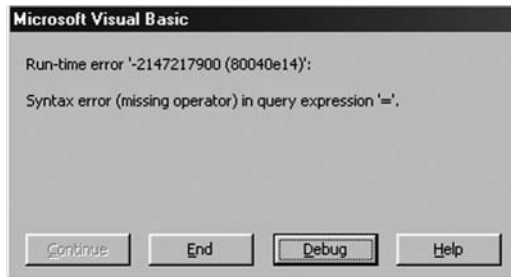


FIGURE 26.14

There is an operator missing in the statement or there is an invalid use of the operator.

15. Duplicate declaration in current scope

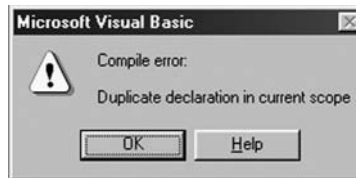


FIGURE 26.15

The variable has been declared twice in the same procedure.

16. Undefined function in expression

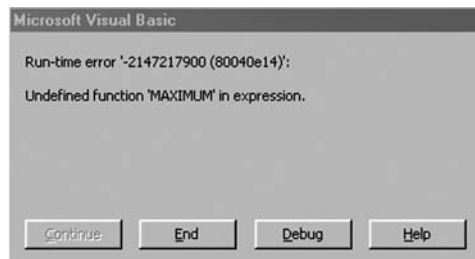


FIGURE 26.16

The function used is not defined in the module or it has been declared private in another module.

17. Invalid use of null

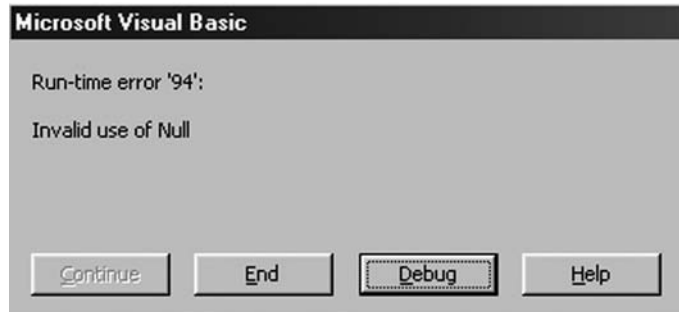


FIGURE 26.17

There is an operation on a null value.

18. Either BOF or EOF is true or the current record has been deleted

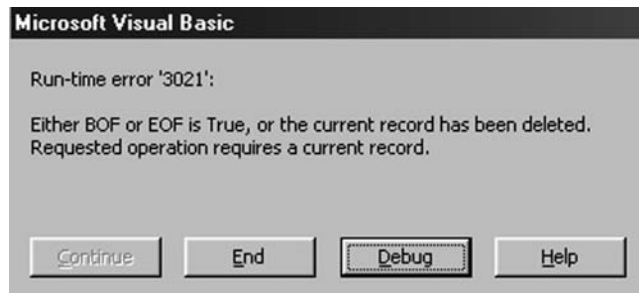


FIGURE 26.18

The operation you are performing does not have a record.

19. Syntax Error in INSERT INTO statement



FIGURE 26.19

You have missed one or both the brackets in the INSERT INTO statement or the viable or control name through which you are inserting the value is misspelled or does not exist.

20. Expected expression

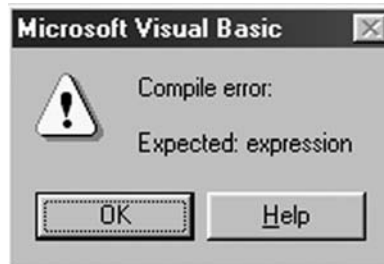


FIGURE 26.20

This error occurs when there is an expression or command missing in the statement line.

21. Field cannot be a zero-length string

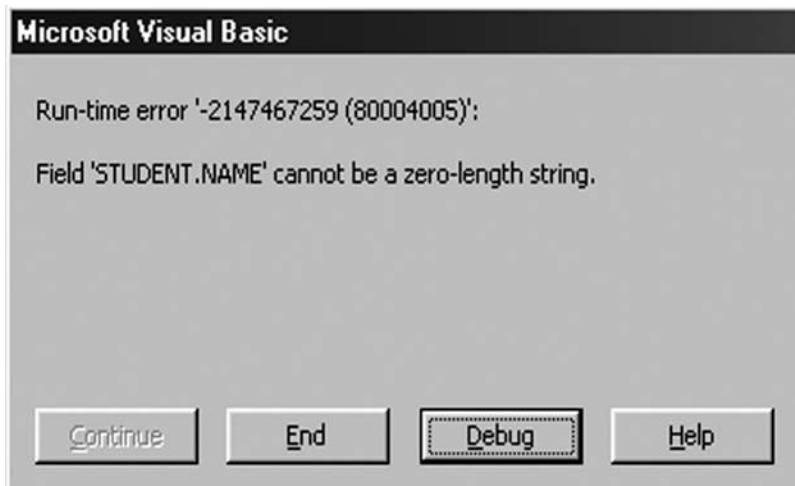


FIGURE 26.21

Do not send a null value in the table; either enter a value or set the Allow Null property of the field to true.

22. No value given for one or more required parameters

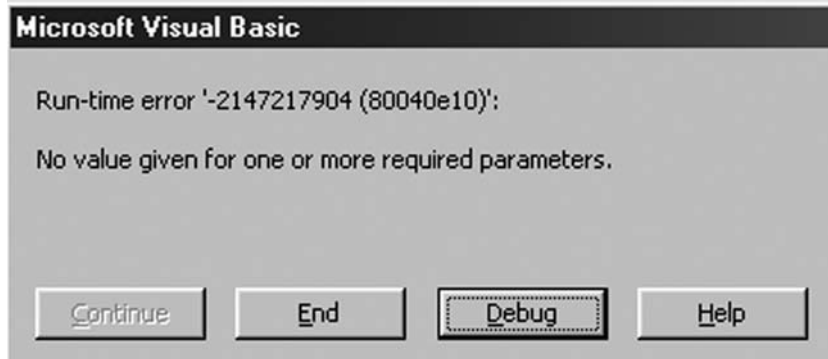


FIGURE 26.22

There is some spelling mistake in the fieldname given in the statement or the field does not exist in the table.

23. Type mismatch

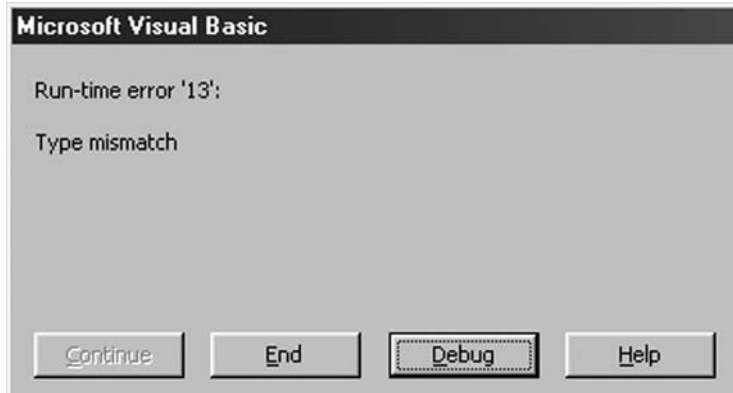


FIGURE 26.23

The type of variable and the type of value you are assigning to it are different.

26.5 PRECAUTIONS

1. Do not allow users to enter invalid characters.
2. Set the **MaxLength** property of the text box equal to the field size, so that the user cannot enter data larger than the field length.

3. Use the **On Error GoTo** statement to avoid termination of an application on error.
4. Use the **App.path** method instead of defining a fixed path for database connectivity in MS Access.
5. The data type of the input data should be the same as defined in the table for that field.
6. Do not forget to close the **recordset** after opening and performing the operation.
7. Close the **data environment** at the terminate event of the data report.
8. Keep all the files and sub-folders related to the same project in a main folder.
9. For proper connection, first install the database (Oracle, Access, SQL Server) before installing Visual Studio.
10. Use proper explanatory remarks while writing the program. This will help you to find the error during debugging.
11. Thoroughly test your project before implementation.

Chapter 27

CREATING THE SETUP PACKAGE

To distribute your application and run your application on other systems it is necessary that all the files that are linked to the application and all dll and supporting files that are used in the application be present in other systems also. There are two ways to run your application on another system. First, is to install Visual Basic 6.0 on that system and copy your project, but this will take much hard disk space and time to install. The second and better way is to create a setup of the application and run the setup file. This will take less space and time to install because the setup file contains only the required files that are necessary to run the application.

27.1 HOW TO CREATE A SETUP

Follow the given steps to create a setup of an application.

