FEDERAL UNIVERSITY OF TECHNOLOGY MINNA,
MINNA, NIGER STATE, NIGERIA

CENTRE FOR OPEN DISTANCE AND
e-LEARNING (CODeL)

B.TECH. COMPUTER SCIENCE
PROGRAMME

COURSE TITLE

SOFTWARE DESIGN, TECHNIQUE AND MANAGEMENT

COURSE CODE

CPT 416

COURSE CODE

CPT 416

COURSE UNIT

2

Course Coordinator
Bashir MOHAMMED (Ph.D.)
Department of Computer Science
Federal University of Technology (FUT) Minna
Minna, Niger State, Nigeria.

Course Development Team

CPT 416: DATA AND ANALYSIS OF ALGORITHM

Subject Matter Experts	S.A BASHIR FUT Minna, Nigeria.
Course Coordinator	Bashir MOHAMMED (Ph.D.) Department of Computer Science FUT Minna, Nigeria.
Instructional Designers	Oluwole Caleb FALODE (Ph.D.) Bushrah Temitope OJOYE (Mrs.) Centre for Open Distance & e-Learning, FUT Minna, Nigeria
ODL Experts	Amosa Isiaka GAMBARI (Ph.D.) Nicholas E. ESEZOBOR
Language Editors	Chinenye Priscilla UZOCHUKWU (Mrs.) Mubarak Jamiu ALABEDE
Centre Director	Abiodun Musa AIBINU (Ph.D.) Centre for Open Distance & e-Learning FUT Minna, Nigeria.

CPT 416 Study Guide

Introduction

CPT 416 Software Design, Technique and Management is a 2 credit unit course for students studying towards acquiring a bachelor of science in computer science and other related disciplines. The course is divided into 7 module and 15 study units.

Software is a kind of system or we can say the package which is used in many organizations. It is a general term for the various kinds of programs used to operate computers and related devices. It can be thought of as the variable part of a computer and hardware the invariable part Software is often divided into two categories: System Software and application software. There are various types of application software such as scientific applications, mathematical applications, engineering applications, business applications.

Software Development is a process to maintain software. At first glance to a developer - this is the coding process. This is when you sit down the front of computer and start to write codes that later processed on compiled and run. Become the actual software that is used by the end user. It is the process of developing software through successive techniques in a continuous way. This process includes not only the actual writing of code but also the preparation of requirements and objectives, the design of what is to be coded, and confirmation that what is developed has met objectives.

Software Development Techniques are well defined and systematic approach, put into practice for the development of a reliable high quality information system. However, the complexity of modern systems and computer products long ago made the need clear for some kind of orderly development process.

In this course an introduction to the fields of software development are provided. It begins by considering a number of common development techniques upon which many systems are currently based and show how different techniques can be combined in a single software development. Software development techniques are used on a large scale by various organizations.

What you will Learn in this Course

The overall aim of this course, CPT 416 is to introduce you to basic concepts of process involved in software development.

Software development is the set of activities and processes for programmers that will eventually result in a software product. This may include requirement analysis, software design, implementation, testing, documentation, maintenance and then describing computer programs that meet user requirements within the constraints of the environment. It is a structure imposed on the development of software product. Software development is the most important process in developing a Software/tool. The successful execution of the project highly depends on the techniques used to develop the model.

Software development technology has an under the model-explicit or implicit-of the development process. In order to understand more about the development process and the methodologies, we abstract from these. The perspective chosen for the abstraction include models developed during the process and the kind of abstraction involved in the techniques of the process.

Course Aim

This course aim to introduce student to the basic concept of Software, Design, Management and Techniques. It also introduces the reader to some basic programming languages like C++, Java (Applet) Web Server and HTML. It will help the reader to understand the level of disparity and relationship between these languages

Course Objectives

It is important to note that each unit has specific objective. Student should study them carefully before proceeding to subsequently unit. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objective after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit. However, below is overall objective of this course. On completing this course the following would be learnt:

- Software characteristics, Software components, Software applications
- Software Engineering and Techniques, Software Process, Software Process Model
- About Management Spectrum, About People, About Problem, About Process
- About Software Scope, About Software Resources, Software Project Estimations
- Basic Concepts, About The Relationship between, People and Effort, defining a task set for the Software Project
- About Quality Concepts, About Quality Movement, About Software Reviews
- About SCM Process, About Version Control, About Change Control
- About Software Design, About Design Process, About Design Principles, About Design Concepts, Design Documentation
- About Architectural Design Optimization, About API Design, About Procedural Design
- About Programming, About C++ (Brief Introduction)
- About Client/Server Computing, About various types of servers
- About Middleware, About its prospects
- About a web, About tags in web
- About an applet, About how it works

Working through this Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contain some Self-Assessment Exercise(s)s and tutor assignments, and at some point in these courses, you required to submit the tutor marked assignments. There is also final examination at the end of these courses. Stated below are the component of these courses and what you have to do.

Course Materials

1. Course Guide
2. Study Units
3. Text Books

4. Assignment Files
5. Presentation Schedule

Study Unit

There are 15 study unit and 7 modules in this course, they are:

Module 1:

Unit 1: The Software Product

Unit 2: The Software Process

Module 2:

Unit 1: The Project Management Concept

Unit 2: Software Project Planning

Module 3:

Unit 1: Application Programming Interface, Project Scheduling
And Tracking

Unit 2: Software Quality Assurance

Module 4:

Unit 1: Software Configuration Management

Unit 2: Design Concepts and Principles

Module 5:

Unit 1: Design Methods

Unit 2: Programming

Module 6:

Unit 1: Client/Server Computing

Unit 2: Client/Server Strategies

Module 7:

Unit 1: Middleware

Unit 2: Web Technology

Unit 3: Introduction to Applets

Recommended Texts

Module 1, Unit 1

Brooks, F., The Mythical Man-Month, Addison-Wesley, 1975.
[DEJ98] De Jager, P. et al., Countdown Y2K: Business Survival Planning for the Year 2000, Wiley, 1998.
[DEM95] DeMarco, T., Why Does Software Cost So Much? Dorset House, 1995, p. 9.
[FEI83] Feigenbaum, E.A. and P. McCorduck, The Fifth Generation, Addison-Wesley, 1983.

Module 1, Unit 2
Brooks, F., The Mythical Man-Month, Addison-Wesley, 1975.
[DEJ98] De Jager, P. et al., Countdown Y2K: Business Survival Planning for the Year 2000, Wiley, 1998.
[DEM95] DeMarco, T., Why Does Software Cost So Much? Dorset House, 1995, p. 9.
[FEI83] Feigenbaum, E.A. and P. McCorduck, The Fifth Generation, Addison-Wesley, 1983, softwareengineering.com

Module 2, Unit 1
[AIR99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics," Draft Report, March 8, 1999.
[BAK72] Baker, F.T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal,* vol. 11, no. 1, 1972, pp. 56–73.
[BOE96] Boehm, B., "Anchoring the Software Process," *IEEE Software,* vol. 13, no. 4,July 1996, pp. 73–82.[CON93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization, *CACM,* vol. 36, no. 10, October 1993, pp. 34–43.
[COU80] Couga, http://www.isaca.org/Journal/Past-Issues/2006, www.google.com

Module 2, Unit 2
[BRO95] Brooks, M., The Mythical Man-Month, Anniversary Edition, AddisonWesley, 1995.
[FLE98] Fleming, Q.W. and J.M. Koppelman, "Earned Value Project Management," Crosstalk, vol. 11, no. 7, July 1998, p. 19.
[HUM95] Humphrey, W., A Discipline for Software Engineering, Addison-Wesley, 1995.
[PAG85] Page-Jones, M., Practical Project Management, Dorset House, 1985, pp. 90–91.

Module 3, Unit 1
[BEN92] Bennatan, E.M., Software Project Management: A Practitioner's Approach, McGraw-Hill, 1992.
[BOE81] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.
[BOE89] Boehm, B., Risk Management, IEEE Computer Society Press, 1989.
[BOE96] Boehm, B., "Anchoring the Software Process," IEEE Software, vol. 13, no. 4, July 1996, pp. 73–82.

Module 3, Unit 2
www. Wikipedia.com
[ALV64] Alvin, W.H. von (ed.), Reliability Engineering, Prentice-Hall, 1964.
[ANS87] ANSI/ASQC A3-1987, Quality Systems Terminology, 1987.
[ART92] Arthur, L.J., Improving Software Quality: An Insider's Guide to TQM, Wiley,1992.
[ART97] Arthur, L.J., "Quantum Improvements in Software System Quality, CACM,vol. 40, no. 6, June 1997, pp. 47–52.
[BOE81] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.
[CRO79] Crosby, P., Quality Is Free, McGraw-Hill, 1979.
[DEM86] Deming, W.E., Out of the Crisis, MIT Press, 1986.

[DEM99] DeMarco, T., “Management Can Make Quality (Im)possible,” Cutter IT Summit, Boston, April 1999.
[DIJ76] Dijkstra, E., A Discipline of Programming, Prentice-Hall, 1976.
[DUN82] Dunn, R. and R. Ullman, Quality Assurance for Computer Software, McGrawHill,1982.
[FRE90] Freedman, D.P. and G.M. Weinberg, Handbook of Walkthroughs, Inspections and Technical Reviews, 3rd ed., Dorset House, 1990.
[GIL93] Gilb, T. and D. Graham, Software Inspections, Addison-Wesley, 1993.
[GLA98] Glass, R., “Defining Quality Intuitively,” IEEE Software, May 1998, pp. 103–104, 107.
[HOY98] Hoyle, D., ISO 9000 Quality Systems Development Handbook: A Systems Engineering Approach, Butterworth-Heinemann, 1998.

Module 4, Unit 1

[BAB86] Babich, W.A., Software Configuration Management, Addison-Wesley, 1986.
[BAC98] Bach, J., “The Highs and Lows of Change Control,” Computer, vol. 31, no. 8, August 1998, pp. 113–115.
[BER80] Bersoff, E.H., V.D. Henderson, and S.G. Siegel, software Configuration Management,Prentice-Hall, 1980.
[CHO89] Choi, S.C. and W. Scacchi, "Assuring the correctness of a Configured Software Description," Proc. 2nd Intl. Workshop on Software Configuration Management, ACM, Princeton, NJ, October 1989, pp. 66–75.
[CLE89] Clemm, G.M., "Replacing Version Control with Job Control," Proc. 2nd Intl. Workshop on Software Configuration Management, ACM, Princeton, NJ, October 1989, pp. 162–169.
[GUS89] Gustavsson, A., "Maintaining the Evaluation of Software Objects in an Integrated Environment,"Proc.2nd Intl. Workshop on Software Configuration Management, ACM, Princeton, NJ, October 1989, pp. 114–117.
[HAR89] Harter, R., "Configuration Management," HP Professional, vol. 3, no. 6, June 1989.
[IEE94] Software Engineering Standards, 1994 edition, IEEE Computer Society, 1994.
[LIE89] Lie, A. et al., "Change Oriented Versioning in a Software Engineering Database,"Proc. 2nd Intl. Workshop on Software Configuration Management, ACM, Princeton,NJ,October,1989, pp. 56–65.[NAR87]
Narayanaswamy, K. and W. Scacchi, "Maintaining Configurations of Evolving Software Systems," IEEE Trans. Software Engineering, vol. SE-13, no. 3, March 1987, pp. 324–334.

[REI89] Reichen berger, C., "Orthogonal Version Management," Proc. 2nd Intl. Workshop on Software Configuration Management, ACM, Princeton, NJ, October 1989, pp.137–140.

Module 4, Unit 2

[AHO83] Aho, A.V., J. Hopcroft, and J. Ullmann, Data Structures and Algorithms, Addison-Wesley, 1983.

[BAS98] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice, Addi-son-Wesley, 1998.

[BEL81] Belady, L., Foreword to Software Design: Methods and Techniques (L.J. Peters, author), Yourdon Press, 1981.

[BRO98] Brown, W.J., et al., Anti-Patterns, Wiley, 1998.

[BUS96] Buschmann, F. et al., Pattern-Oriented Software Architecture, Wiley, 1996.

[DAH72] Dahl, O., E. Dijkstra, and C. Hoare, Structured Programming, Academic Press,1972.

[DAV95] Davis, A., 201 Principles of Software Development, McGraw-Hill, 1995.

[DEN73] Dennis, J., "Modularity," in Advanced Course on Software Engineering (F. L. Bauer, ed.), Springer-Verlag, New York, 1973, pp. 128–182.

[GAM95] Gamma, E. et al., Design Patterns, Addison-Wesley, 1995.

[GAN89] Gonnet, G., Handbook of Algorithms and Data Structures, 2nd ed., AddisonWesley,1989.

[GAR95] Garlan, D. and M. Shaw, "An Introduction to Software Architecture," Advances in Software Engineering and Knowledge Engineering, vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.

[JAC75] Jackson, M.A., Principles of Program Design, Academic Press, 1975.

[JAC83] Jackson, M.A., System Development, Prentice-Hall, 1983.

[JAC92] Jacobson, I., Object-Oriented Software Engineering, Addison-Wesley, 1992.

[KAI83] Kaiser, S.H., The Design of Operating Systems for Small Computer Systems, Wiley-Interscience, 1983, pp. 594 ff.