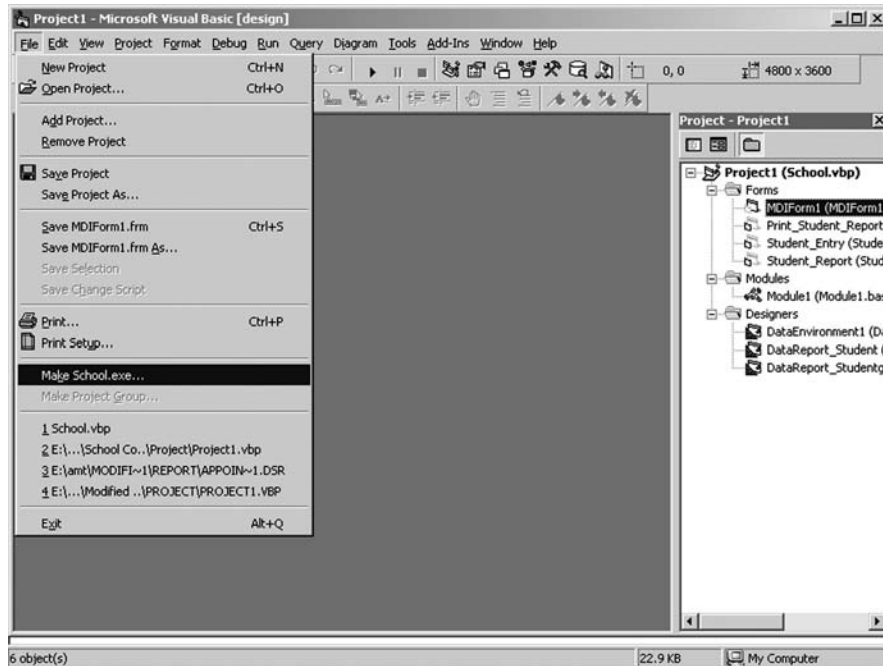


FIGURE 27.1

1. Create the .exe (executable) file of the project and save it in the same folder where the project file is present. To create the .exe file, open the SCHOOL project and go to the menu **File** → **Make School.exe** option. This will compile the project and create the School.exe file in the selected folder. It's good practice to create the .exe file before creating the setup package because during compilation an error may occur that you can rectify.

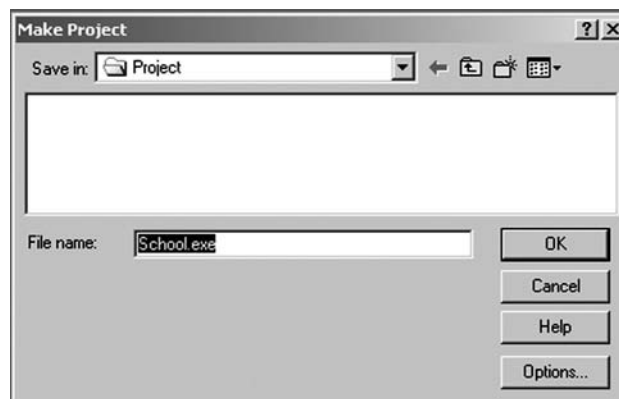


FIGURE 27.2

2. Go to **Start** → **Programs** → **Microsoft Visual Studio 6.0** → **Microsoft Visual Studio 6.0 Tools** → **Package & Deployment Wizard**.

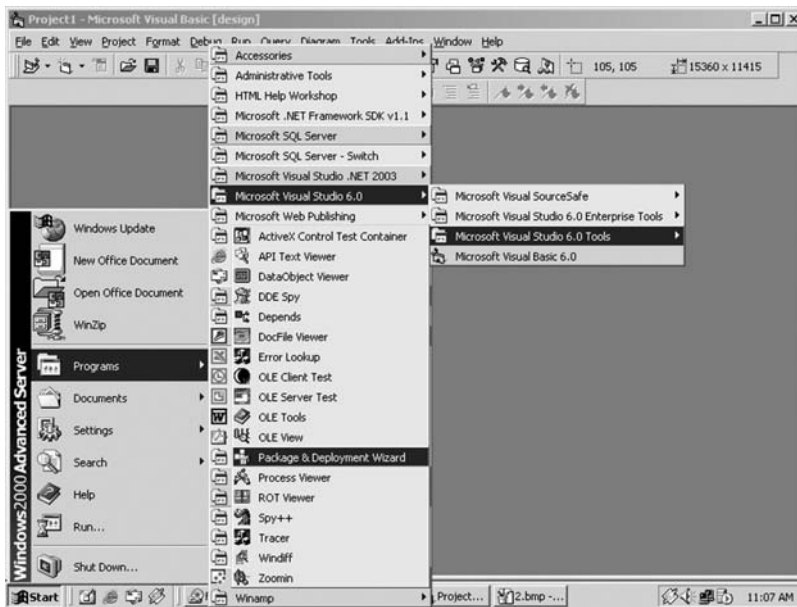


FIGURE 27.3

3. Browse for the project file **School.vbp** and click the **Package** button. If this gives the message to recompile the project, choose the **No** button to use the existing executable file.



FIGURE 27.4

4. In the next screen, select Standard Setup Package from the list and click **Next**.



FIGURE 27.5

5. Select the folder where you want to create the setup files and folder.



FIGURE 27.6

6. If you have no dependency information about the files, click **OK** without selecting its checkbox.

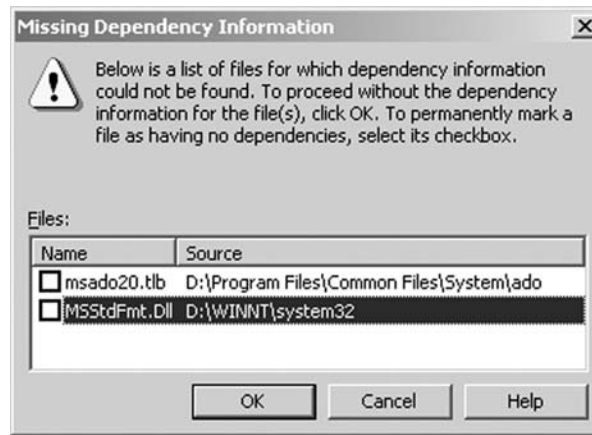


FIGURE 27.7

7. Click **OK** to proceed with out-of-date dependency information of the files (see Figure 27.8).

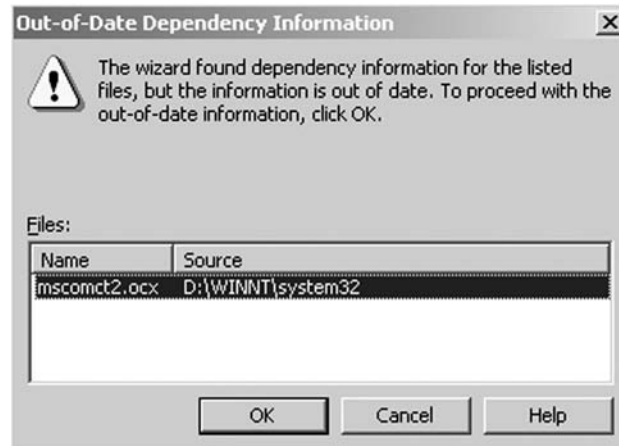


FIGURE 27.8

8. Add the files which you want to include in your package. All the *.dll and *.ocx files are automatically added. You have to add only database files (*.mdb), text files (*.Txt), document files (*.doc), image files (*.jpg), or other files which have not been added in the list. In this case add only the School.mdb file.

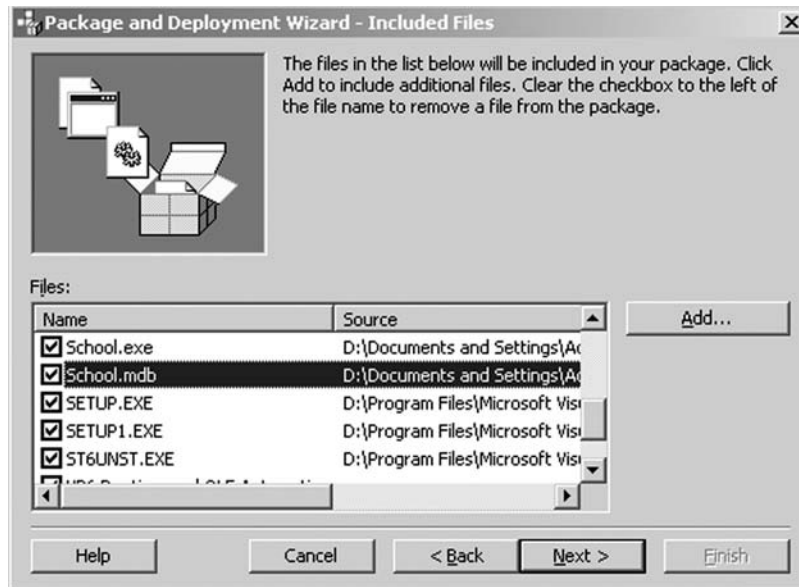


FIGURE 27.9

- Choose the option single cab or multiple cab. The single cab option is used if you want to copy the package on a CD, pen drive, or a large storage device. Multiple cab is used if you want to copy the package on a floppy.

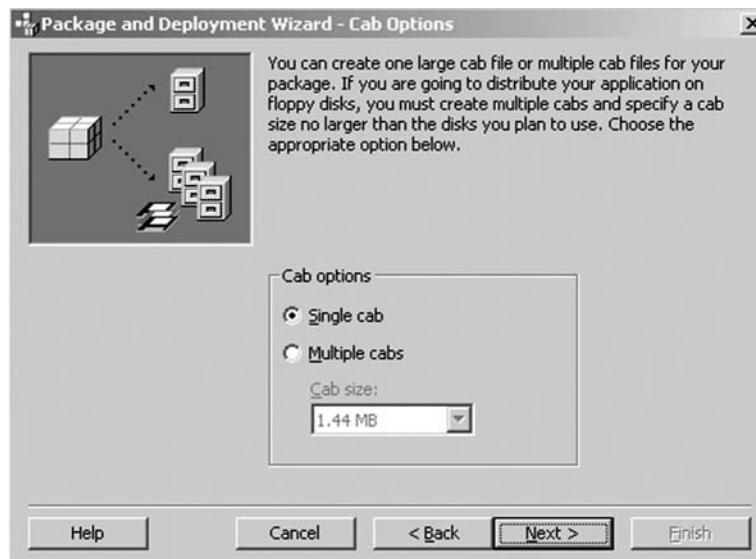


FIGURE 27.10

10. Give the installation title School Management System and click **Next**.



FIGURE 27.11

11. In the Start Menu Items window, you can add a new group/item, remove an existing group/item, or rename a group/item. If a group or item is added, select the item and click the Properties button to give the .exe filename to which this new item will be attached.