[KRU84] Kruse, R.L., Data Structures and Program Design, Prentice-Hall, 1984.

[MCG91] McGlaughlin, R., “Some Notes on Program Design,” Software Engineering Notes, vol. 16, no. 4, October 1991, pp. 53–54.

[MEY88] Meyer, B., Object-Oriented Software Construction, Prentice-Hall, 1988.

[MIL72] Mills, H.D., "Mathematical Foundations for Structured Programming," Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, Maryland, 1972.

[MYE78] Myers, G., Composite Structured Design, Van Nostrand,1978.

[ORR77] Orr, K.T., Structured Systems Development, Yourdon Press, 1977.

[PAR72] Parnas, D.L., "On Criteria to Be Used in Decomposing Systems into Modules," CACM, vol. 14, no. 1, April 1972, pp. 221–227.

[ROS75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," IEEE Computer, vol. 8, no. 5, May 1975.

[SHA95a] Shaw, M. and D. Garlan, “Formulations and Formalisms in Software Architecture,” Volume1000—Lecture Notes in Computer Science, Springer-Verlag, 1995.

[SHA95b] Shaw, M. et al., “Abstractions for Software Architecture and Tools to Support Them,” IEEE Trans. Software Engineering, vol. SE-21, no. 4, April 1995, pp. 314–335.

Module 5, Unit 1

[LEA88] Lea, M., "Evaluating User Interface Designs," User Interface Design for Computer Systems, Halstead Press (Wiley), 1988.

[MAN97] Mandel, T., The Elements of User Interface Design, Wiley, 1997.

[MON84] Monk, A. (ed.), Fundamentals of Human-Computer Interaction, Academic Press, 1984.

[MOR81] Moran, T.P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," Intl. Journal of Man-Machine Studies, vol. 15, pp. 3–50.

[MYE89] Myers, B.A., "User Interface Tools: Introduction and Survey, IEEE Software, January 1989, pp. 15–23.

[NOR86] Norman, D.A., "Cognitive Engineering," in User Centered Systems Design, Lawrence Earlbaum Associates, 1986.

[RUB88] Rubin, T., User Interface Design for Computer Systems, Halstead Press (Wiley),1988.

[SHN90] Shneiderman, B., Designing the User Interface, 3rd ed., Addison-Wesley, 1990.

Module 5, Unit 2
http://www.Glearn.net/programming simplified
http://tecfa.unige.ch/moo/book2/node74.html
chapter 11
http://101.lv/learn/access/aba20fi.htm
http://www.stratiss.com/clientsrvrapp.shtml
http://en.wikipedia.org/wiki/Client%E2%80%93server_model

Module 6, Unit 1
http://101.lv/learn/access/aba20fi.htm
http://www.stratiss.com/clientsrvrapp.shtml

Module 6, Unit 2
[1] <http://www.sei.cmu.edu/str/descriptions/middleware.html>
[2] Inmon, William. "A Brief History of Integration." EAI Journal.
[3] Ren, Frances. "The Marketplace of Enterprise Application Integration (EAI). <http://www.public.asu.edu/~mbfr2047/eai.html>
[4] Vander Hey, Dan. "One Customer, One View." Intelligent Enterprise. March 2000.
[5] Yee, Andre. "Demystifying Business Process Integration." EaiQ.
[6]<http://eai.ittoolbox.com/browse.asp?c=EAIPeerPublishing&r=%2Fpub%2Feai%5F overview%2Ehtm>
[7] Newton, Harry. Netwon's Telecom Dictionary. 15th ed. New York: Miller Freeman, Inc., 1999.
[8] <http://www.feer.com/adv/supp/novc.html>
[9] <http://www.wallstreetandtech.com/story/stp/WST20010406S0004>
[10] <http://www.wallstreetandtech.com/story/itWire/INW20020703S0006>
[11] IDC. "The Enterprise Application Integration Market Simmers with Robust Growth Expectations." February 28, 2001.
[12]<http://www.javaworld.com/javaworld/jw-03-1999/jw-03-middleware. html#sidebar1>

Module 6, Unit 3
www.FaaDooEngineers.com

www.wikipedia.com
chapter 15 www.FaaDooEngineers.com

Assignment File

The assignment file will be given to you in due course. In this file you will find all the detail of the work you must submit to your tutor for marks for marking. The marks you obtain for these assignments will count toward the final mark for the course. Altogether, there are tutor marked assignments for this course.

Presentation schedule

The presentation schedule included in this course guide provides you with importance date for completion of each tutor marked assignment. You should therefore endeavour to meet the deadline.

Assignment

There are two aspects to the assessment of this course. First there are tutor marked assignments: and second the written examination. Therefore, you are expected to take note of the fact information and problem solving gathered during the course. The tutor marked assignment must be submitted to your tutor for formal assessments. In accordance to the deadline given. The work submitted will count for 40%of your total course mark. At the end of the course you will need to sit for a final written examination. This examination will account for 60%of your total score.

Tutor Marked Assignment (TMAs)

There are TMAs in this course; you need to submit all the TMAs. The best 10 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possible of extension. Extension will not be granted after the deadline unless on extraordinary cases.

Final Examination and Grading

Final examination for CPT 416 will last for period of 2hours and have a values 60% of the total course grade. The examination will consist of questions which reflect the Self-Assessment Exercise(s) and tutor marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

The following are practical strategies for working through this course.

1. Read the course guide thoroughly
2. Organize a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather all this information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.

3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.
4. Turn to unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.
8. Review the objectives of each study unit to confirm that you have achieved them. If you are not certain about any of the objectives, review the study material and consult your tutor.
9. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to place your study so that you can keep yourself on schedule.
10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have and questions or problems.

After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) ant the course objectives (listed in this course guide).

Tutors and Tutorials

There are 8 hours of tutorial provide in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group. Your tour tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

Don not hesitates to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary:

- You don't not understand any part of the study units or the assigned readings;
- You have difficulty with the self-test or exercise.

- You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should endeavor to attend the tutorials. This is the only opportunity to have face to face contract with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, have some questions handy before attending them. You will learn a lot from participating actively in discussions.

Good luck!

Table of Content

Module 1

Unit 1: The Software Product

Unit 2: The Software Process

Unit 1

The Software Product

Contents

1.0 Introduction

Today software takes on a dual role. It is a product and at the same time the vehicle for delivering a product. As a product it delivers the computing potential embodied by computer hardware. Whether it resides within a cellular phone or operates inside a mainframe computer software is an information transformer producing, managing, acquiring, modifying, displaying or transmitting information that can be as simple as a single bit or as complex as a multimedia simulation. As the vehicle used to deliver the product software acts as the basis for the control of the computer (operation systems) the communication of information (networks) and the creation and control of other programs (software tools and environments) Software delivers what many believe will be the most important product of the twenty-first century information. Software transforms personal data (e.g., an individual financial transactions) so that the data can be more useful in a local context it manages business information to enhance competiveness it provides a gateway to worldwide information networks (e.g., the Internet); and it provides the means for acquiring information in all of its forms.

2.0 Learning Outcome:

In this unit the following will be learnt:

a. Software Characteristics
b. Software Components
c. Software Applications

3.0 Learning Content

3.1 Software

In 1970s less than 1 percent of the public could have intelligently d4escribed what "computer software" meant. Today most professionals and many members of the public at large feel that they understand software. But do they?

A textbook description of software might take the following form:

1. Software is a set of instructions (computer programs) that when executed provide desired function and performance.
2. Software is a data structure that enables the programs to adequately manipulate information.
3. Software is a document that describes the operation and use of the programs.

There is no question that other more complete definitions could be offered. But we need more than a formal definition.

3.Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering) it is important to examine the characteristic of software that make it different from other things that human beings build. When hardware is built the human creative process (analysis, design, construction) is ultimately translated into a physical form. If we build a new computer our initial sketches formal design drawings and bread boarded prototypes evolve into a physical product (VLSI chips, circuit boards, power supplies etc). Software is a logical rather than a physical system element. Therefore, software has characteristic that differ considerably from those of hardware. ***Software is developed or engineered****; it is not manufactured in the classical sense*. Although some similarities exist between software development and hardware manufacture 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 (or easily corrected) for software. Both activities depend on people but the relationship between people applied and work accomplished is entirely different. ***Software doesn't wear out.*** Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve", indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects) defects are corrected, and the failure rate drops to a steady state level (hopeful, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

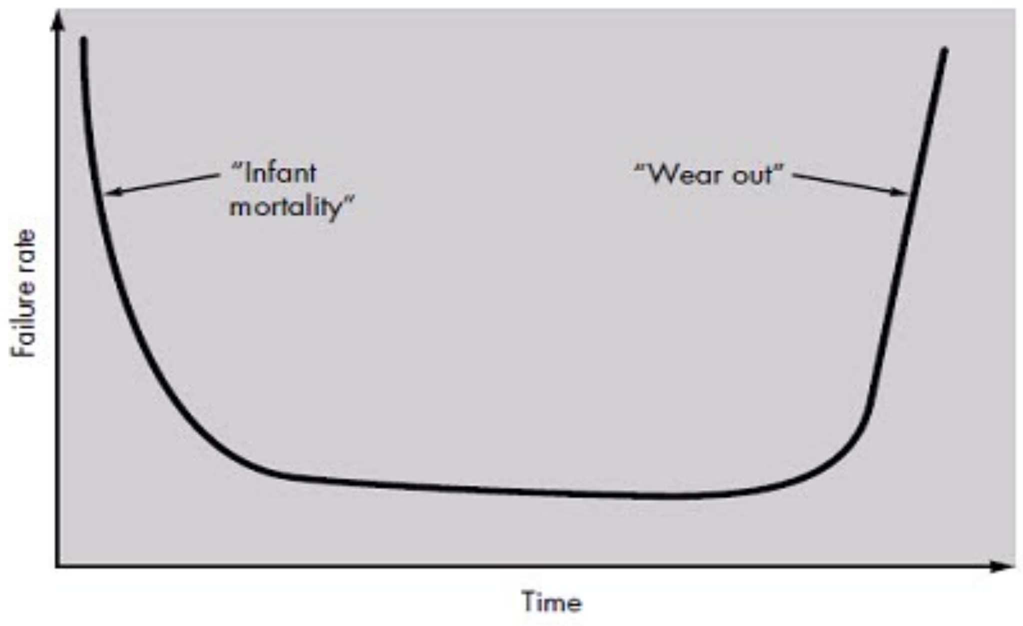


Figure 1.1 Failure curve for hardware

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form shown Figure 1.2 Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (hopefully without introducing other) and the curve flattens as shown as shown Figure 1.2 is a gross over simplification of actual failure models for software. However, the implication is clear software doesn't wear out. But it does deteriorate! This seeming contradiction can best be explained by considering Figure 1.3 During its life software will undergo change (maintenance) As changes are made it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.3 Before the curve can return to the originally steady state failure rate, another change is requested causing the curve to spike again Slowly the minimum failure rate level begins to rise the software is deteriorating due to change.

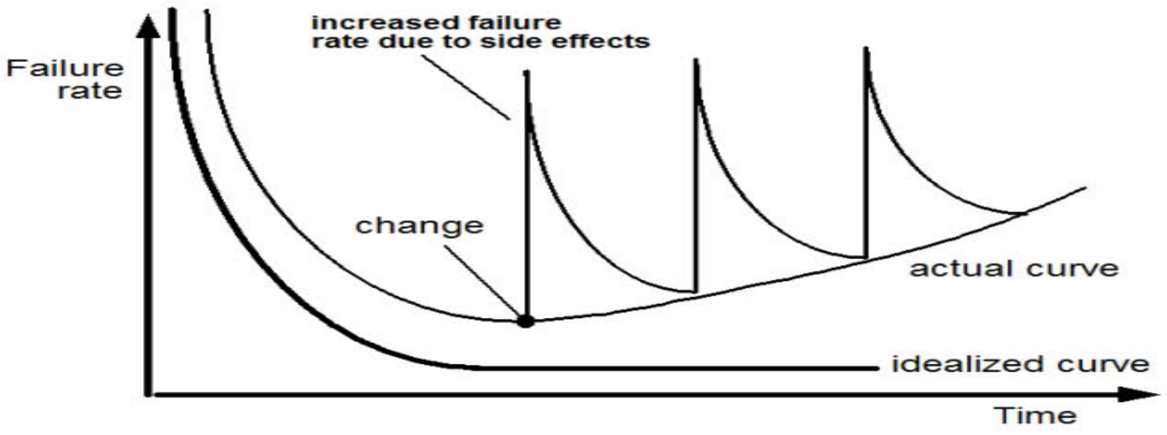


Figure 1.2 Failure curve for software (idealized)

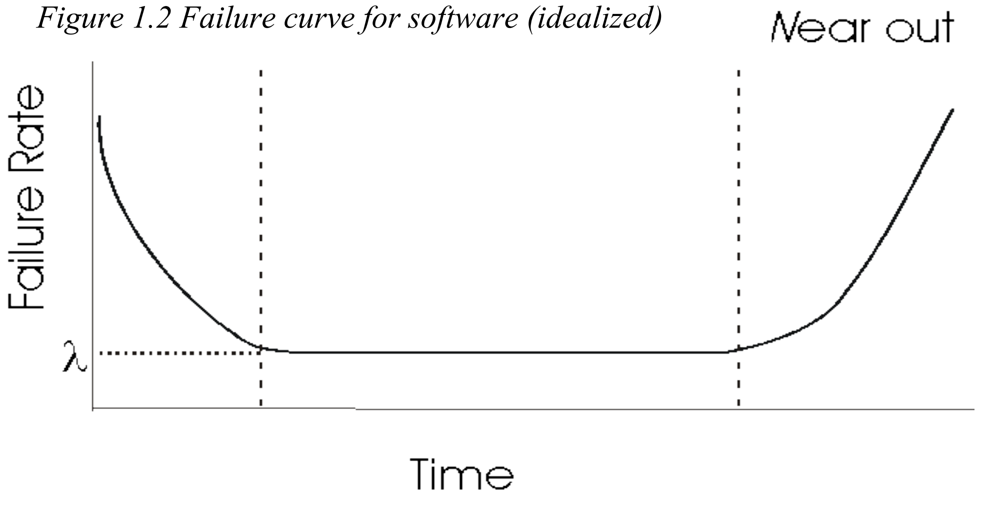


Figure 1.3 Actual failure curve for software

Most software is custom–built rather than being assembled from existing components. Consider the manner in which the control hardware for a microprocessor based product is designed and built. The design engineer draws a simple schematic of the digital circuitry does some fundamental analysis to ensure that proper function will be achieved and then refers to a catalog of digital components. Each integrated circuit (often called an “IC” or a “Chip”) has a part number a defined validated function a well –defined interface and a standard set of integration guidelines. After each component is selected it can be ordered off the shelf. Sadly, software designers are not afforded the luxury described above. With few exceptions there are no catalogs of software components. I t is possible to order off- the –shelf software, but only as a complete unit not as components that can be reassembled into new programs. Although much has been written about “software reusability we are only beginning to see successful implementations of the concept.

3.3 Software Components

As an engineering discipline evolves a collection of standard design components is created. Standard screws and off-the –shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design (i.e., the parts of the design that represent something new). In the hardware world, component reuse is a natural part of the engineering process. In the software world it is something that has yet to be achieved on a broad scale. Reusability is an important characteristic of a high quality software component. A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today we have extended our view of reuse components encapsulate both data and the processing that is applied to the data enabling the software engineer to create new application from reusable parts. For example, today's interactive interfaces are built using reusable components that enable the creation of graphics windows pull-down menus and a wide variety of interaction mechanism. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction. Software components are built using a programming language that has a limited vocabulary an explicitly defined grammar and well-formed rules of syntax and semantics. At the lowest level the language mirrors the instruction set of the hardware. At mid-level programming languages such as Ada 95, C or Smalltalk are used to create a procedural description of the program. At the highest level the language uses graphical icons or other symbols to represent the requirements for a solution. Executable instructions are automatically generated. Machine level language symbolic representation of the CPU instruction set. When a good software developer produces a maintainable well documented program machine level language can make extremely efficient use of memory and "optimize" program execution speed. When a program is poorly designed and has little documentation machine language is a nightmare. Mid-level languages allow the software developer and the program to be machine-independent. When a more sophisticated translator is used, the vocabulary, grammar, syntax and semantics of a mid-level language can be such more sophisticated that machine-level languages. In fact, mid-level language compilers ad interpreters produce machine-level language as output. Although hundreds of programming languages are in use today fewer than ten mid-level programming languages are widely used in the industry. Languages such as COBOL and FORTRAN remain in widespread use more than 30 years after their introduction. More modern programming languages such as Ada95, C, C++, Eiffel, Java and Smalltalk have each gained an enthusiastic following. Machine code assembly languages and mid-level programming languages are often referred to as the first three generation of computer languages. With any of these languages the programmer must be concerned both with the specification of the information structure and the control of the program itself. Hence languages in the first three generations are termed procedural languages. Fourth generation languages also called nonprocedural languages move the software developer even further from the computer hardware. Rather than requiring the developer to specify procedural detail, the nonprocedural language implies a program by "specifying the desired result rather than specifying action required to achieve that result" [COB85]. Support software translates the specification of result into a machine executable program.

3.4 Software Applications

Software may be applied in any situation for which a pre-specified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert systems and artificial neural network software). Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information for example many business applications make use of highly structured input data and produce formatted “reports” Software that controls an automated machine (e.g., numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession. *Information determinacy* refers to the predictability of the order and timing of information. An engineering analysis program accepts data that have a predefined order executes the analysis algorithm without interruption and produces resultant data in report or graphical format. Such applications are determinate.

A multiuser operating system on the other hand accepts inputs that have varied content and arbitrary timing executes algorithms that can be interrupted by external conditions and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate. It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

i. **System Software -**System software is a collection of programs written to service other programs. Some system software (e.g., compiler s editors and file management utilities) processes complex but determinate information structures. Other systems application (e.g., operating system components driver’s telecommunications processors) process largely indeterminate data. In either case the systems software area is characterized by heavy interaction with computer hardware heavy usage by multiple users; concurrent operation that requires scheduling resource sharing and sophisticated process management; complex data structures and multiple external interfaces.

ii. **Real-Time Software -** Programs that monitor/analyze/ control real world events as they occur are called real-time software. Elements of real-time software include a data gathering component that collects and formats information from an external environment an analysis component that transforms information as required by the application a control / output component that responds to the external environment so that real-time response (typically ranging from 1 millisecond to 1 minute) can be maintained. It should be noted that the term “real-time” differs from “interactive” or timesharing”. A real-time system must respond within strict time constraints. The response time of an interactive (or time-sharing) system can normally be exceeded without disastrous results.

iii. **Business Software** Business information processing is the largest single software application area. Discrete “systems” (e.g., payroll accounts receivable/payable inventory, etc.,) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operation or management decision making. In addition to conventional data processing applications, business software applications also encompass interactive and client/server computing (e.g., point-of scale transaction processing).

iv. **Engineering and Scientific Software -**Engineering and Scientific software has been characterized by “number crunching” algorithms. Application range from astronomy to volcano logy from automotive Software Engineering – Concepts & Implementation stress analysis to space shuttle orbital dynamics and from molecular biology to automated manufacturing. However new applications with the engineering/scientific area are moving away from conventional numerical algorithms. Computer aided design system simulation and other interactive applications have begun to take on real-time and even system software characteristics.

v. **Embedded Software-** Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., digital functions in an automobile such as fuel control, dashboard displays, breaking systems, etc.).

vi. **Personal Computer Software -**The personal computer software market has burgeoned over the past decade. Word processing, spreadsheets, computer graphics, multimedia entertainment, database management personal and business financial applications and external network or database access are only a few of hundreds of applications.

vii. **Artificial Intelligence Software** Artificial Intelligence (AI) software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. An active AI area is expert systems also called knowledge-based systems. However other application areas for AI software are pattern recognition (image and voice) theorem proving and game playing. In recent years a new branch of AI software called artificial neural networks, has evolved. A neural network simulates the structure of brain processes (the functions of the biological neuron) and may ultimately lead to a new class of software that can recognize complex patterns and learn from past experience.

3.5 Software: A Crisis on the Horizon

Many industry observers have characterized the problems associated with software development as a “crisis” Yet what we really have may be something rather different. The word “crisis“is defined in Webster’s Dictionary as “ a turning point in the course of anything :decisive or crucial time stage or event “ Yet for software there has been no “turning point” no “decisive time” only slow evolutionary change. In the software industry we have had a “crisis “that has been with us for close to 30 years and that is a contradiction in terms. Anyone who looks up the word “crisis” in the dictionary will find another definition: the turning point in the course of a disease when it becomes clear whether the patient will live or die. “This definition may give us a clue about the real nature of the problems that have plagued software development. We have yet to reach the stage of crisis in computer software. What we really have is a chronic affliction. The word “affliction “is define as anything causing pain or distress” But it is the definition of the adjective “chronic” that is the key to our argument: “lasting a long time or recurring often; continuing indefinitely”. It is far more accurate to describe what we have endured for the past three decades as a chronic affliction rather that a crisis. There are no miracle cures, but there are many ways that we can reduce that pain as we strive to discover a cure.

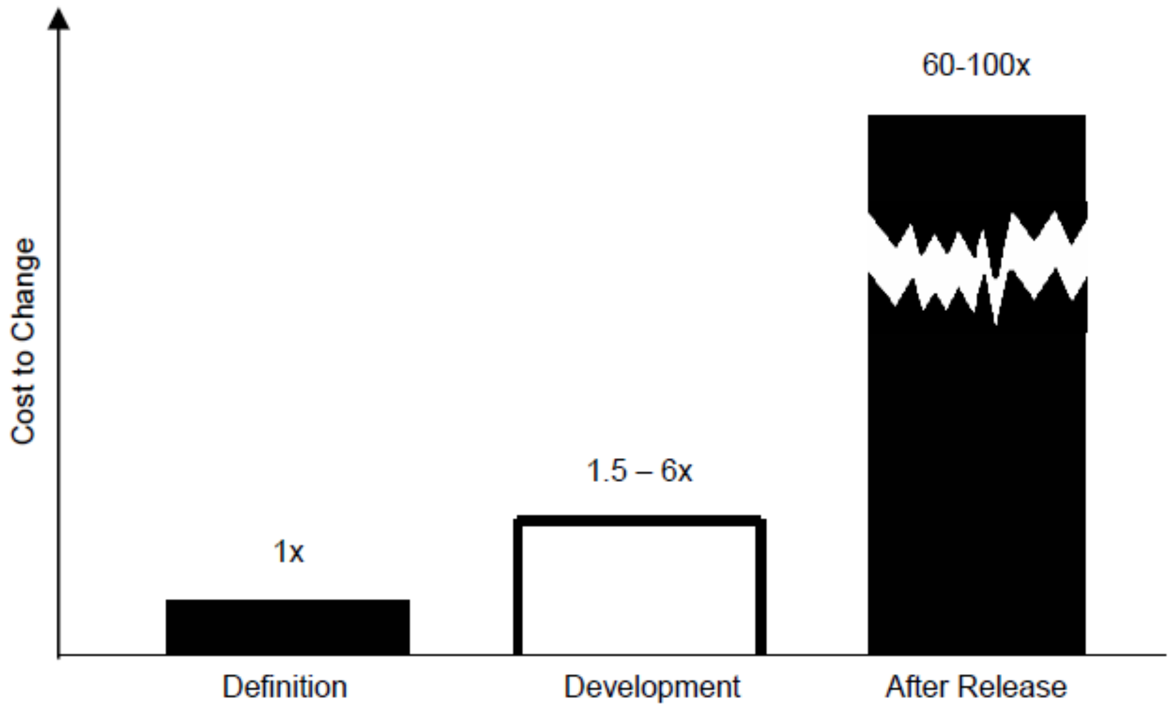


Figure 1.4 The Impact of Change

Whether we call it a software crisis or a software affliction the term alludes to asset or problems that are encountered in the development of computer software. The problems are not limited to software that “doesn’t function properly” Rather, the affliction encompasses problems associated with how we develop software how we maintain a growing volume of existing software and how we can expect to keep pace with a growing demand for more software. Although reference to a crisis or even an affliction can be criticized for being melodramatic the phrases do serve a useful purpose by denoting real problems that are encountered in all area of software development.

Self-Assessment Exercise(s)

What is a software failure?

A. What types of software might be useful to an organization that wants to make resources more findable?

B. What types of software might be useful to an organization that wants to make resources more findable?

Self-Assessment Answer(s)

A. What is a software failure?

system failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function and/or service". This definition applies to both hardware and software system failures. Faults or bugs in a hardware or a software component cause error. An error is defined as that part of the system which is liable to lead to subsequent failure, and an error affecting the service is an indication that a failure occurs or has occurred. If the system comprises of multiple components, errors can lead to a component failure. As various components in the system interact, failure of one component might introduce one or more faults in another. Figure below shows this cyclic behavior.

B. What types of software might be useful to an organization that wants to make resources more findable?

Enterprise search: If you're like most organizations, you have useful information and resources scattered among different file servers, intranets, the company Web site, and more. A number of vendors (such as Google and Microsoft) offer search solutions that allow keyword searching across a multitude of sources and formats.

Tagging solutions: Asking staff members to "tag" documents with keywords, and then allowing others to browse by those keywords can be an affordable way to surface key documents. De.licio.us and Flikr offer free functionality that allows you to tag resources in a public environment. Several vendors also offer tagging solutions appropriate for organization use. Note, however, that no tagging solution is likely to make all your documents easily findable unless every staff member tags nearly everything he or she touches.

Intranets and shared document spaces: Consider providing areas where teams or experts can upload resources for others to use. These areas might take the form of an organization intranet (built through tools like Drupal or Share point) or shared document spaces in a group collaboration tool (consider Google Groups or Web Office). Keep in mind, though, that a staff member isn’t likely to voluntarily upload a lot of documents unless it saves her a lot of time to do so.

Content management systems: If you formalize your methods of tagging and uploading documents, you enter the realm of content management. A content management system (CMS) is used to display text and documents on a website. While these applications are most familiar as a way to update text, more sophisticated CMS tools have powerful mechanisms for organizing and browsing through documents which make them very applicable in this space. Consider tools like Drupal, Plone, or Common Spot.