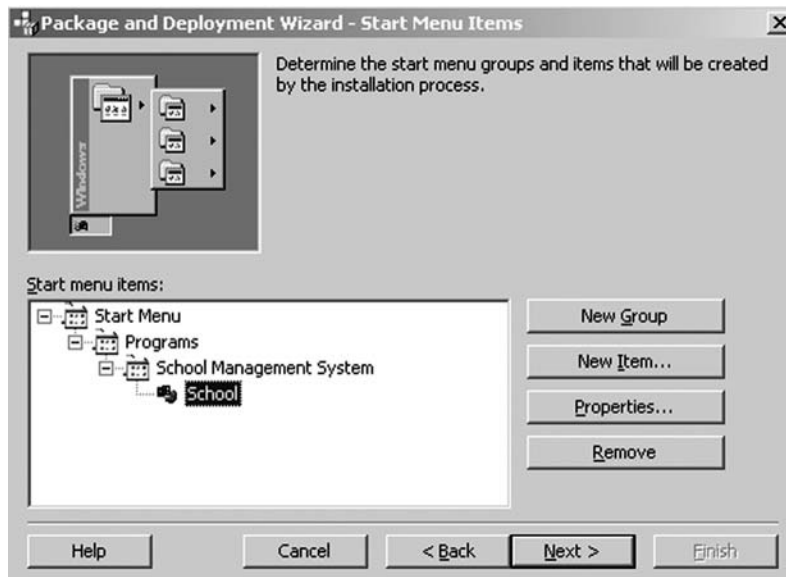


FIGURE 27.12

12. In the Install Locations window, you can modify the install location for each of the listed files by changing the macro assigned to the file. If desired, you can add sub-folder information to the end of a macro. For example, & (AppPath)\Database. Choose the file you want to modify, then change the information on the install location column.
13. Check the files if you want to install as shared. Click **Next** when the package is installed and the checked files will be installed as shared files. The shared files may be used by more than one program. They are only removed if every program which uses them is removed (see Figure 27.13).

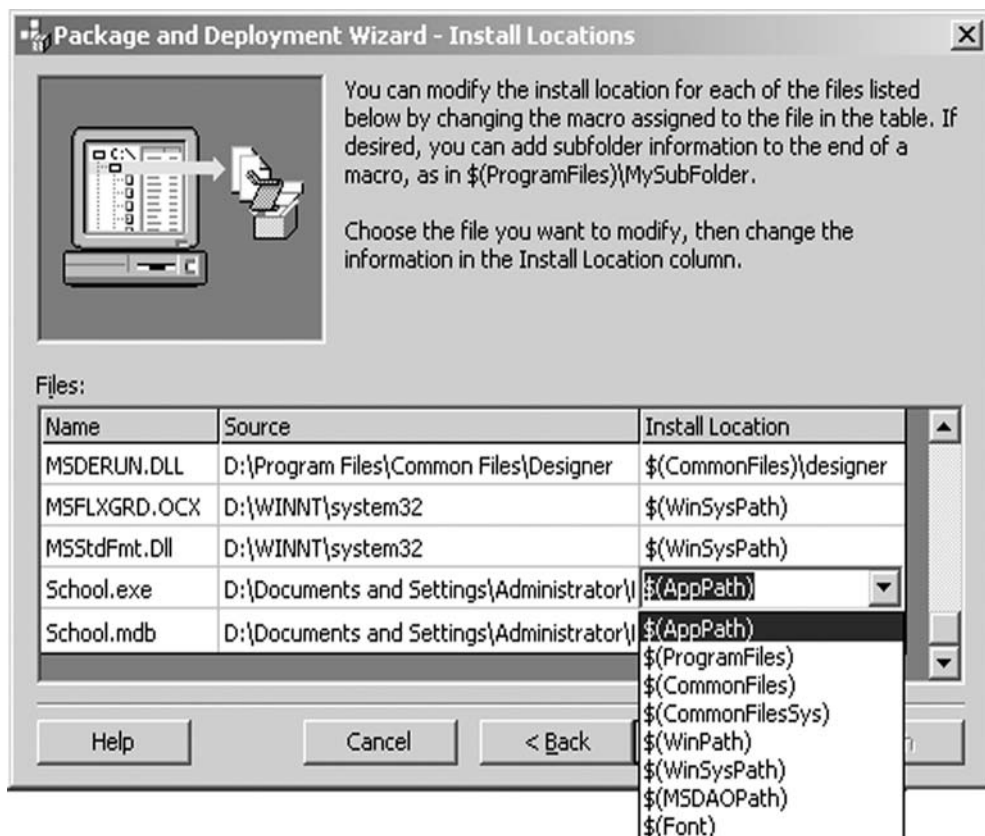


FIGURE 27.13

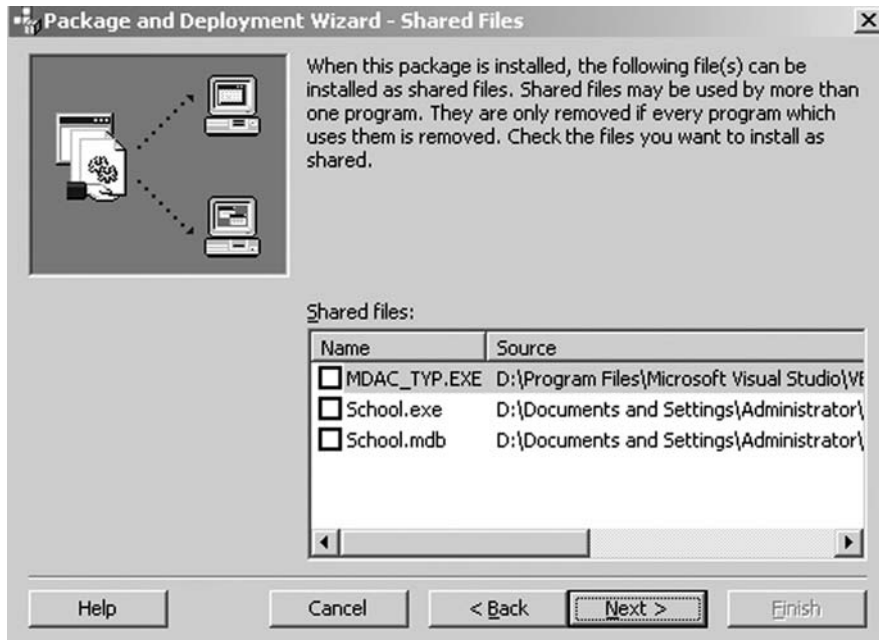


FIGURE 27.14

14. Give the script the name School Management System. The script name is used to save the settings that you have made to create this package. Click **Finish**.



FIGURE 27.15

15. After generating the cab files a Packaging Report window will be displayed to save the packaging report. If you want to save this report click the **Save Report** button, otherwise click **Close**.

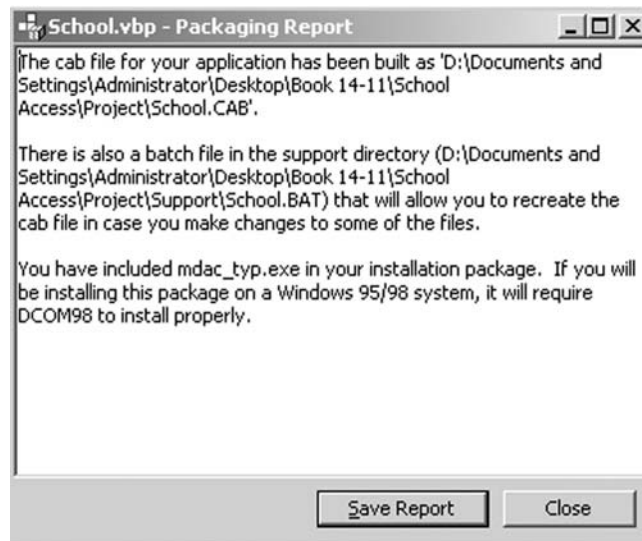


FIGURE 27.16

16. The setup and cab files created will be displayed in the package folder as given in the figure.

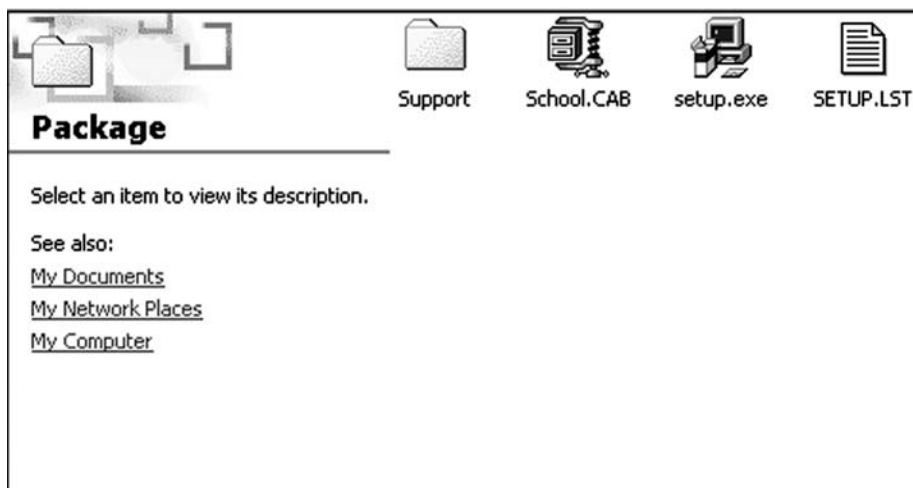


FIGURE 27.17

REFERENCES

1. Software Engineering — Roger Pressman
2. Mastering in Visual Basic — Evangelos Petrouteous
3. Visual Basic Objects — John Smiley
4. Visual Basic Databases — John Smiley
5. Visual Basic Examples — John Smiley
6. Visual Basic Developer's Guide to the Win32API — Steve Brown
7. Ground up Visual Basic 6.0 — Garrey Connell
8. Database Developers Guide with VB 6.0 — Roger Jennings
9. Mastering Access 2000 — Simpson
10. Teach Yourself Access for windows 95 — Seigel
11. Access 2000. No Experience Required — Robinson
12. Access 2000 from A TO Z — Julie Kelly, Stephen L. Nelson
13. Oracle 8 — Evan Bayross
14. Mastering Crystal Report — BPB

INDEX

A

- Abstraction, 121
- Abstraction level, 239
- Acceptance testing, 40, 173
- Access database wizard, 320-321
- Accumulating errors during software, 96
- ActiveX control, 266
- ActiveX DLL, 266
- ActiveX document DLL, 266
- ActiveX document EXE, 266
- ActiveX EXE, 266
- Adaptability, 58, 105
- Adaptive maintenance, 35, 40, 203
- Adding a primary key in an existing table, 345-346
- Addins in VB, 268
- Advantages of 4GT, 256
- Advantages of black-box testing, 178
- Advantages of CASE tools, 234-235
- Advantages of decision tables, 69
- Advantages of flowcharts, 134
- Advantages of function points, 155
- Advantages of iterative-enhancement model, 48
- Advantages of modular systems, 126-127
- Advantages of prototyping models, 43
- Advantages of pseudo-code, 130
- Advantages of spiral model, 46
- Advantages of structural/white-box testing, 175
- Advantages of structured programming, 255
- Advantages of waterfall model, 41
- Algorithms, 263
- Allow zero length, 327
- Alpha testing, 40, 172
- Alternate key (in a database), 317
- Ambiguity, 75
- Ambiguous decision table, 68
- Analysis, 134
- Analysis tools, 234
- Analyzability, 105
- Application areas of reverse engineering, 239
- Application generation, 50
- Application software, 5
- Architectural design, 123-125
 - Objectives of architectural design, 124-125
- Architecture of CASE environment, 224-226
 - User interface, 224

- Tools-management services (tools-set), 225
- Object-management system (OMS), 225
- Repository, 226
- Arrows (in structure charts), 128
- Artificial-intelligence software, 17
- Attaching a database, 376-378
- Attributes, 81-82, 250
- Attributes of effective software metrics, 150-151
- Authentication, 386-387
- AVAIL (Availability), 111

B

- Backtracking, 187
- Backward traceability, 78
- BASIC, 265
- Basic building blocks of a structure chart, 128-129
- Basic COCOMO model, 212-214
- Basic elements of a decision table, 67
- Bath-tub curve, 12-13
- Belady and Lehman model, 205-206
- Benefits of ISO-9000 certification, 102
- Benefits of SRS, 74
- Beta testing, 40, 172
- Binary relationship, 80
- Black-box testing example, 177
- Blank Access database, 321-324
- Boehm model, 206-207
- Bottom-up design, 121-122
- Bottom-up integration testing, 170-170
- Boxes (in structure charts), 128
- Branching, 253
- Breadth-first integration, 169
- Brute-force debugging, 187

- Building blocks for CASE, 226
 - Environment architecture, 226
 - Portability services, 226
 - Integration framework, 226
 - CASE tools, 226
- Builds, 171
- Business modeling, 49-50
- Business software, 16-17

C

- Calling a report in Visual Basic, 436
- Candidate key (in a database), 317
- Capability maturity model (SEI-CMM), 96-99, 106
- Caption in MS Access, 327
- Cardinality, 80
- CASE building blocks, 226
- CASE classification, 231-232
- CASE database, 226
- CASE repository, 226, 229-231
- CASE support in software life-cycle, 227-228
 - Prototyping support, 227
 - Structured analysis and design, 227
 - Code generation, 227-228
 - Test CASE generator, 228
- CASE tool support, 60
- CASE tools, 226, 232-235
- Categories of black-box testing, 176-177
- Categories of CASE tools, 233-234
 - Vertical CASE tools, 233
 - Horizontal CASE tools, 233
 - Upper CASE tools/Front-end CASE tools, 233-234
 - Lower CASE or Back-end tools, 234

- Cross life-cycle CASE or integrated tools, 234
 - Reverse-engineering tools, 234
- Categories of maintenance, 203
 - Corrective maintenance, 203
 - Adaptive maintenance, 203
 - Perfective maintenance, 203
 - Preventative maintenance, 203
- Categories of metrics, 149
- Cause elimination, 187
- CCB, 195
- Changeability, 105
- Change-control process, 197-198
- Changing management processes, 60
- Characteristics of bugs, 189
- Characteristics of SRS, 78-79
 - Correctness, 78
 - Completeness, 78
 - Unambiguous, 78
 - Verifiable, 78
 - Modifiable, 78
 - Traceable, 78
 - Consistency, 78-79
 - Traceability, 79
 - Clarity, 79
 - Feasibility, 79
- Check boxes, 283-285
- Checklist for code inspections, 185
- Choosing components and options, 352
- Clarity, 79
- Classes of requirements management, 59
- Classes of software, 8-9
 - Generic software, 8-9
 - Customized software, 9
- Classification of failures, 109
- Classification of modules, 126
- Classification of software qualities, 89-90
- Clusters, 170-171
- Code defects, 251
- Code generation, 227-228
- Code generators, 234
- Code inspections, 184-186, 251
- Code reading, 251
- Code restructuring, 243
- Code reviews, 251-252
- Code walk-throughs, 183-184
- Code window, 271, 274
- Coding and module testing, 39
- Cohesion, 139-142
- Cohesion-strength of relation within modules, 139
- Combo box in VB, 287-288
- Command button, 282-283
- Common coupling, 138
- Common errors in computation, 166
- Communication, 134
- Communicational cohesion, 140-141
- Comparison of ISO-9000 certification and the SEI-CMM, 106-107
- Comparison of various process models, 50-51
- Compilation in Visual Basic, 268
 - Limitations, 268
- Compilation phase, 186
- Compilers and interpreters, 5
- Complete COCOMO model, 214-215
- Completeness, 69, 78, 118, 239
- Compliance, 104
- Components of SRS, 75-77
 - Functional requirements, 75-76
 - Performance requirements, 76
 - Design constraints, 76-77

- Standards compliance, 76
- Reliability and fault tolerance, 77
- Security, 77
- External interface requirements, 77
- Composite attribute, 81
- Composite key (in a database), 318
- Comprehensibility, 58
- Computation defects, 251
- Computer attributes, 214
- Computer science, 22-23
- Computer-aided software engineering (CASE), 223-235
- Concurrent-version control, 199
- Conditional searching, 339
- Configuration accounting, 195-197
- Configuration audit, 196-197
- Configuration control, 194-195
- Configuration control system, 194
- Configuration identification, 194
- Configuration-management environment, 201
- Configuration-management plan, 194
- Configuration-management team, 200
- Configuration-management tools, 194
- Conflict resolution, 57
- Conformance, 105
- Connecting Crystal Reports with Visual Basic, 462-464
- Connector symbol, 133
- Consistency, 78-79
- Constructive cost model (COCOMO), 211-215
 - Basic COCOMO model, 212-214
 - Intermediate COCOMO model, 214
 - Complete COCOMO model, 214-215
 - Phase-sensitive effort multipliers, 215
 - Three-level product hierarchy, 215
- Content coupling, 138
- Control constructs, 249
- Control coupling, 138
- Control events, 296
- Controls in Visual Basic, 273-276
 - Object window, 273
 - Code window, 274
 - Form properties, 274-276
- Control-flow arrows, 129
- Controlled objects, 194
- Controlling, 208
- Controlling changes during prototyping, 43
- Copy database wizard, 380
- Corrective maintenance, 35, 40, 203
- Correctness, 78, 92-94, 118
- Cosmetic changes, 43
- Cost-estimation process, 209
- Coupling and cohesion, 136-143
 - Coupling, 136-139
 - Factors affecting coupling between modules, 137
 - Types of couplings, 137-139
 - Cohesion, 139-142
 - Types of cohesion, 140-142
 - Relationship between coupling and cohesion, 142-143
- Creating a connection using DSN, 456-457
- Creating a connection with Visual Basic, 387-389
- Creating a database using the create database wizard in Enterprise Manager, 358
- Creating a database in MS Access 2000, 319-324
 - Access database wizard, 320-321
 - Blank Access database, 321-324
 - Datasheet view, 322
 - Table wizard, 322
 - Design view, 322
 - Import table, 322

- Link table, 322
- Creating a new user in Oracle, 331-332
- Creating the setup package, 481-490
 - How to create a setup, 481-490
- Creating reports using DSN of the SQL Server 2000 database, 451-456
- Creation of reports using an SQL Server 2000 database, 457-460
- Cross life-cycle CASE or integrated tools, 234
- Crystal reports, 447-464
 - Advantages over Visual Basic data reports, 448
 - Starting with Crystal Report 8.0, 448-451
 - Creating reports using DSN of the SQL Server 2000 database, 451-456
 - How to make a DSN, 451-456
 - Creating a connection using DSN, 456-457
 - Creation of reports using an SQL Server 2000 database, 457-460
 - Inserting formulas into the report, 460-461
 - Grouping of fields, 462
 - Connecting crystal reports with Visual Basic, 462-464
- Crystal Report Data Explorer, 449
- Crystal Report Gallery, 449
- Customized software, 9
- Cycle of design phase, 33
- Cyclomatic complexity, 157-159