4.0 Summary/Conclusion

Software has become the key element in the evolution of computer based systems and products. Over the past four decades, software has evolved from a specialized problem-solving and information analysis tool to an industry in itself. But early ‘programming ‘culture and history have created a set of problems that persists today. Software has become a limiting factor in the evolution of computer based systems. Software is composed of programs, data, and documents. Each of these items comprises a configuration that is created as part of the software engineering process. The intent of software technique is to provide a framework for building software with higher quality

5.0 Tutor-Marked Assignments

1. Write a short note on software components.
2. Write a short note on software applications.

6.0 References/Further readings

Brooks, F., The Mythical Man-Month, Addison-Wesley, 1975.

[DEJ98] De Jager, P. et al. Countdown Y2K: Business Survival Planning for the Year 2000, Wiley, 1998.

[DEM95] DeMarco, T., Why Does Software Cost So Much? Dorset House, 1995, p. 9.

[FEI83] Feigenbaum, E.A. and P. McCorduck, The Fifth Generation, Addison- Wesley, 1983.

Unit 2

The Software Process

Contents

1.0 Introduction

Software techniques and development is performed by creative, knowledgeable people who should work within a defined and mature software process. The intent of this lecture is to provide a survey of the current state of the software process and to provide pointers to more detailed discussion of management.

2.0 Learning Outcome:

In this unit the following will be learnt:

a. Software Engineering and Techniques.
b. Software Process
c. Software Process Model

Learning Content

3.1 Software Engineering – Layered Technology

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer at the seminal conference on the subject still serves as a basis for discussion:

"*Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."*

Almost every reader will be tempted to add to this definition. It says little about the technical aspects of software quality; it does not directly address the need for customer satisfaction or timely product delivery; it omits mention of the importance of measurement and metrics; it does not state the importance of amateur process. And yet, Bauer's definition provides us with a baseline. What are the "sound engineering principles" that can be applied to computer software development? How do we "economically" build software so that it is "reliable"? What is required to create computer programs that work "efficiently" on not one but many different "real machines"? These are the questions that continue to challenge software engineers.

The IEEE [IEE93] has developed a more comprehensive definition when it states: Software Engineering The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software that is, the application of engineering to software.

3.2 Process, Methods, and Tools

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a focus on quality. The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.

Process defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed.

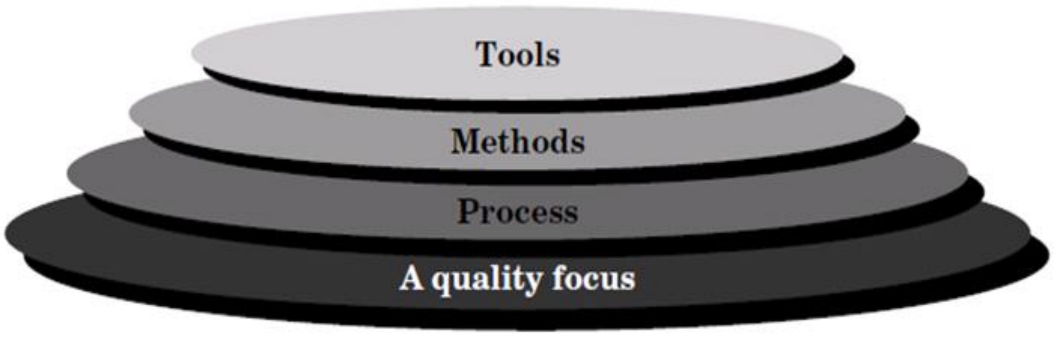


Figure 2.1 software engineering layers

Software engineering methods provide the technical "how to's" for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and maintenance. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called **computer-aided software engineering**, is established. CASE combines software, hardware, and a software engineering database to create a software engineering environment that is analogous to CAD/CAE for hardware.

3.3 A Generic View of Software Technique

Technique is the systematic procedure by which a complex or scientific task is accomplished. With regards to SOFTWARE, it is the method used for the analysis, design, construction, verification, and management of entities. Regardless of the software that is to be developed, the following questions must be asked and answered:

• What is the problem to be solved?

• What is the process that will be employed to solve the problem?

• How will the end result?

• How will it be designed?

• What approach will be used to uncover errors that were made in the design of the software?

• How will the software be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the software?

The work that is associated with software engineering can be categorized in to three generic phases, regardless of application area, project size, or complexity. Each phase addresses one or more of the questions noted above.

The definition phase focuses on what. That is, during definition, the software developer attempts to identify what information is to be processed what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the software are identified. Although the methods applied during the definition phase will vary depending upon the software engineering paradigm (or combination of paradigms) that is applied, three major tasks will occur in some form. System or information engineering software project, planning and requirements analysis.

The development phase focuses on how*.* That is during development a software engineer attempts to define how data are to be structured, how function is to be implemented as a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language) and how testing will be performed. The methods applied during the development phase will vary, but three specific technical tasks should always occur software design code generation and software testing.

The maintenance phase focuses on change that is associated with error correction, adaptations required as the software's environment evolves and changes due to enhancements brought above by changing customer requirements. The maintenance phase reapplies the steps of the definition and development phases, but does so in the context of existing software.

Four types of change are encountered during maintenance phase.

Correction: Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software, corrective maintenance changes the software to correct defects.

Adaptation: Over time, the original environment (e.g. CPU, operating system, business rules external product characteristics) for which the software was developed is likely to change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.

Enhancement: As software is used, the customer/user will recognize additional functions that will provide benefit, perfect maintenance extends the software beyond its original functional requirements.

Prevention: Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapter, and enhanced. Today, the "aging software plant" is forcing many companies to pursue software reengineering techniques. In a global sense, software development is often considered as part of business process. The phases and related steps described in the generic view of software techniques are complemented by a number of umbrella activities. Typical activities in this category include:

- Software project tracking and control
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Document preparation and production
- Reusability management
- Measurement
- Risk management

3.4 The Software Process

A software process can be characterized as shown in fig 2.2. 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.

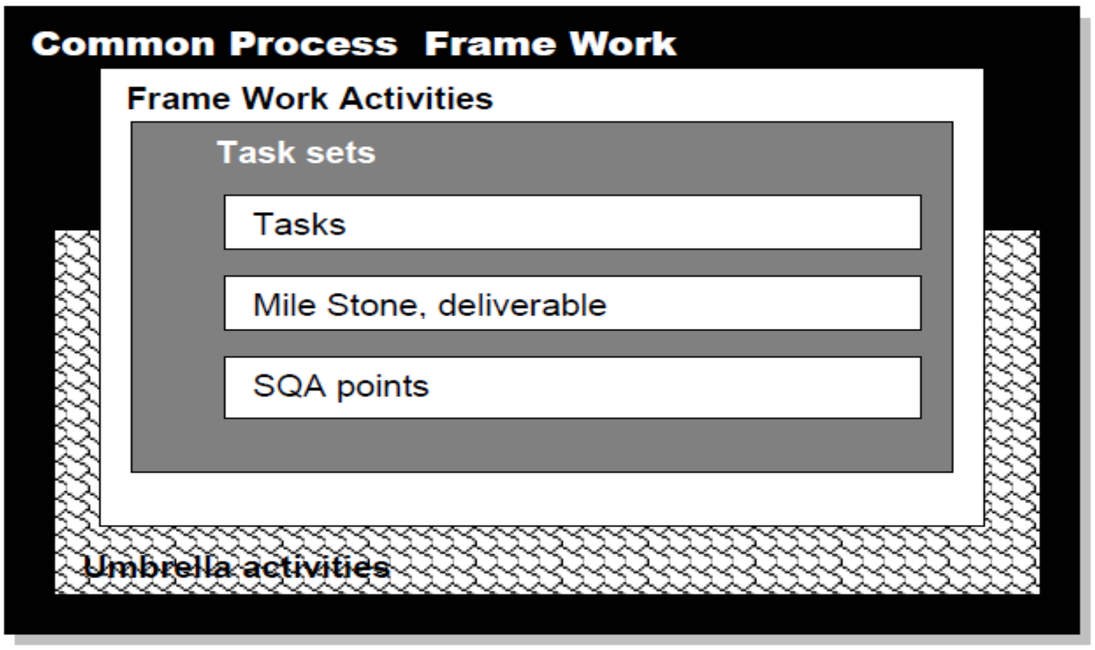


Figure 2.2 The Software Process

In recent years, there has been a significant emphasis on “process maturity” [PAU 93]. The Software Engineering Institute (SEI) has developed a comprehensive model that is predicated on a set of software technique capabilities that should be present as organizations reach different levels of process maturity.

To determine an organization’s current state of process maturity, the SEI uses an assessment questionnaire and a five-point grading scheme. The grading scheme determines compliance with a capability maturity model [PAU93] that defines key activities required at different levels of process maturity. The SEI approach provides a measure of the global effectiveness of a company’s software technique practices and established five process maturity levels, which are defined in the following manner:

Level 1: Initial –The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: Repeatable – Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: Defined – The software process for both management and engineering activities is documented, standardized and integrated into an organization wide software process. All projects use a document and approved version of the organization’s process for developing and maintaining software. This level includes all characteristics defined for level 3.

Level 4: Managed – Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

Level 5: Optimizing – Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

The five levels defined by the SEI are derived as a consequence of evaluating responses to the SEI assessment questionnaire. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization’s process maturity.

The SEI has associated key process areas with each of the maturity levels. The KPAs describe those software engineering functions (e.g. software project planning, requirements management) that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics.

- Goals the overall objectives that the KPA must achieve.
- Commitments requirements (imposed on the organization) that must be met achieve the goals, and that provide proof of intent to comply with the goals.
- Abilities those things that must be in place (organizationally and technically) that will enable that organization to meet the commitments.
- Activities the specific tasks that are required to achieve the KPA function.
- Methods for monitoring implementation – the manner in which the activities are monitored as they are put into place.

- Methods for verifying implementation – the manner in which proper practice for the KPA can be verified. Eighteen KPAs (each described using the structure noted above) are defined across the maturity model and are mapped into different levels of process maturity. The following KPAs should be achieved at each process maturity level:

Process maturity level 2

⊥ Software configuration management

⊥ Software quality assurance

⊥ Software subcontract management

⊥ Software project tracking and oversight

⊥ Software project planning

⊥ Requirement management

Process maturity level 3

⊥ Peer reviews

⊥ Inter group coordination

⊥ Software product engineering

⊥ Integrated software management

⊥ Training program

⊥ Organization process definition

⊥ Organization process focus

Process maturity level 4

⊥ Software quality management

⊥ Quantitative process management

⊥ Process maturity level 5

⊥ Process change management

⊥ Technology change management

⊥ Defect prevention

Each of the KPAs is defined by a set of key practices that contribute to satisfying its goals. The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted. The SEI defines key indicators as “those key practices or components of key practices that offer the greatest insight into whether the goals of a key process area have been achieved. Assessment questions are designed to probe for the existence (or lack thereof) of a key indicator.

3.5 Software Process Models

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers described in previous section and the generic phases also discussed in previous section. This strategy is often referred to as a process model or a software engineering paradigm. A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. In an intriguing paper on the nature of the software process, L.B.S. raccoon [RAC95] uses fractals as the basis for a discussion of the true nature of the software process.

All software development can be characterized as a problem solving loop (figure 2.3a) in which four distinct stages are encountered; status quo, problem definition, technical development, and solution integration. Status quo "represents the current state of affairs"; problem definition identifies the specific problem to be solved; technical development solves the problem through the application of some technology, and solution integration delivers the results who requested the solution in the first place.

The problem solving loop described above applies to software engineering work at many different levels of resolution. It can be used at the macro level when the entire application is considered; at a middle level when program components are being engineered, and even at the line of code level. Therefore, a figure 2.3b, each stage in the problem solving loop contains an identical problem solving loop, which contains still another problem solving loop (this continues to some rational boundary; for software, a line of code)

Realistically, it is difficult to compartmentalize activities as neatly as figure 2.3b implies because cross talk occurs within and across stages, yet this simplified view leads to a very important idea; regardless of the process model that is chosen for a software project, all of the stages – status quo, problem definition, technical development, and solution integration-coexist simultaneously at some level of detail. Given the recursive nature figure 2.3b the four stages discussed above apply equally to the analysis of a complete application and to the generation of a small segment of code.

Raccoon suggests a "Chaos model"; that describes "software development a continuum from the user to the developer to the technology". As work progresses toward a complete system, the stages described above are applied recursively to user needs and the developer's technical specification of the software.

3.6 Linear Sequential Model

Figure 2.4 illustrates that linear sequential model for software engineering. Sometimes called the "classic life cycle" or the "waterfall model", the linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design coding, testing, and maintenance. Modeled after the conventional engineering cycle, the linear sequential model encompasses the following activities.

System / information engineering and modeling: Because software always part of a larger system (or business), work begins by establishing requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when software must interface with other elements such as hardware, people, and gathering at the system level with a small amount of top level analysis and design. Information engineering encompasses requirements gathering the strategic business level and at the business area level.