D

- Data abstraction, 121
- Data abstraction modules, 126
- Data coupling, 137-138
- Data dictionary, 230
- Data environment and the connection in Visual Basic, 425-429
- Data export at runtime, 420
- Data export in Oracle, 346-347
- Data-flow diagrams, 62-67
 - Symbols uses for constructing DFDs, 62-63
 - Function symbol, 62
 - External entity, 62
 - Data-flow symbol, 62
 - Data-store symbol, 62-63
 - Output symbol, 63
 - Example DFD, 63-64
 - Levels of a DFD, 64-66
 - General guidelines and rules for constructing DFDs, 66-67
- Data import in Oracle, 347-348
- Data insertion in a table, 339
- Data integrity, 318
- Data links, 414-416
- Data modeling, 50
- Data operations and computation defects, 251
- Data project, 267
- Data re-engineering, 241
- Data report controls in Visual Basic, 433-435
- Data report creation in Visual Basic, 425
- Data report designing in Visual Basic, 430-433
- Data restructuring, 243-244
- Data selection, 339
- Data transformation services, 380-385
- Data types, 271-272, 361-362
- Data types in MS Access, 324
- Data types in Oracle, 335
- Database connectivity, 400-404
- Database designing in VB, 399-412
 - Structure of the table, 399-400

- Modules, 400
- Database connectivity, 400-404
- Code for module, 405
- Code for MDI form, 405
- Code for student entry form, 405-410
- Code for student report form, 410-412
- Database management system (DBMS), 7, 315
- Databases, 315-318
- Data-centric architecture, 123-124
- Data-flow architecture, 123-124
- Data-flow arrows (in structure charts), 128
- Data-flow diagrams (DFD), 263
- Data-flow symbol, 62
- Datasheet view in MS Access, 322
- Data-store symbol, 62-63
- DBMSs, 5
- Debug tool bar, 270
- Debugging, 39, 127, 134, 186-189, 262-263
- Debugging guidelines, 189
- Debugging process, 187
- Decision symbol, 132
- Decision tables, 67-70
 - Basic elements of a decision table, 67
 - Limited-entry decision table, 68
 - Ambiguous decision table, 68
 - Incomplete and over-specified decision table, 69
 - Advantages of decision tables, 69
 - Disadvantages of decision tables, 70
- Decoupling data structures, 247
- Defect repair ratio, 204
- Definition of software, 4
- Definition of software design, 117-118
- Definition of software metrics, 149-150
- Definition of software quality, 89
- Deleting a primary key in Oracle, 346
- Deleting a table in Oracle, 338
- Deleting records in Oracle, 341
- Deliverable increment, 47
- Deliverables and milestones, 25-26
- Delivery and maintenance, 40
- Depth, 125
- Depth-first integration, 169
- Derived attribute, 81-82
- Design constraints, 76-77
- Design objectives/properties, 118-119
- Design principles, 119-123
- Design recovery, 238
- Design view in MS Access, 322
- Design specifications, 144-145
- Detaching a database in SQL, 378-380
- DHTML application, 267
- Diagnosibility, 105
- Diagramming tools, 234
- Dir list box, 291-292
- Directionality, 239
- Disclosures, 219
- Do while... statement in VB, 301
- Document restructuring, 242-243
- Domain understanding, 56
- Draw programs, 8
- Drawbacks of function points, 155-156
- Drive list box, 291
- Drivers, 167, 170-171
- Dropping a table, 364
- Dynamic structures, 252-253

E

- Economic feasibility, 31
- Edit tool bar, 270

- Editing, 232
- Editing SQL statements, 337
- Efficiency, 91-94, 105, 118-119, 262
- Embedded project, 212-213
- Embedded software, 16
- Enduring requirements, 59
- Engineering, 23-25
- Engineering and scientific software, 17-18
- Entity relationships, 80
- Entity-relationship diagram, 79-81
 - Entity, 79
 - Relationship, 80
 - Binary relationship, 80
 - Attributes, 81
- Entity types, 79
- Environment architecture, 226
- Error handling in Visual Basic, 465-480
 - Key handling, 465-469
 - Key locking at key press event, 469-470
 - Validation for integers, 469-470
 - Validation for characters, 470
 - Validation for float numbers, 470
 - Other error-handling methods, 470-471
 - On Error GoTo, 470-471
 - On Error Resume Next, 471
 - Some common errors, 471-479
 - Precautions, 479-480
- Error reports, 34
- Errors, 189-190
- Estimating, 207-215
- Estimating cost, 209-210
- Estimating effort, 209
- Estimating schedule, 209
- Estimating size, 208
- Event, 269

- Evolution of art to an engineer discipline, 20
- Evolution of software, 20-22
 - First era, 20
 - Second era, 21
 - Third era, 21
 - Fourth era, 21-22
- Evolutionary development model, 46-47
 - Need of an evolutionary model, 46-47
 - Uses of an evolutionary model, 47
- Evolvability, 95
- Example DFD, 63-64
- Example flowchart, 134-135
- Example of black-box testing, 177-178
- External coupling, 138
- External entity, 62
- External inquiry types, 155
- External interface file types, 155
- External interface requirements, 77
- External versus internal qualities, 90
- Extract abstractions, 239

F

- Factors affecting coupling, 137
- Factors affecting effort, 204
- Factors affecting re-engineering costs, 244
- Failure classifications, 109
- Failure intensity, 13
- Failures, 190
- Fan-in, 125
- Fan-out, 125
- Fault tolerance, 104
- Faults, 190
- Fault-tolerance requirements, 77
- Fault-tree analysis, 187

- Feasibility, 79
- Feasibility study, 31, 37-38
- Field properties in MS Access, 324-327
 - Field size, 324-325
 - Format, 325-326
 - Input mask, 326-327
 - Caption, 327
 - Default value, 327
 - Validation rules and validation text, 327
 - Required, 327
 - Allow zero length, 327
 - Index, 327
- Field size, 324-325
- File list box, 292-293
- First era, 20
- Flexibility, 92-94
- Flow lines, 132
- Flowchart drawing rules, 133
- Flowchart symbols, 132-133
- Flowcharts, 131-136, 157, 263
- For loop in VB, 302
- Foreign key, 318
- Form and report generator tools in VB, 234
- Form editor in VB, 270
- Form events in VB, 295
- Form properties in VB, 274-276
- Form window in VB, 270
- Forms in VB, 268
- Formal review, 58
- Formal technical review, 182
- Format, 325-326
- Forms and generator tools, 234
- Forward engineering, 244
- Forward traceability, 78
- Fourth-generation techniques (4GT), 255-257
 - Use of fourth-generation technologies, 256
 - Advantages of 4GT, 256
 - Disadvantages of 4GT, 256
 - Differences between 3GLs and 4GLs, 256-257
- Frames in VB, 281-282
- Function points, 154
- Function-point based measures, 154-157
 - Function points, 154
 - Function-point metric, 154-155
 - Special features, 155
 - Advantages of function points, 155
 - Drawbacks of function points, 155-156
 - Function-point metrics, 156-157
- Function symbol, 62
- Functional abstraction, 121
- Functional cohesion, 140
- Functional modules, 126
- Functional requirements, 54, 75-76
- Functional/black-box testing, 175-178
- Functionality, 90, 104, 250
- Functional-oriented versus the object-oriented approach, 143-144
- Function-point based measures, 154-157
- Function-point contribution of an element, 154
- Function-point metrics, 154-157
- Functions, 140-141
- Functions in VB, 303-313

G

- General guidelines and rules for constructing DFDs, 66-67
- Generating an SQL script, 370-375
- Generic risks, 219

Generic software, 8-9
 Global changes, 43
 Global variables, 250
 Gold-plating, 48
 Gotos, 249
 Graphical user interface (GUI), 42, 265
 Graphs in Visual Basic, 421-424
 Grouping in data reports in Visual Basic, 440-446
 Grouping of data, 343
 Grouping of fields, 462

H

Halstead's software science, 151-154
 Hardware interface requirements, 77
 Hardware and software requirements for Visual Basic, 266
 Editions, 266
 Hardware modules, 126
 Hexagon symbol, 133
 High-level design, 39
 Highly coupled, 136
 Horizontal CASE tools, 233
 Horizontal partitioning, 119-120
 Horizontal scroll bar, 289
 How to create a database using Enterprise Manager, 354-358
 How to create a setup, 481-490
 How to make a DSN, 451-456
 How to use Query Analyzer, 368-369
 How to use the script, 374-375
 How to write a good program, 262-263
 Readability, 262
 Design, 262
 Efficiency, 262
 Debugging, 262-263
 Testing, 263
 Human errors, 109

I

I-CASE environment, 231
 IEEE, 4, 9, 73-74, 200
 IEEE standards for SRS documents, 73-74
 IEEE recommended approaches for SRS, 73
 Benefits of SRS, 74
 IEEE recommended practice for software requirements specification, 74
 IEEE-SA, 73
 If-else statement in VB, 301
 IIS application, 268
 Image editors, 8
 Image processors, 8
 Implementation, 34-35
 Import table in MS Access, 322
 Importance of software, 4
 Importance of software quality, 96
 Importing a table in MS Access, 328
 Incomplete and over-specified decision table, 69
 Inconsistency, 75
 Incorrect fact, 75
 Increasing criticality of software, 96
 Increment approach to testing, 168-169
 Incremental module, 126
 Index, 327
 Index number of data report section in Visual Basic, 439
 Informal review, 58
 Informal technical review, 182