Software requirements analysis: The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer must understand the information domain for the software, as well as required function, behavior, performance, and interfacing requirements for both the system and the software are documented and reviewed with the customer.

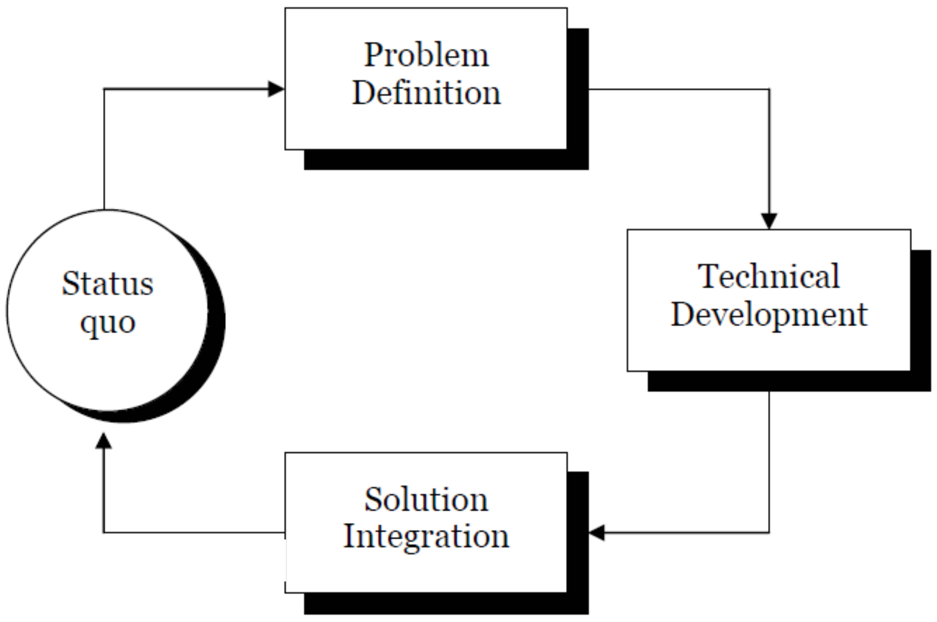


Figure 2.3a The phases of a problem solving loop

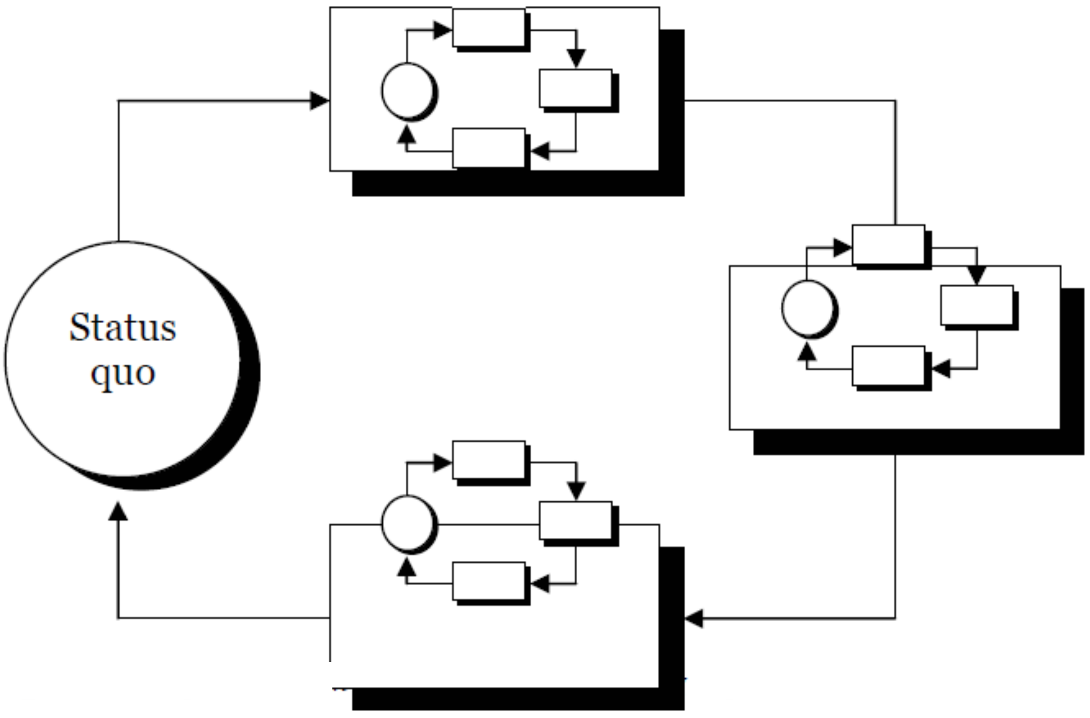


Figure 2.3b the phases within phases of the problem solving loop

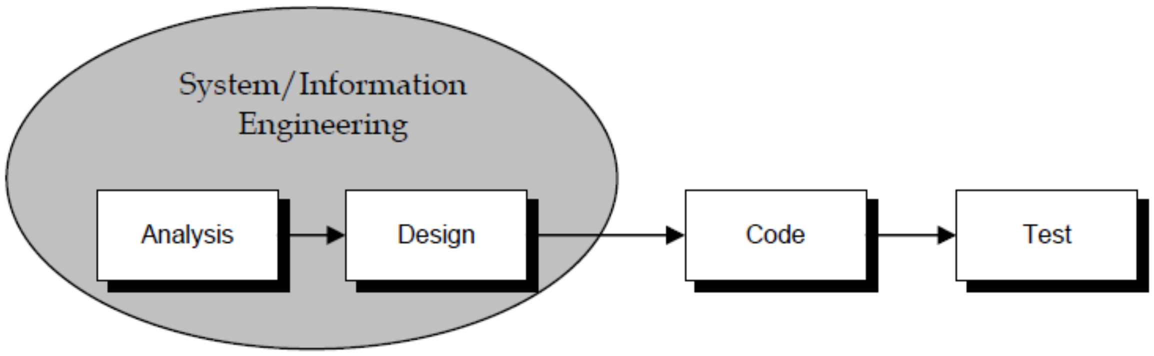


Figure 2.4 The linear sequential model.

Design: Software design is actually a multistep process that focuses of four distinct attributes of a program; data structure, software architecture interface representations, and procedural detail. The design process translates requirements into a representation of the software that can be assessed for quality before code generation begins. Like requirements, the design is documented and becomes part of the software configuration.

Code generation: The design must be translated into a machine readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing: Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, assuring that all statements have been tested, and on the functional externals that is conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required result.

Maintenance: Software will undoubtedly undergo change after it is delivered to the customer. Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g. a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software maintenance reapplies each of the preceding phases to an existing program rather than a new one.

The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticisms of the paradigm have caused even active supporters to question its efficacy. Among the problems that are sometimes encountered when the linear sequential model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

4. Developers are often delayed unnecessarily, in an interesting analysis of actual projects, Bradac found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking states tend to be more prevalent at the beginning and end of a linear sequential process.

Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and maintenance can be placed. The classic life cycle remains the most widely used process model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development.

3.7 Evolutionary Software Process Model

There is growing recognition that software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

The linear sequential model is designed for straight line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

3.8 The Incremental Model

The incremental model combines elements of the linear sequential model with the iterative philosophy of prototyping. As figure 2.5 shows, the incremental model applies linear sequences in a staggered fashion as calendar time progress. Each linear sequence produces as deliverable "increment" of the software. For example, word processing software developed using the incremental paradigm might deliver basic 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 can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features, (some known, others 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. Early increments are "stripped down" versions of the final product but they do provide capability that serves the user and also provide a platform for evaluation by the user.

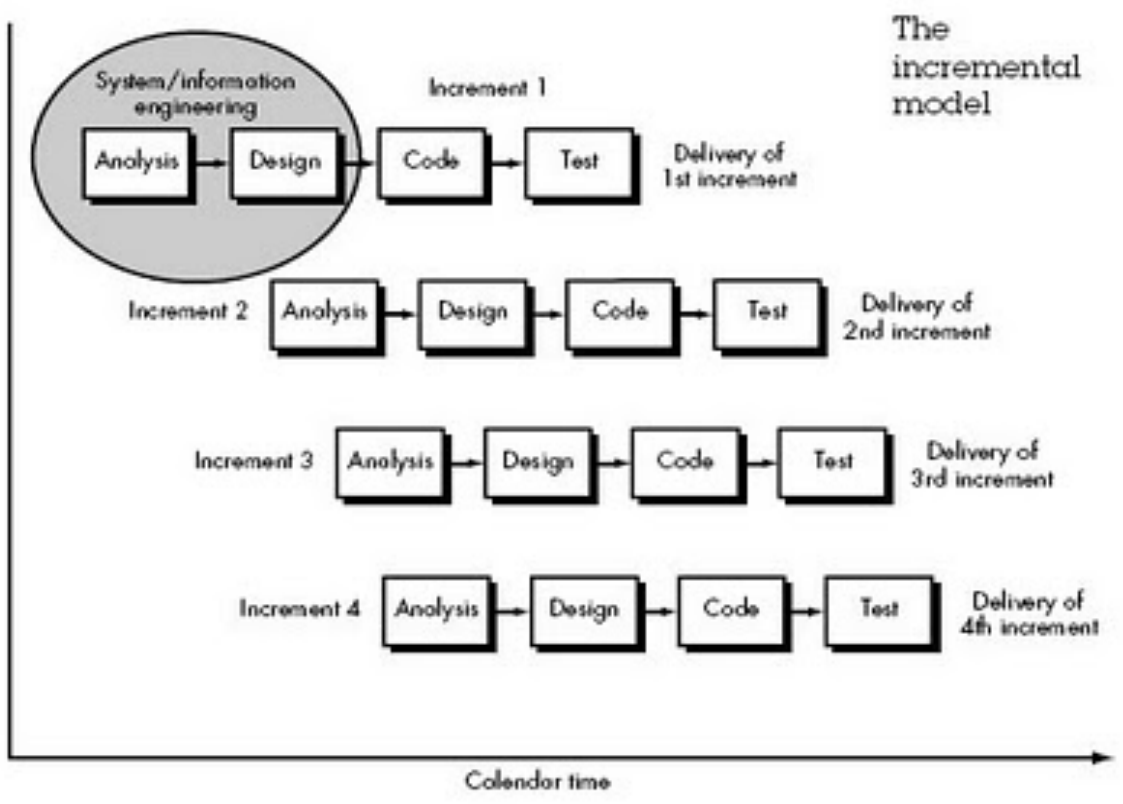


Figure 2.5 The incremental model

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. In might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

3.9 The Formal Methods Model

The formal methods model encompasses a set of activities that lead to mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer based system by applying a rigorous, mathematical notation. A variation on this approach, called clean room software engineering, is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms, ambiguity, in completeness, and inconsistency can be discovered and corrected more easily not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might otherwise go undetected. Although not yet a mainstream approach, the formal methods model offers the promise of defect free software. Yet, concern about its applicability in a business environment has been voiced.

1. The development of formal models is currently quire time consuming and expensive.
2. Because few software developers have the necessary background to apply formal methods, extensive training is required.
3. It is difficult to use the models as a communication mechanism for technically unsophisticated customers. These concerns notwithstanding, it is likely that the formal methods approach will gain adherents among software developers that must built safety critical software and among developers that would suffer severe economic hardship should software errors occur.

Self-Assessment Exercise(s)

A. Explain the main layers of the software engineering layers

Self-Assessment Answer(s)

Software Engineering can be viewed as a layered technology. The main layers are:-

Process layer: - It is an adhesive that enables rational and timely development of computer software. It defines an outline for a set of key process areas that must be acclaimed for effective delivery of software engineering technology.

Method layer: - It provides technical knowledge for developing software. This layer covers a broad array of tasks that include requirements, analysis, design, program construction, testing and support phases of the software development.

Tools layer: - It provides computerized or semi-computerized support for the process and method layer. Sometimes, tools are integrated in such a way that other tools can use the information created by one tool. This multi-usage is commonly referred to as computer-aided software engineering or CASE. Case combines software, hardware and software engineering database to create software engineering analogous to computer-aided design or CAD for hardware. CASE helps in application development including analysis, design, code generation, and debugging and testing etc.

4.0 Summary/Conclusion/Conclusion

Software engineering is a discipline that integrates process, methods, and tools for the development of computer software. A number of different process models for software engineering have been proposed, each exhibiting strengths and weaknesses, but all having a series of generic phases in common. The principles, concepts, and methods that enable us to perform the process that we call software engineering are considered throughout the remainder of this book.

5.0 Tutor-Marked Assignments

1. Explain Software Process

2. Write a note on Incremental Model
3. Write a note on Linear Sequential Model.

6.0 References/Further readings

Brooks, F., The Mythical Man-Month, Addison-Wesley, 1975.

[DEJ98] De Jager, P. et al., Countdown Y2K: Business Survival Planning for the Year 2000, Wiley, 1998.

[DEM95] DeMarco, T., Why Does Software Cost So Much? Dorset House, 1995, p. 9.

[FEI83] Feigenbaum, E.A. and P. McCorduck, The Fifth Generation, Addison-Wesley, 1983, softwareengineering.com

Module 2

Unit 1

The Project Management Concept

Contents

1.0 Introduction

In this lecture we are going to learn about Management spectrum and people, the problem and the process in the Management spectrum.