- Information flow model (IFM), 61
 - Information hiding, 247-249
 - Information modeling, 61
 - Input mask, 326-327
 - Input/output symbol, 132
 - Installability, 105
 - Installing SQL Server on a database server, 352
 - Intangibility of software, 96
 - Integrated data environment, 265
 - Integrated I-CASE tools, 231
 - Integrated development environment (IDE), 269-272
 - Tool bar, 270-271
 - Data types, 271-272
 - Integration and system testing, 40
 - Integration framework, 226
 - Integration testing, 167-172
 - Integrity, 92-94
 - Interactivity, 239
 - Iterative-enhancement model, 47-49
 - Advantages of iterative-enhancement model, 48
 - Disadvantages of iterative-enhancement model, 48-49
 - Interface design language (IDL), 11
 - Interface requirement, 77
 - Intermediate COCOMO model, 214
 - Internal documentation, 250-251
 - International standard organization (ISO), 99-106
 - ISO-9000 mission, 100-101
 - ISO certification, 101-102
 - Benefits of ISO-9000 certification, 102
 - Limitations of ISO-9000, 103
 - Uses of ISO, 103
 - Salient features of ISO-9001 requirements, 103
 - ISO-9126, 103-106
 - Internet information server (IIS), 265
 - Interoperability, 92-95, 104
 - Inventory analysis, 242
 - ISO certification, 101-102, 106
 - ISO standards, 106
 - ISO-9000, 100-107
 - ISO-9001, 100-106
 - ISO-9002, 100-101
 - ISO-9003, 101
 - ISO-9126, 103-106
 - Iterations, 253-255
 - Iterative paradigm, 47
 - Iterative process, 46
-
- ## J
-
- Jelinski-Moranda model, 112-114
-
- ## K
-
- Key features of structured programming, 255
 - Key handling, 465-469
 - Key locking at key press event, 469-470
 - Key process areas (KPA's), 98-99
 - Keys, 317-318
 - Candidate key, 317
 - Primary key, 317
 - Alternate key, 317
 - Composite key, 318
 - Foreign key, 318
 - Keywords, 130

L

- Labels in VB, 278-279
- Language-processing integration, 232
- Layered architecture, 123-124
- Learnability, 105
- Learning curve, 235
- Lehman's first law, 193
- Lehman's second law, 193
- Lehman's third law, 193
- Levels of a DFD, 64-66
- Levels of CASE, 224
 - Production process support technology, 224
 - Process management and technology, 224
 - Meta-CASE technology, 224
- Levels of reverse engineering, 237-238
- Levels of testing, 165-173
- Library modules, 128
- Limited-entry decision table, 68
- Linear sequential model, 47
- Link table in MS Access, 322
- List box in VB, 288-289
- Little Wood and Verall's model, 114
- Local changes, 43
- Logic error, 190
- Logical and control defects, 251
- Logical cohesion, 141-142
- Logical internal file types, 155
- Logical operators in VB, 298-300
- Loosely coupled, 136
- Lower CASE or Back-end tools, 234
- Low-level design, 39, 125-136
 - Modularization, 125-127
 - Classification of modules, 126
 - Advantages of modular systems, 126-127

- Structure charts, 127-129
 - Basic building blocks of a structure chart, 128-129
- Pseudo-code, 130-131
 - Advantages of pseudo-code, 130
 - Disadvantages of pseudo-code, 130-131
- Flowcharts, 131-136
 - Flowchart symbols, 132-133
 - Flowchart drawing rules, 133
 - Advantages of flowcharts, 134
 - Limitations of flowcharts, 134
 - Example of a flowchart, 134-135
 - Difference between flowcharts and structure charts, 136

M

- Main-control module, 169
- Maintainability, 91-94, 104
- Maintenance, 35-36
- Maintenance costs, 204-207
 - Factors affecting effort, 204
 - Modeling maintenance effort, 205-207
 - Belady and Lehman model, 205-206
 - Boehm model, 206-207
- Maintenance to-do list, 202-203
- Management of risks, 216-219
- Mandatory participant, 82
- Many to many cardinality relationship, 80
- Mapping, 236-237
- Maturity, 104
- Maximum cardinality relationship, 82
- McCall's quality factors, 91-92
- MDI form, 392
- Mean time between failures (MTBF), 110

Mean time to failure (MTTF), 110
 Mean time to repair (MTTR), 110
 Measurements of reliability and availability, 111-112
 Measures, metrics, and indicators, 26
 Meta-CASE technology, 224
 Microsoft Chart Control 6.0, 421
 Minimum cardinality relationship, 82
 Modeling maintenance effort, 205-207
 Modifiable, 78
 Modified E-R diagram, 82
 Modifying a record in Oracle, 341
 Modifying a table in MS Access, 328
 Modifying the data type and size of the column in Oracle, 338
 Modifying the structure of a table in Oracle, 338
 Modifying the structure of an existing table in SQL, 363
 Modular systems, 125-126
 Modularity, 127
 Modularization, 125-127
 Module interface, 250
 Module size, 249
 Modules, 121-146, 249-250, 400
 Monitoring and control for coding, 251-252
 Monitoring and control for design, 146
 Multidimensional spreadsheets, 5
 Multivalued attributes, 81

N

Naming, 248-249
 Need of an evolutionary model, 46-47
 Need for maintenance, 202-203
 Maintenance to-do list, 202-203

Negative functional testing, 177
 Nesting, 249
 Nesting construct, 254
 Nonfunctional requirements, 54

O

Object window, 273
 Objectives for reviews, 182-183
 Objectives of architectural design, 124-125
 Objectives of CASE, 228-229
 Improve productivity, 228
 Improve information system quality, 228-229
 Improve effectiveness, 229
 Objectives of structured programming, 253
 Object-management system (OMS), 225
 Object-oriented architecture, 123-124
 Object-oriented design approach, 39
 Omissions, 75
 On Error GoTo, 470-471
 On Error Resume Next, 471
 One to many cardinality relationship, 81
 Operability, 105
 Operating systems, 5
 Operational feasibility, 31
 Operator and operand count for a FORTRAN routine, 153
 Option button in VB, 285-286
 Organic project, 212-213
 Organization of SRS document, 70-72
 Organizational feasibility, 31
 Output symbol, 63

P

-
- Paint programs, 8
 - Parallel module, 126
 - Parameters, 250
 - Perfective maintenance, 35-36, 203
 - Performance, 94
 - Performance requirements, 76
 - Personal computer software, 17
 - Person-month curve, 212
 - Personnel attributes, 214
 - Phases of project management, 146
 - Phase-sensitive effort multipliers, 215
 - Picture box in VB, 277-278
 - Planning, 208, 232
 - Pointer in VB, 277
 - Portability, 91-94, 104
 - Portability services, 226
 - Positive functional testing, 176-177
 - Precautions, 479-480
 - Precontrolled objects, 194
 - Presentation graphics, 7
 - Preventive maintenance, 36, 203
 - Primary key (in a database), 317
 - Primary keys in Oracle, 345-346
 - Adding a primary key, 345
 - Adding a primary key in an existing table, 345-346
 - Deleting a primary key, 346
 - Principles of re-engineering, 240
 - Principles of structured programming, 253-254
 - Prioritization, 57
 - Private access specifier, 248
 - Probability of failure on demand (POFOD), 110-111
 - Problem partitioning, 119-120
 - Procedural cohesion, 141
 - Process descriptions, 88
 - Process management and technology, 224
 - Process metrics, 150
 - Process model of elicitation and analysis, 56-57
 - Process modeling, 50
 - Process of requirements engineering, 55-60
 - Requirement elicitation and analysis, 55-56
 - Process model of elicitation and analysis, 56-57
 - Domain understanding, 56
 - Requirements collection, 56
 - Classification, 56
 - Conflict resolution, 57
 - Prioritization, 57
 - Requirements checking, 57
 - Requirements documentation, 57-59
 - Requirements definition document, 57-58
 - Requirements review, 58-59
 - Requirements management, 59
 - Classes of requirements management, 59
 - Requirements management planning, 60
 - Process support modules, 126
 - Processing symbol, 132
 - Product and process, 26
 - Product and process qualities, 90
 - Product attributes, 214
 - Product metrics, 150
 - Product operation and quality factors, 92
 - Product plans, 88
 - Production introduction, 88
 - Production process support technology, 224
 - Productivity, 95
 - Program analysis tools, 232

- Program debugging, 188-189
 - Program design methods, 253
 - Program documentation, 250-251
 - Program layout, 249
 - Program modularization, 241
 - Program phase, 262
 - Program slicing, 187
 - Program structure improvement, 241
 - Program-design language, 130
 - Programming, 232
 - Programming languages, 250
 - Programming in Visual Basic with MS Access 2000, 391-392
 - MDI form, 392
 - Programming with Oracle and SQL Server 2000, 413-420
 - Table creation, 413
 - Data links, 414-416
 - Creating a connection, 417-418
 - Working with the project, 418-420
 - Required modifications, 418-419
 - Use of condade function, 419-420
 - Data export at runtime, 420
 - Working in a project with an SQL Server 2000 database, 420
 - Programming tools, 263
 - Algorithms, 263
 - Flowcharts, 263
 - Pseudo-code, 263
 - Data-flow diagrams (DFD), 263
 - Programs, 26
 - Programs versus software products, 26-27
 - Programs, 26
 - Software products, 27
 - Programming style, 248-250
 - Naming, 248-249
 - Control constructs, 249
 - Information hiding, 249
 - Gotos, 249
 - User-defined type, 249
 - Nesting, 249
 - Module size, 249
 - Program layout, 249
 - Module interface, 250
 - Robustness, 250
 - Side effects, 250
 - Project analysis, 32
 - Project attributes, 214
 - Project estimating, 207-215
 - Project explorer, 271
 - Project metrics, 150
 - Project-estimation guidelines, 211
 - Project-estimation process, 210
 - Project-specific risks, 219
 - Property window, 270
 - Prototyping model, 41-44
 - Reasons for using prototyping model, 42-43
 - Controlling changes during prototyping, 43
 - Advantages of prototyping models, 43
 - Limitations of prototyping models, 43-44
 - Prototyping, 41-44, 47, 232
 - Prototyping support, 227
 - Pseudo-code, 130-131, 263
- ## Q
-
- Quality goals, 88
 - Query Analyzer, 368-369

R

-
- RAD model, 49-50
 - Business modeling, 49-50
 - Data modeling, 50
 - Process modeling, 50
 - Application generation, 50
 - Testing and turnover, 50
 - Disadvantages of RAD model, 50
 - Rate of occurrences of failure (ROCOF), 111
 - Readability, 262
 - Real-time software, 16
 - Reasons for poor/inaccurate estimation, 210-211
 - Reasons for using prototyping model, 42-43
 - Reasons white-box testing is performed, 174-175
 - Recognition of need, 31
 - Recoverability, 104
 - Re-documentation, 238
 - Re-engineering process, 240-241
 - Re-engineering tools, 232
 - Regression testing, 171
 - Relational database management system (RDBMS), 315, 329
 - Relationship between coupling and cohesion, 142-143
 - Relationship cardinalities, 82
 - Reliability, 91-94, 104
 - Reliability and fault tolerance, 77
 - Reliability assessment, 115
 - Reliability growth modeling, 112-114
 - Jelinski-Moranda model, 112-114
 - Little Wood and Verall's model, 114
 - Step function model, 114
 - Reliability issues, 107-109
 - Software reliability, 107-108
 - Software-reliability specifications, 108-109
 - Reliability terminologies, 109
 - Classification of failures, 109
 - Reliability measurement process, 115
 - Reliability metrics, 110-111
 - Mean time to failure (MTTF), 110
 - Mean time to repair (MTTR), 110
 - Mean time between failures (MTBF), 110
 - Probability of failure on demand (POFOD), 110-111
 - Rate of occurrences of failure (ROCOF), 111
 - AVAIL (Availability), 111
 - Reliability terminologies, 109
 - Renaming a table, 338
 - Repairability, 95
 - Repetition, 129
 - Replaceability, 105
 - Repository, 226
 - Representative qualities, 93-96
 - Required modifications, 418-419
 - Requirement analysis and specification, 38
 - Requirement definition and description (RDD), 54-56
 - Requirement elicitation and analysis, 55-56
 - Requirements checking, 57
 - Requirements collection, 56
 - Requirements cross-reference, 144
 - Requirements definition document, 57-58
 - Requirements documentation, 57-59
 - Requirements engineering (RE), 53-56
 - Types of requirements, 54
 - Functional requirements, 54
 - Nonfunctional requirements, 54
 - Requirements identification, 60
 - Requirements management, 59-60
 - Requirements review, 58-59

Retrieval of selected data in the data report in Visual Basic, 436-439

Return value of command buttons in VB, 310

Reusability, 92-94

Reverse software engineering, 234-239

- Definition, 235
- Purpose of, 236
- Reverse-engineering process, 236
- Reverse-engineering tasks, 236-237
- Levels of reverse engineering, 237-238
 - Re-documentation, 238
 - Design recovery, 238
- Characteristics of reverse engineering, 239
 - Abstraction level, 239
 - Completeness, 239
 - Interactivity, 239
 - Directionality, 239
 - Extract abstractions, 239
- Application areas of reverse engineering, 239

Review meeting, 183

Revision control, 199

Risk analysis, 218

Risk and risk management, 88

Risk avoidance, 218

Risk category, 218

Risk detection, 218

Risk elimination, 219

Risk hierarchy, 216

Risk management, 215-219

Risk pending, 219

Risk prioritization, 218

Risk recovery, 219

Risk resolution, 219

Risk transfer, 219

Risk-analysis table, 218

Risk-management tool, 217

RMS calculating software, 129

Robustness, 94, 250

S

Sandwich integration testing, 171-172

Saving projects and forms in VB, 393-399

- Design of student entry form, 393-396
- Design of student report form, 396-399

Saving a table in MS Access, 327

Schedule estimation, 209

Scopes of variables in VB, 298

Security, 77, 104

SEI-CMM model, 106-107

Selection symbol, 129

Semi-automatic grounded environment (SAGE), 36

Semi-detached project, 212-213

Sequence constructs, 253-254

Sequencing, 255

Sequential cohesion, 140

Sequential module, 126

Shape control in VB, 293-294

Side effects, 250

Simple attribute, 81

Simplicity, 119

Single entry-single exit constructs, 34

Single valued attribute, 81

Single-entry, 255

Single-exit, 255

Smoke testing, 171

Software applications, 15-18

- System software, 15-16
- Real-time software, 16

- Embedded software, 16
- Business software, 16-17
- Personal computer software, 17
- Artificial-intelligence software, 17
- Web-based software, 17
- Engineering and scientific software, 17-18
- Software as an evolutionary model, 193
- Software breakage, 48
- Software characteristics, 12-13
- Software crisis, 13-15
- Software components, 11
- Software-configuration management, 200-202
 - Versions and releases, 201
 - Version and release management, 201-202
- Software-configuration management activities, 193-197
 - Configuration identification, 194
 - Configuration control, 194-195
 - Configuration accounting, 195
 - Status accounting, 195-196
 - Configuration audit, 196-197
- Software curve, 13
- Software development, 19
- Software-development life-cycle, 29-36
 - Recognition of need, 31
 - Feasibility study, 31
 - Project analysis, 32
 - System design, 32-33
 - Coding, 33-34
 - Testing, 34
 - Implementation, 34-35
 - Maintenance, 35-36
 - Corrective maintenance, 35
 - Adaptive maintenance, 35
 - Perfective maintenance, 35-36
 - Preventive maintenance, 36
- Software engineering institute, 215
- Software-engineering principles, 10-11
- Software-engineering processes, 18-20
- Software evolution, 19-20
- Software metrics, 149-151
 - Definition of software metrics, 149-150
 - Categories of metrics, 149
 - Attributes of effective software metrics, 150-151
- Software myths, 15
- Software products, 27
- Software-project estimation, 207-211
 - Estimating size, 208
 - Estimating effort, 209
 - Estimating schedule, 209
 - Estimating cost, 209-210
 - Reasons for poor/inaccurate estimation, 210-211
 - Project-estimation guidelines, 211
- Software project management, 146
- Software quality, 89-96
 - Definition of software quality, 89
 - Classification of software qualities, 89-90
 - External versus internal qualities, 90
 - Product and process qualities, 90
 - Software quality attributes, 90-91
 - Functionality, 90
 - Reliability, 91
 - Usability, 91
 - Efficiency, 91
 - Maintainability, 91
 - Portability, 91
 - McCall's quality factors, 91-92
 - Product operation and quality factors, 92

- Software quality criteria, 92-93
- Representative qualities, 93-96
 - Correctness, 93
 - Reliability, 93-94
 - Robustness, 94
 - Performance, 94
 - Verifiability, 94
 - Repairability, 95
 - Evolvability, 95
 - Understandability, 95
 - Interoperability, 95
 - Productivity, 95
 - Timeliness, 95-96
 - Visibility, 96
- Importance of software quality, 96
- Software quality assurance (SQA), 87-89
 - SQA objectives, 87
 - SQA goals, 87-88
 - SQA plan, 88-89
- Software quality attributes, 90-91
- Software quality criteria, 92-93
- Software quality plan, 88-89
 - Production introduction, 88
 - Product plans, 88
 - Process descriptions, 88
 - Quality goals, 88
 - Risks and risk management, 88
- Software re-engineering, 240-245
 - Principles of re-engineering, 240
 - Re-engineering process, 240-241
 - Source-code translation, 241
 - Reverse engineering, 241
 - Program structure improvement, 241
 - Program modularization, 241
 - Data re-engineering, 241
- Software re-engineering process model, 241-244
 - Inventory analysis, 242
 - Document restructuring, 242-243
 - Reverse engineering, 243
 - Code restructuring, 243
 - Data restructuring, 243-244
 - Forward engineering, 244
- Factors affecting re-engineering costs, 244
- Differences between forward engineering and re-engineering, 244
- Advantages and disadvantages, 245
- Software reliability, 107-108
- Software reliability curve, 107
- Software-reliability specifications, 108-109
- Software requirements specification (SRS), 57, 156, 163
- Software-risk analysis and management, 215-220
 - Risk management, 215-216
 - Management of risks, 216-219
 - Risk management categories, 216-219
 - Risk avoidance, 218
 - Risk detection, 218
 - Risk analysis, 218
 - Risk category, 218
 - Risk prioritization, 218
 - Risk recovery, 219
- Sources of risks, 219
 - Generic risks, 219
 - Project-specific risks, 219
- Software specifications, 18
- Software testing, 161-179
 - Testing principles, 162-163
 - Testing objectives, 163-164
 - Testing oracles, 164
 - Levels of testing, 165-173