2.0 Learning Outcome:

In this unit the following will be learnt:

a. About Management Spectrum

b. About People

c. About Problem

d. About Process

3.0 Learn Content

The Management Spectrum

Effective software project management focuses on the three P’s: people, problem, and process. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive customer communication early in the evolution of a project risks building an elegant solution for the wrong problem. Finally, the manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum.

3.1.1 The People

The cultivation of motivates, highly skilled software people has been discussed since the 1960s In fact, the “people factor” is so important that the Software engineering Institute has developed a people management capability maturity model “to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability”.

The people management maturity model defines the following key practice areas for software people career development, selection performance management, training, compensation, career development, organization and work design and team culture development. Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implement effective software engineering practices.

3.1.2 The Problems

Before a project can be planned, its objectives and scope should be established, alternative solutions should be considered and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost; an effective assessment of risk; a realistic breakdown of project tasks; or a manageable project schedule that provides a meaningful indication of process.

The software developer and customer must meet to define project objectives and scope. In many cases, this activity begins as part of the system engineering process and continues as the first step in software requirements analysis. Objectives identify the overall goals of the project without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the problem, and more important attempts to bind these characteristics in a quantitative manner.

Once the project objectives and scope are understood, alternative solutions are considered. Although very little details is discussed, the alternatives enable managers and practitioners to select a “best” approach, given the constraints imposed by delivery deadlines, budgetary restrictions personnel availability, technical interfaces, and myriad other factors.

3.1.3 The Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets tasks, milestones, 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.

3.2 People

In a study published by the IEEE, the engineering vice presidents of three major technology companies were asked the most important contributor to a successful software project. They answered in the following way.

VP 1: I guess if you had to pick one thing out that is most important in our environment, I’d say it’s not the tools that we use, it’s the people.

VP 2: The most important ingredient that was successful on this project was having smart people. Very little else matters in my opinion. The most important thing you do for a project is selecting the staff. The success of the software development organization is very, very much associated with the ability to recruit good people.

VP3: The only rule I have in management is to ensure I have good people real good people and that I grow good people and that I provide an environment in which good people can produce.

Indeed, this is a compelling testimonial on the importance of people in the software engineering process. And yet, all of us, from senior engineering vice presidents to the lowliest practitioner, often take people for granted. Managers argue (as the group above had done) that people are primary, but their actions sometimes belie their words. In this section we examine the players who participate in the software process and the manner in which they are organized to perform effective engineering.

3.2.1 The Players

The software process (and every software project) is populated by players who can be categorized into one of five constituencies.

1. Senior managers, who define the business issues that often have significant influence on the project.
2. Project (technical) managers, who must plan, motivate, organize, and control the practitioners who do software work.
3. Practitioners, who deliver the technical skills that, are necessary to engineer a product or application.
4. Customers, who specify the requirements for the software to be engineered.
5. End users, who interact with the software once it is released for productions use.

Every software project is populated by the players noted above. To be effective, the project team must be organized in a way that maximizes each person's skills and abilities. That's the job of the team leader.

What do we look for when we select someone to lead a software project? In an excellent book of technical leadership, Jerry Weinberg [WEI86] attempts to answer this question by suggesting the MOI Model leadership:

Motivation: The ability to encourage (by "push or pull") technical people to produce to their best ability.

Organization: The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.

Ideas or innovation: The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Weinberg suggests that successful project leaders apply a problem solving management style. That is, a software project manager should concentrate on understanding the problem to be solved, managing the flow of ideas, and at the same time letting everyone on the team know (by words, and far more important, by actions) that quality counts and that it will not be compromised.

Another view of the characteristics that define an effective project manager emphasizes four key traits.

Problem solving: An effective software project manager can diagnose that technical and organizational issues that are most relevant, systematically structure a solution or properly motivate other practitioners to develop the solution, apply lessons learned from past projects to new situations, and remain flexible enough to change direction if initial attempts at problem solution are fruitless.

Managerial identity: A good project manager must take charge of the project. She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement: To optimize the productivity of a project team, a manager must reward initiative and accomplishment, and demonstrate through his own actions that controlled risk taking will not be punished.

Influence and Team Building: An effective project manager must be able to read people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high stress situations.

3.2.2 The Software Team

There are almost as many human organizational structures for software development as there are organizations that develop software. For better or worse, organizational structure cannot be easily modified. Concern with the practical and political consequences of organizational change is not within the software project manager's scope of responsibilities. However, the organization of the people directly involved in a new software project is within the project manager's purview.

The following options are available for applying human resources to a project that will require n people working for k years:

1. *n* individuals are assigned to m different functional tasks; relatively little combined work occurs coordination is the responsibility of a software manager who may have six other projects to be concerned with.
2. *n* individuals are assigned to m different functional tasks, *(m<n)*so that informal "teams" are established; an ad hoc team leader may be appointed; coordination among teams is the responsibility of a software manager;
3. *n* individuals are organized into t teams; each team is assigned one or more functional tasks; each team has a specific structure that is defined for all teams working on a project coordination is controlled by both the team and a software project manager.

Although it is possible to voice pro and con arguments for each of the above approaches, there is a growing body of evidence that indicates that a formal team organization (option 3) is most productive. The "best" team structure depends on the management style of an organization, the number of people who will populate the team and their skill levels and the overall problem difficulty. Mantei, suggests three generic team organizations:

Democratic decentralized (DD) - This software engineering team has no permanent leader. Rather," task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks." Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.

Controlled decentralized (CD) - This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtask. Problem solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader.

Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

Controlled Centralized (CC) - Top-level problem solving and internal team coordination are managed by a team leader. Communication between the leader and team members is vertical. Mantei also describes seven project factors that should be considered when planning the structure of software engineering teams:

- The difficulty of the problem to be solved
- The time the team will stay together (team lifetime)

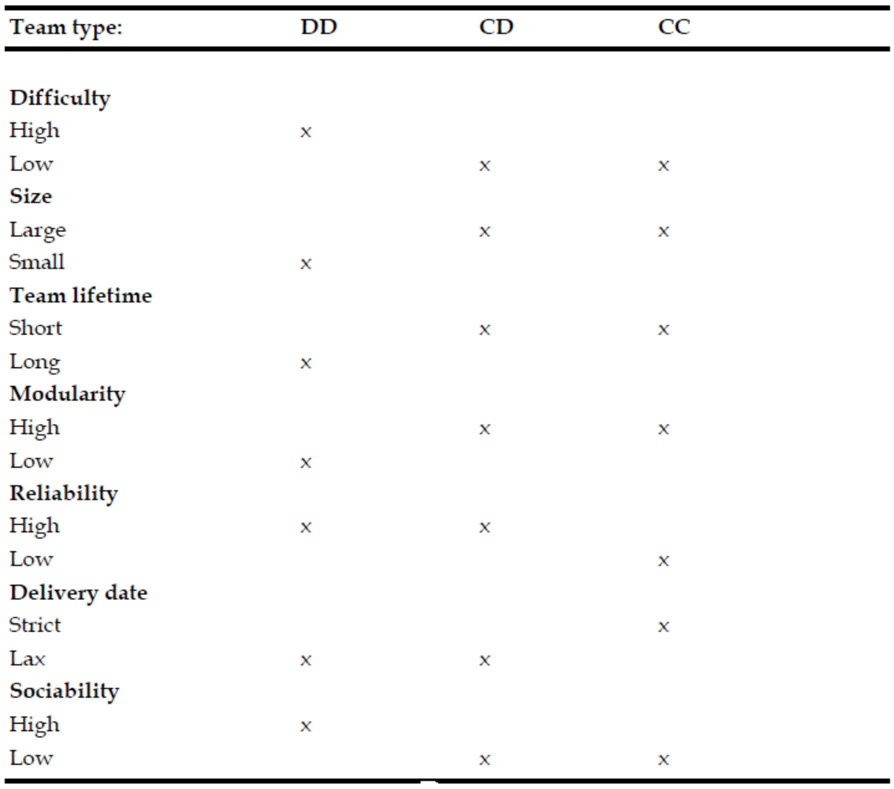
- The degree to which the problem can be modularized
- The required quality and reliability of the system to be built
- The rigidity of the delivery date
- The degree of sociability (communication) required for the projects

Table 3.1 summarizes the impact of project characteristics on team organization. Because a centralized structure completes tasks faster, it is the most adept at handling simple problems.

Decentralized teams generate more and better solutions than individuals. Therefore, such teams have a greater probability of success when working on difficult problems. Since the CD team is centralized for problem solving, either the CD or the CC team structure can be successfully applied to simple problems. A

DD structure is best for difficult problems. Because the performance of a team is inversely proportional to the amount of communication that must be conducted, very large projects are best addressed by teams with a CC or CD structure when sub grouping can be easily accommodated.

The impact of project characteristics on team structure



Team type:	DD	CD	CC
Difficulty			
High	x		
Low		x	x
Size			
Large		x	x
Small	x		
Team lifetime			
Short		x	x
Long	x		
Modularity			
High		x	x
Low	x		
Reliability			
High	x	x	
Low			x
Delivery date			
Strict			x
Lax	x	x	
Sociability			
High	x		
Low		x	x

Table 3.1: The impact of project characteristics on team structure

The length of time the team will “live together” affects team morale. It has been found that DD team structures result in high morale and job satisfaction and are therefore good for long lifetime teams. The DD team structure is best applied to problems with relatively low modularity because of the higher volume of communication that is needed. When high modularity is possible (and people can do their own thing) the CC or CD structure will work well.

CC and CD teams have been found to produce fewer defects than DD teams, but these data have much to do with the specific quality assurances activities that are applied by the team. Decentralized teams generally require more time to complete a project than a centralized structure and at the same time are best when high sociability is required. Constantine suggests four "organizational paradigms" for software engineering teams:

1. A closed paradigm structures a team along a traditional hierarchy of authority (similar to a CC team). Such teams can work well when producing software that is quite similar to past efforts, but they will be less likely to be innovative when working within the closed paradigm.
2. The random paradigm structures a team loosely and depends on individual initiative of the team members. When innovation or technological breakthrough is required, teams following the random paradigm will excel. But such teams may\ struggle when "orderly performance" is required.
3. The open paradigm attempts to structure a team in manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm. Work is performed collaboratively with heavy communication and consensus based decision making. Open paradigm team structures are well suited to the solution of complex problems, but may not perform as efficiently as other teams.
4. The synchronous paradigm relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problems with little active communication among themselves.

As an historical footnote, the earliest software team organization was a controlled centralized (CD) structure originally called the chief programmer team. This structure was first proposed by Harlan Mills and described by Baker]. The nucleus of the team is composed of a senior engineer ("the chief programmer") who plans, coordinates and reviews all technical activities of the team; technical staff (normally two to five people) who conduct analysis and development activities and a backup engineer who supports the senior engineer in project continuity.

The chief programmer may be served by one or more specialists (e.g. telecommunications expert, database designer), support staff (e.g., technical writers, clerical personal) and a *software librarian*. The librarian serves many teams and performs the following functions: maintains and controls all elements of the software configuration (i.e., documentation, source listings, data, magnetic media); helps collect and format software productivity data; catalogs and indexes reusable software modules; and assists the teams in research, evolution, and document preparation. The importance of a librarian cannot be overemphasized.

The librarian acts as a controller, coordinator and potentially, an evaluator of the software configuration. Regardless of team organization, the objective for every project manager is to help create a team that exhibits cohesiveness.

But many of these groups just don't seem like teams. They don't have a common definition of success or any identifiable team spirit. What is missing is a phenomenon that we call *jell*. A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts…Once a team begins to jell, the probability of success goes way up. The team can become unstoppable, a juggernaut for success… they don't need to be managed in the traditional way, and they certainly don't need to be motivated. They've got *momentum.*

DeMarco and Lister contend that members of jelled teams are significantly more productive and more motivated than average. They share a common goal, a common culture, and in many cases, a "sense of elatedness" that makes them unique.

3.2.3 Coordination and Communication Issues

There are many reasons that software projects get into trouble. The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. *Uncertainly* is common, resulting in a continuing stream of changes that ratchets the project team. *Interoperability* has become a key characteristic of many systems. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

These characteristics of modern software-scale, uncertainty, and interoperability-are facts of life. To deal with them effectively, a software engineering team must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings and other relatively non-interactive and impersonal communication channels". Informal communication is more personal. Members of a software engineering team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another daily.

Kraul and Streeter examine a collection of project coordination techniques that are categorized in the following manner.

Formal, impersonal approaches - Include software engineering documents and deliverables technical memos, project milestones, schedules and project control tools changes requests and related documentation error tracking reports, and repository data.

Formal, interpersonal procedures - Focus on quality assurance activities applied to software engineering work products. These include status review meetings and design and code inspections.

Informal, interpersonal procedures - Include group meetings for information dissemination and problem solving and "collocation of requirements and developments staff".

Electronic communication - Encompasses electronic mail, electronic bulletin boards, Web sites, and by extension, video-based conferencing systems.

Interpersonal network - Informal discussion with those outside the project who may have experience or insight that can assist team members.

3.3 The Problem

A software project manager is confronted with a dilemma at the very beginning of a software engineering project. Quantitative estimates and an organized plan are required, but solid information is unavailable.

A detailed analysis of software requirements would provide necessary information for estimates, but analysis often takes weeks or months to complete. Worse requirements may be fluid, changing regularly as the project proceeds. Yet, a plan is needed "now". Therefore, we must examine the problem at the very beginning of the project. At a minimum, the scope of the problem must be established and bounded.

3.3.1 The Software Scope

The first software project management activity is the determination of software scope. Scope is defined by answering the following questions:

Context: how does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?

Information objectives: What customer visible data objects are produced as output from the software? What data objects are required for input?

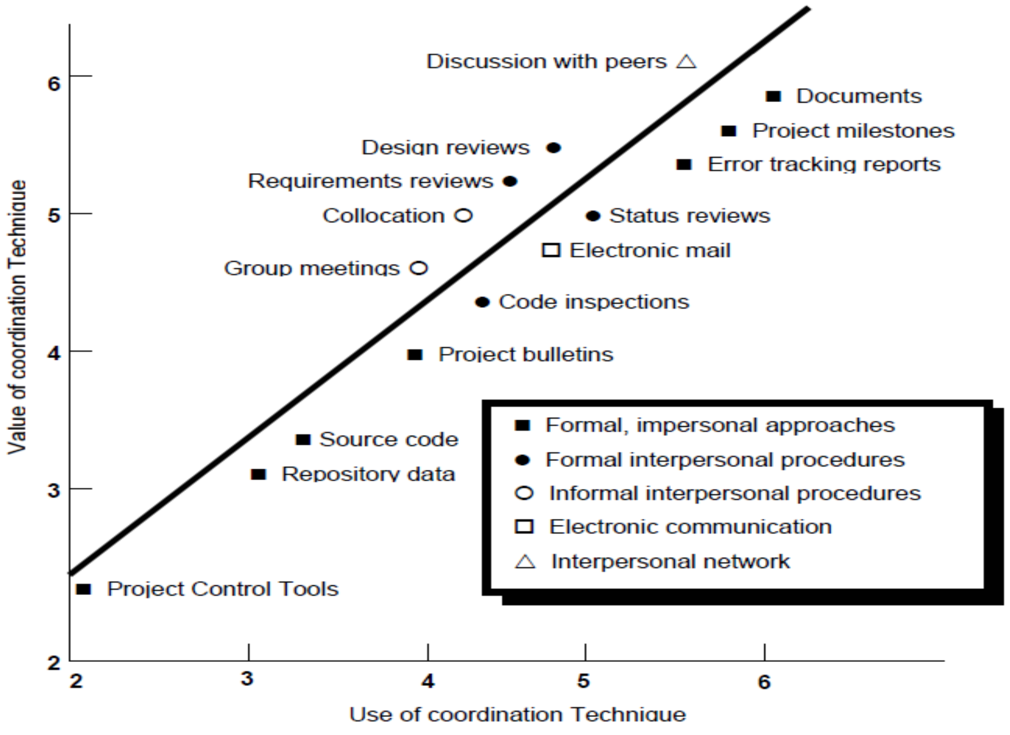


Figure 3.1 Value and use of coordination and communication techniques

Function and performance: What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software project scope must be unambiguous and understandable at management and technical levels. A statement of software scope must be bonded. That is, quantitative data (e.g., number of simultaneous users, size of mailing list, maximum allowable response time) are stated explicitly; constraints and/or limitations (e.g. product cost restricts memory size) are noted, and mitigating factors are described.

3.3.2 Problem Decomposition

Problem decomposition, sometimes called partitioning, is an activity that sits at core of software requirements analysis (later chapters). During the scooping activity there is no attempt to fully decompose the problem. Rather decomposition is applied in two major areas: **1.** the functionality that must be delivered and **2.** the process that will be used to deliver it.

Human beings tend to apply a divide and conquer strategy when they are confronted with a complex problem. Stated simply, a complex problem is partitioned into smaller problems that are more manageable. This is the strategy that is applied as project planning begins. Software functions, described in the statement of scope, are evaluated and regained to provide more detail prior to the beginning of estimation. Because both cost and schedule estimated are functionally oriented, some degree of decomposition is often useful.

3.3.3 The Process

The generic phases that characterize the software process definition, development, and maintenance are applicable to all software. The problem is to select the process model that is appropriate for the software to be engineered by a project team.

- The linear sequential model
- The prototyping model
- The RAD model
- The incremental model
- The spiral model
- The component development model
- The formal methods model
- The fourth generation techniques model

The project manager must decide which process model is most appropriate for the project and then define a preliminary plan based on the set of common process frame work activities. Once the preliminary plan is established, process decomposition begins. That is, a complete plan reflecting the work tasks required to populate the framework activities must be created. We explore these activities briefly in the sections that follow and present a more detailed view in later chapters.

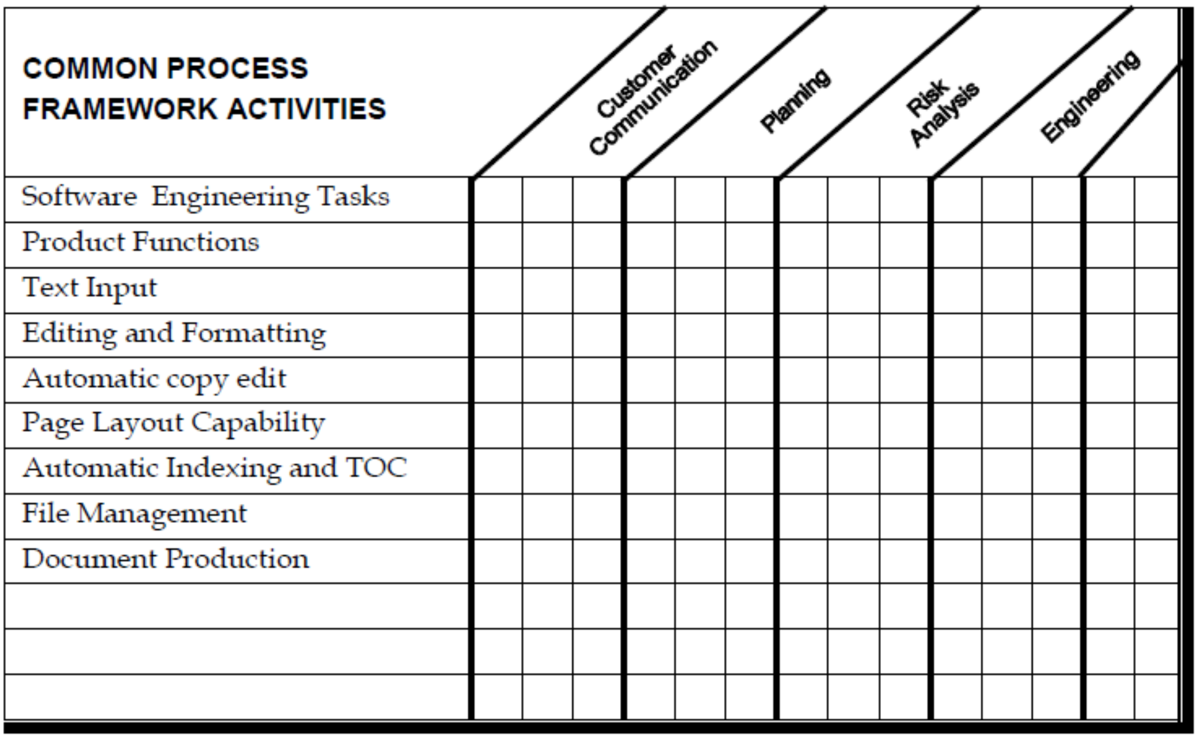
3.3.4 Melding the Problem and the Process

Project planning begins with the melding of the problem and the process. Each function to be engineered by the software team must pass through the set of framework activities that have been defined for a software organization. Assume that the organization has adopted the following set of framework activities.

- Customer communication – tasks required to establish effective communication between developer and customer.
- Planning – tasks required to define resources, timelines, and other project related information.
- Risk analysis – tasks required to assess both technical and management risk.
- Engineering -tasks required to build one or more representations of the application.
- Construction and release - tasks required to construct, test, install, and provide user support.

• Customer evaluation – tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

The team members who work on each function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in figure 3.2 is created. Each major problem function is listed in the left hand column. Framework activities are listed in the top row. Software engineering work tasks would be entered in the following row. The job of the project manager is to estimate resource requirements for each matrix, cell start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each cell, these issues are considered in later chapters.



COMMON PROCESS FRAMEWORK ACTIVITIES	Customer Communication	Planning	Risk Analysis	Engineering
Software Engineering Tasks				
Product Functions				
Text Input				
Editing and Formatting				
Automatic copy edit				
Page Layout Capability				
Automatic Indexing and TOC				
File Management				
Document Production				

Figure 3.2 Process Decomposition

3.3.5 Process Decomposition

A software team should have a significant degree of flexibility in choosing the software engineering paradigm that is best for the project and the software engineering tasks that populate the process model once it is chosen. A relatively small project that is similar to past efforts might be best accomplished using the linear sequential approach. If very tight time constraints are imposed and the problem can be heavily compartmentalized, the RAD model is probably the right option. If the deadline is so tight that full functionality cannot reasonably be delivered, an incremental strategy might be best. Similarly, projects with other characteristics will lead to the selection of other process models.

Once the process model has been chosen, the common process framework is adapted to it. In every case, the CPF discussed earlier in this chapter customer communication, planning, risk analysis, engineering, construction and release, customer evaluation can be fitted to the paradigm. It will work for linear models, for iterative and incremental models, for evolution models, and even for concurrent or component assembly models.

The CPF is invariant and serves as the basis for all software work performed by a software organization. But actual work tasks do vary. Process decomposition commences when the project manager asks: “how do we accomplish the CPF activity”? For example, a small, relatively simple project might require the following work tasks for the customer communication activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.
3. Jointly develop a statement of scope
4. Review the state of scope with all concerned
5. Modify the statement of scope as required.

These events might occur over a period of less than 48 hours. They represent a process decomposition that is appropriate for the small, relatively simple project. Now, we consider a more complete project that has a broader scope and more significant business impact. Such a project might require the following work tasks for the customer communication activity.

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
3. Conduct research to define proposed solutions and existing approaches.
4. Prepare a “working document” and an agenda for the formal meeting
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the

   software.
7. Review each mini spec for correctness, consistency, and lack of ambiguity.
8. Assemble the mini specs into a scooping document.
9. Review the scooping document with all concerned.
10. Modify the scooping document as required.

Both projects perform the frame work activity that we call customer communication, but the first project team performs half as many software engineering work tasks as the second.

4.0 The Project

Jaded industry professionals often refer to 90-90 rule when discussing particularly difficult software projects: the first 90 percent of a system absorbs 90 percent of the allotted effort and time. The last 10 percent takes the other 90 percent of the allotted effort and time. This statement tells us much about the state of a project that gets into trouble:

The manner in which progress is assessed is flawed. (Obviously, if the 90-90 rule is true, 90 percent complete is not an accurate indicator).

There is no way to calibrate progress because quantitative metrics are unavailable. The project plan has not been designed to accommodate resources required at the end of a project.

Risks have not been considered explicitly, and a plan for mitigating, monitoring, and managing them has not been created. The schedule is (1) unrealistic or (2) flawed.

To overcome these problems, time must be spent at the beginning of a project to establish a realistic plan, during the project to monitor the plan, and throughout the project to control quality and change.

Self-Assessment Exercise(s)

1. What is Software Process Model?
2. State and explain the focus of software project management?

Self-Assessment Answer(s)

1. What is Software Process Model?

The software process model maybe defined as a simplified description of a software process, presented from a particular perspective. In essence, each stage of the software process is identified and a model is then employed to represent the inherent activities associated within that stage. Consequently, a collection of 'local' models may be utilised in generating the global picture representative of the software process.

2. State and explain the focus of software project management?

The People

The following categories of people are involved in the software process.

- Senior Managers
- Project Managers
- Practitioners
- Customers
- End Users

Senior Managers define the business issue. Project Managers plan, motivate, Organize and control the practitioners who do the Software work. Practitioners deliver the technical skills that are necessary to engineer a product or application. Customer specifies the requirements for the software to be developed. End Users interact with the software once it is released.

The Product

Before a software project is planned, the product objectives and scope should be established, technical and management constraints should be identified. Without this information it is impossible to define a reasonable cost, amount of risk involved, the project schedule etc.

A software project scope must be unambiguous and understandable at the management and technical levels.

To develop a reasonable project plan we have to functionally decompose the problem to be solved.

The Process
Here the important thing is to select an appropriate process model to develop the software. There are different process models available. They are Water fall model, Iterative water fall model, Prototyping model, Evolutionary model, AD (Rapid Application Development) model, Spiral model. In practice we may use any one of the above models or a combination of the above models.