- Unit testing, 165-167
- Integration testing, 167-172
- System testing, 172-173
- White-box testing/structural testing, 173-175
 - Reasons white-box testing is performed, 174-175
 - Advantages of structural/white-box testing, 175
- Functional/black-box testing, 175-178
 - Categories of black-box testing, 176-177
 - Example of black-box testing, 177-178
 - Advantages of black-box testing, 178
- Test plan, 178-179
- Test-case design, 179
- Software-testing strategies, 181-190
 - Static-testing strategies, 181
 - Formal technical reviews, 182
 - Objectives for reviews, 182-183
 - Types of reviews, 182
 - Review meeting, 182
 - Results of formal technical review, 183
 - Code walk-throughs, 183-184
 - Code inspections, 184-186
 - Checklist for code inspections, 185
 - Differences between walk-throughs and inspections/reviews, 185-186
 - Debugging, 186-189
 - Debugging tactics/categories, 186-187
 - Debugging process, 187
 - Program debugging, 188-189
 - Debugging guidelines, 189
 - Characteristics of bugs, 189
- Errors, 189-190
 - Types of errors, 190
 - Faults, 190
 - Failures, 190
- Software validation, 19
- Software-version control, 199-200
- Source-code control, 199
- Source-code translation, 241
- Sources of risks, 219
- Special features, 155
- Spiral model, 44-46
 - Characteristics of spiral model, 45
 - Limitations of spiral model, 46
 - Advantages of spiral model, 46
 - Disadvantages of spiral model, 46
- Spreadsheets, 6
- SQA goals, 87-88
- SQA objectives, 87
- SQA plan, 88-89
- SQL Query Analyzer, 349, 368
- SQL Server 2000, 349-389
 - SQL Query Analyzer, 349
 - Starting SQL Server 2000, 349-350
 - Installing, 351-352
 - Choosing components and options, 352
 - Installing SQL Server on a database server, 352
 - Using SQL Server with Client/Server applications, 352-353
 - Other SQL Server components, 353
 - Creating a database, 353-354
 - How to create a database using Enterprise Manager, 354-358
 - Creating a database using the create database wizard in Enterprise Manager, 358
 - Creating a new table, 358-361
 - Data types, 361-362
 - Viewing the structure of the table, 362

- Modifying the structure of an existing table, 363
- Dropping a table, 364
- Opening a table, 365-367
- Query Analyzer, 368-369
 - How to use Query Analyzer, 368-369
- Generating an SQL script, 370-375
 - How to use the script, 374-375
- Attaching a database, 376-378
- Detaching a database, 378-380
- Copy database wizard, 380
- Importing and exporting a database, 380
 - Data transformation services, 380-385
- SQL Service Manager, 386-389
 - Authentication, 386-387
 - Creating a connection with Visual Basic, 387-389
- SQL Server authentication, 387
- SQL Server Desktop Engine, 353
- SQL Server editions, 350
- SQL Service Manager, 386-389
- SQL templates, 368
- SRS document, 38, 70-72
 - Organization of SRS document, 70-72
 - Uses for SRS documents, 72
- SRS validation, 75
 - Omission, 75
 - Inconsistency, 75
 - Incorrect fact, 75
 - Ambiguity, 75
- Stability, 105
- Stamp coupling, 138
- Standard EXE, 266
- Standard tool bar, 270
- Standards compliance, 76
- Starting SQL Server 2000, 349-350
- Starting with Crystal Report 8.0, 448-451
- Starting with Oracle, 329-331
- Static analysis, 251
- Static structures, 252-253
- Static-testing strategies, 181
- Status accounting, 195-196
- Step function model, 114
- Stepwise refinement, 253-254
- Structural testing, 173-175
- Structure charts, 127-129
- Structure of a database, 317
- Structure of a table, 399-400
- Structured analysis and design, 227
- Structured programs, 252-255
- Structured programming, 34, 252-255
 - Objectives of structured programming, 253
 - Principles of structured programming, 253-254
 - Key features of structured programming, 255
 - Advantages of structured programming, 255
- Stubs, 167
- Suitability, 104
- Symbolic execution, 251
- Symbols uses for constructing DFDs, 62
- Syntax error, 190
- Syntax and query in Oracle, 336-343
 - Table creation, 336
 - Table with data, 336
 - To view all the tables of the currently connected user, 337
 - To view the structure of a table, 337
 - Creation of a table from another table, 337

- Editing SQL statements, 337
- Renaming a table, 338
- Deleting a table, 338
- Modifying the structure of a table, 338
- Modifying the data type and size of the column, 338
- To add a new column in a table, 339
- Data insertion in a table, 339
- Data selection, 339
- Conditional searching, 339
- Insertion of data from another table, 341
- Modifying a record, 341
- Deleting records, 341
- Use of NOT, 342
- Use of LIKE, 342
- Use of BETWEEN, 342
- Use of IN, 343
- Use of NOT IN, 343
- Grouping of data, 343
- Synthesis, 134
- System design, 32-33, 117-146
- System error, 109
- System failure, 109
- System fault, 109
- System software, 5, 15-16
- System testing, 172-173
- System/software design, 117-123
 - Definition of software design, 117-118
 - Design objectives/properties, 118-119
 - Design principles, 119-122
 - Problem partitioning, 119-120
 - Abstraction, 121
 - Top-down and bottom-up design, 121-122

T

- Table creation in Oracle, 336, 413
- Table wizard in MS Access, 322
- Tables (in databases), 316
- Technical feasibility, 31
- Templates, 161, 232
- Temporal cohesion, 141
- Terminal symbol, 132
- Terminologies, 25-26
 - Deliverables and milestones, 25-26
 - Product and process, 26
 - Measures, metrics, and indicators, 26
- Test CASE generator, 228
- Test plan, 178-179
- Test reports, 34
- Testability, 92-94, 105
- Test-case design, 179
- Testing and turnover, 50
- Testing objectives, 163-164
- Testing oracles, 164
- Testing principles, 162-163
- Text box controls in VB, 279-281
- Three-level product hierarchy, 215
- Timeliness, 95-96
- Timer, 290
- Tool bar in VB, 270-271
- Tool-box controls in VB, 271, 276-296
 - Tool box, 276
 - Pointer, 277
 - Picture box, 277-278
 - Label, 278-279
 - Text box, 279-281
 - Frame, 281-282
 - Command button, 282-283

- Check box, 283-285
- Option button, 285-286
- Combo box, 287-288
- List box, 288-289
- Horizontal scroll bar, 289
- Vertical scroll bar, 290
- Timer, 290
- Drive list box, 291
- Dir list box, 291-292
- File list box, 292-293
- Shape, 293-294
- Line, 294-295
- Image, 295
- Form events, 295
- Control events, 296
- Tools-management services (tools-set), 225
- Top-down design, 121-122
- Top-down integration, 169-170
- Total quality management (TQM), 107
- Traceability, 58, 79, 118
- Traceability policies, 60
- Traceable, 78
- Tracing, 34, 262
- Traditional design approach, 38-39
- Types of attributes, 81-82
 - Simple attribute, 81
 - Composite attribute, 81
 - Single valued attribute, 81
 - Multivalued attribute, 81
 - Derived attribute, 81-82
- Types of cohesion, 140-142
- Types of couplings, 137-139
- Types of errors, 190
- Types of module cohesion, 140
- Types of reviews, 182

- Types of software, 5-8
 - System software, 5
 - Application software, 5
 - Operating systems, 5
 - Utilities, 5
 - Compilers and interpreters, 5
 - Word processors, 6
 - Spreadsheets, 6
 - Presentation graphics, 7
 - Database management system (DBMS), 7
 - Image processors, 8
 - Paint programs, 8
 - Draw programs, 8
 - Image editors, 8

U

- Uncontrolled objects, 194
- Uncoupled, 137
- Understandability, 95, 105
- Unit test consideration, 165-166
- Unit test environment, 167
- Unit test procedure, 166-167
- Unit testing, 165-167
- Upper CASE tools/Front-end CASE tools, 233-234
- Usability, 91-94, 104-105
- Use of App.Path in VB, 412
- Use of BETWEEN, 342
- Use of condate function, 419-420
- Use of fourth-generation technologies, 256
- Use of IN, 343
- Use of LIKE, 342
- Use of NOT IN, 343
- Use of NOT, 342

- User creation by navigation in Oracle 8, 332-335
- User interface, 77, 224
- User manual, 77
- User-defined type, 249
- Uses for SRS documents, 72
- Uses of an evolutionary model, 47
- Uses of ISO, 103
- Using SQL Server with Client/Server applications, 352-353
- Utilities, 5

V

- Validation, 85-86
- Validation for characters, 470
- Validation for float numbers, 470
- Validation for integers, 469-470
- Validation rules and validation text, 327
- Variable declaration in VB, 297-298
- Variable naming conventions in VB, 297
- Verifiability, 58, 94, 118
- Verifiable, 78
- Verification and validation, 85-86
- Verification for design, 145
- Version and release management, 201-202
- Versions and releases, 201
- Vertical CASE tools, 233
- Vertical partitioning, 120
- Vertical scroll bar, 290
- Viewing the structure of the table, 362
- Visibility, 96
- Visual Basic application types, 266-268
 - Standard EXE, 266
 - ActiveX EXE, 266
 - ActiveX DLL, 266

- ActiveX control, 266
- ActiveX document EXE, 266
- ActiveX document DLL, 266
- VB application wizard, 267
- VB wizard manager, 267
- Data project, 267
- DHTML application, 267
- IIS application, 268
- Addin, 268
- VB enterprise edition control, 268
- Visual Basic application wizard, 267
- Visual Basic editions, 266
- Visual Basic enterprise edition control, 268
- Visual Basic functions, 303-313
- Visual Basic terminology, 268-269
 - Form, 268
 - Object, 268
 - Control, 268
 - Property, 269
 - Method, 269
 - Event, 269
- Visual Basic tool bars, 270-271
- Volatile requirements, 59

W

- Waterfall model, 36-41
 - Feasibility study, 37-38
 - Requirement analysis and specification, 38
 - Design and specification, 38-39
 - Traditional design approach, 38-39
 - Object-oriented design approach, 39
 - Coding and module testing, 39
 - Integration and system testing, 40
 - Delivery and maintenance, 40

- Advantages of waterfall model, 41
- Disadvantages of waterfall model, 41
- Web-based software, 17
- White-box testing/structural testing, 173-175
- Width, 125
- With-end with statement in VB, 302
- Word processors, 6
- Working in a project with an SQL Server 2000 database, 420