The Project
In order to manage a successful software project, we must understand what can go wrong (so that problems can be Avoided) and how to do it right. A project is a series of steps where we need to make accurate decision so as to make a successful project

4.0 Summary/Conclusion

i. Software project management is an umbrella activity within software techniques and development. It begins before any technical activity is initiated and continues throughout the definition, development, and maintenance of computer software.

ii. There P's have a substantial influence on software project management people, problem and process.

iii. People must be organized into effective teams, motivated to do high quality software work, and coordinated to achieve effective communication.

iv. The problem must be communicated from customer to developer, partitioned into its constituent arts, and positioned for work by the software team.

v. The process must be adapted to the people and the problem. A common process framework is selected, an appropriate software engineering paradigm is applied, and a set of work tasks is chosen to get the job done.

vi. The pivotal element in all software projects is people, software engineers can be organized in a number of different team structures that range from traditional control hierarchies to "open paradigm" teams. A variety of coordination and communication techniques can be applied to support the work of the team. In general, formal reviews and informal person to person communication have the most value for practitioners.

5.0 Tutor-Marked Assignments

1. What is Management Spectrum?
2. Explain briefly about Co-ordination and communication Issues?
3. Define Software Scope?
4. Discuss briefly about Problem Decomposition?
5. Write a Note on Project Decomposition?

References/Further readings

[AIR99] Airlie Council, “Performance Based Management: The Program Manager’s Guide Based on the 16-Point Plan and Related Metrics,” Draft Report, March 8, 1999.

[BAK72] Baker, F.T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal,* vol. 11, no. 1, 1972, pp. 56–73.

[BOE96] Boehm, B., “Anchoring the Software Process,” *IEEE Software,* vol. 13, no. 4, July 1996, pp. 73–82.[CON93] Constantine, L., “Work Organization: Paradigms for Project Management and Organization, *CACM,* vol. 36, no. 10, October 1993, pp. 34–43.

Unit 2

The Project Management Concept Software Project Planning

Contents

1.0 Introduction

The software project management process begins with a set of activities that are collectively called **project planning**. The first of these activities is **estimation**. Whenever estimates are made, we look into the future and accept some degree of uncertainty as a matter of course.

Although estimating is as much art as it is science, this important activity need not be conducted in a haphazard manner. Useful techniques for time and effort estimation do exist. And because estimation lays a foundation for all other project planning activities, and project planning provides the road map for successful software engineering, we would be ill advised to embark without it.

2.0 Learning Outcome:

In this unit the following will be learnt:

- About Software Scope
- About Software Resources
- Software Project Estimations

3.0 Learnt Content

3.1 Observations on Estimating

A leading executive was once asked what single characteristic was most important when selecting a project manager. His response “a person with the ability to know what will go wrong before it actually does”. We might add “and the courage to estimate when the future is cloudy.”

Estimation of resources, cost, and schedule for software development of fort requires experience, access to good historical information, and the courage to commit to quantitative measures when qualitative data are all that exist. Estimation carries inherent risk and it is this risk that leads to uncertainty.

Project complexity has a strong effect on uncertainty that is inherent in planning. Complexity, however, is relative measure that is affected by familiarity with past effort. A real time application might be perceived as “exceedingly complex” to a software group that has previously developed only batch applications.

The same real time application might be perceived as “run-of-the–mill” for a software group that has been heavily involved in high speed process control. A number of quantitative software complexity measures have been proposed. Such measures are applied at the design or code level and are therefore difficult to use during software planning (before a design and code exist). However other, more subjective assessments of complexity can be established early in the planning process.

Project size is another important factor that can affect the accuracy of estimates. As size increase, the interdependency among various elements of the software grows rapidly. Problem decomposition, an important approach to estimating, becomes more difficult because decomposed elements may still be formidable. To paraphrase Murphy’s law: What can go wrong will go wrong”-and if there are more things that can fail, more things will fail.

The degree of ***structural uncertainty*** also has an effect on estimation risk. In this context, structure refers to the degree to which requirements have been solidified, the ease with which function can be compartmentalized, and the hierarchical nature of information that must be processed.

The availability of historical information also determines estimation risk. Santayana once said, “Those who cannot remember that past is condemned to repeat it.” By looking back, we can emulate things that worked and avoid areas where problem arose. When comprehensive software metrics are available for past projects, estimates can be made with greater assurance; schedules can be established to avoid past difficulties, and overall risk is reduced.

Risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and risk become dangerously high. The software planner should demand completeness of function, performance, and interface definitions (contained in a system specification). The planner, and more important the customer, should recognize that variability in software requirements means instability in cost and schedule.

A project manager should not become obsessive about estimation. Modern software engineering approaches (e.g., evolutionary process models) take an iterative view of development. In such approaches, it is possible to revisit the estimate (as more information is known) and revise it when the customer makes changes to requirements.

3.2 Project Planning Objectives

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimate should attempt to define “best case” and “worst case” scenarios so that project outcomes can be bounded.

The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

3.3 Software Scope

The first activity in software project planning is the determination of software scope. Function and performance allocated to software during system engineering should be assessed to establish a project scope that is unambiguous and understandable at management and technical levels.

Software scope describes function, performance, constraints, interfaces, and reliability. Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. *Performance* considerations encompass processing and response time requirements. *Constraints* identify limits placed on the software by external hardware, available memory or other existing systems.

3.4 Obtaining Information Necessary for Scope

Things are always somewhat hazy at the beginning of a software project. A need has been defined and basic goals and objectives have been enunciated, but the information necessary to define scope (a prerequisite for estimation) has not yet been defined.

The most commonly used technique to bridge the communication gap between the customer and developer and to get the communication process started is to conduct a preliminary meeting or interview.

The first meeting between a software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask both are worried that what they do say will be misinterpreted; both are worried that what they do say will be misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here) both want to get the thing over with but at the same time, both want it to be a success.

Yet communication must be initiated. Gause and Weinberg suggest that the analyst start by asking *context free questions*. That is, set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself. The first set of context free questions focus on the customer, the overall goals, and the benefits. For example, the analyst might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution?

The next set of questions enable the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution.

- How would you [the customer] characterize “good” output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Are there special performance issues or constraints that will affect the way the solution is approached?

The final set of questions focus on the effectiveness of the meeting. Gause and Weinberg call these “meta questions” and propose the following (abbreviated) list:

- Are you the right person to answer these questions? Are your answers “official”?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Is there anyone else who can provide additional information?

- Is there anything else that I should be asking you?

These questions (and others) will help to "break the ice" and initiate the communication that is essential to establish the scope of the project. But a question and answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then be replaced by a meeting format that combines elements of problem solving, negotiation and specification.

A number of independent investigators have developed a team oriented approach to requirements gathering that can be applied to help establish the scope of a project. Called *facilitated application specification techniques* (FAST), this approach encourages the creation of a joint team of customer and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of requirements.

3.4.1 Scoping Example

Communication with the customer leads to a definition of the data, functions, and behavior that must be implemented, the performance and constraints that bound the system and related information. As an example, consider software that must be developed to drive a ***conveyor line sorting system*****.** The statement of scope for the CLSS follows.

The conveyor line sorting system (CLSS) sorts boxes moving along a conveyor line. Each box is identified by a bar code that contains a part number and is sorted into one of six bins at the end of the line. The boxes pass by a sorting station that contains a bar code reader and a PC. The sorting station PC is connected to a shunting mechanism that sorts the boxes into the bins. Boxes pass in random order and are evenly spaced. The line is moving at five feet per minute. A CLSS is depicted schematically in Figure 4.1.

CLSS software receives input information from a bar code reader at time intervals that conform to the conveyor line speed. Bar code data will be decoded into box identification format. The software will do a look-up in a part number data base containing a maximum of 1000 entries to determine proper bin location for the box currently at the reader (sorting station). The proper bin location is passed to a sorting shunt that will position boxes in the appropriate bin. A record of the bin destination for each box will be maintained for later recovery and reporting. CLSS software will also receive input from a pulse tachometer that will be used to synchronize the control signal to the shunting mechanism. Based on the number of pulses that will be generated between the sorting station and the shunt, the software will produce a control signal to the shunt to properly position the box.

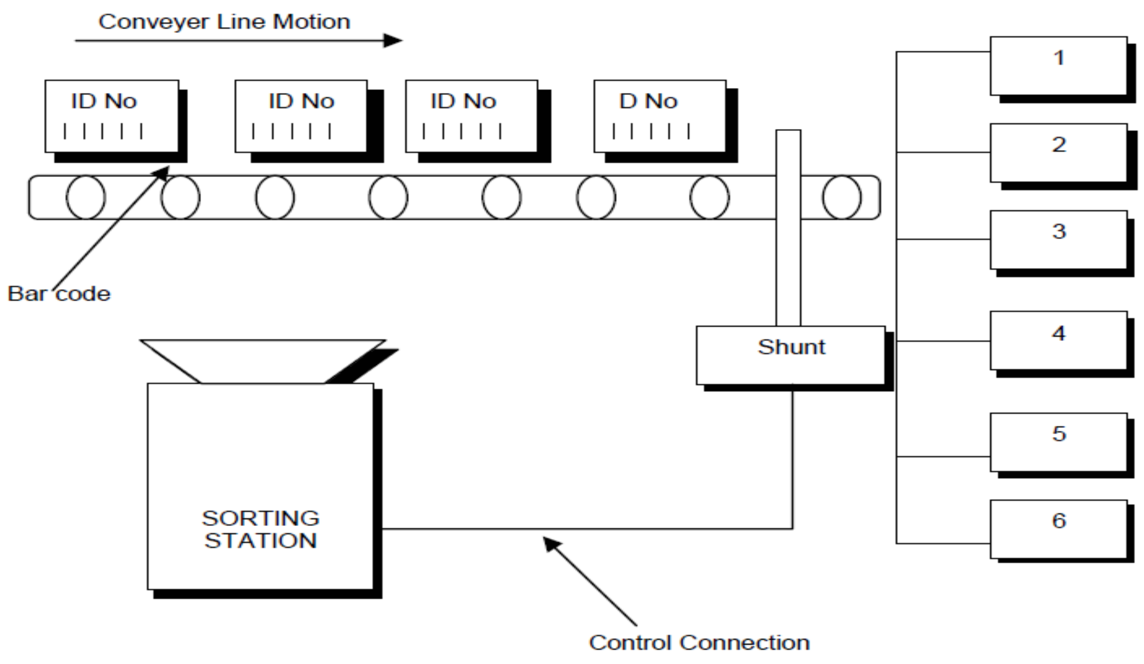


Figure 4.1 Conveyer ling sorting system

The project planner examines the statement of scope and extracts all important software functions. This process, called ***decomposition***, was discussed in the previous lecture and results in the following functions.

• Read bar code input

• Read pulse tachometer

• Decode part code data

• Do database look-up

• Determine bin location

• Produce control signal for shunt

• Maintain record of box destinations'

In this case, performance is dictated by conveyor line speed. Processing for each box must be completed before the next box arrives at the bar code reader. The CLSS software is constrained by the hardware it must access (the bar code reader, the shunt, the PC), the available memory, and the overall conveyor line configuration (evenly spaced boxes).

Function performance, and constraints must be evaluated together. The same function can precipitate and order of magnitude difference in development effort when considered in the context of different performance bounds. The effort and cost required to develop CLSS software would be dramatically it function remains the same but performance varies. For instance, if conveyor line average speed increase by a factor of 10 (performance) and boxes are no longer spaced evenly (a constraint) software would become considerably more complex and thereby require more effort. Function, performance, and constraint are intimately connected.

Software interacts with other elements of a computer based system. The planner considers the nature and complexity of each interface to determine any effect on development respires, cost, and schedule. The concept of an interface is interpreted to mean (1) hardware (e.g. machines, displays) that are indirectly controlled by the software (2) software that already exists (e.g. database access routines, reusable software components, operating system) and must be linked to the new software; (3) people who make use of the software via keyboard or other I/O devices; and (4) procedures that precede or succeed the software as a sequential series of operations. In each case the information transfer across the interface must be clearly understood.

The least precise aspect of software scope is a discussion of reliability. Software reliability measures do exist but they are rarely used at this stage of a project. Classic hardware reliability characteristics like meantime-between failure (MTBF) can be difficult to translate to the software domain. However, the general nature of the software may dictate special considerations to ensure “reliability”.

For example, software for a traffic control system or the Space Shuttle (both human-rated systems) must not fail or human life may be lost. An inventory control system or word-processing software should not fail, but the impact of failure is considerably less dramatic. Although it may not be possible to quantify software reliability as precisely as we would like in the statement of scope, we can use the nature of the project to aid in formulating estimates of effort and cost to assure reliability.

If a system specification has been properly developed, nearly all information required for a description of software scope is available and documented before software project planning begins. In cases where a specification has not been developed, the planner must take on the role of system analyst to determine attributes and bounds that will influence estimation tasks.

3.5 Resources

The second task of software planning is estimation of resources required to accomplish the software development effort. Figure 4.2 illustrates development resources as a pyramid. The ***development environment-***hardware and software tools-sits at the foundation of the resources pyramid and provide the infrastructure to support the development effort. At a higher level we encounter ***reusable software components*** software building blocks that can dramatically reduce development costs and accelerate delivery. At the top of the pyramid is the primary resource-people. Each resources are specified with four characteristics; description of the resources, a statement of availability, chronological time that the resource will be required, and duration of time that the resource will be applied.

The last two characteristics can be viewed as a *time window*. Availability of the resource of a specified window must be established at the earliest practical time.

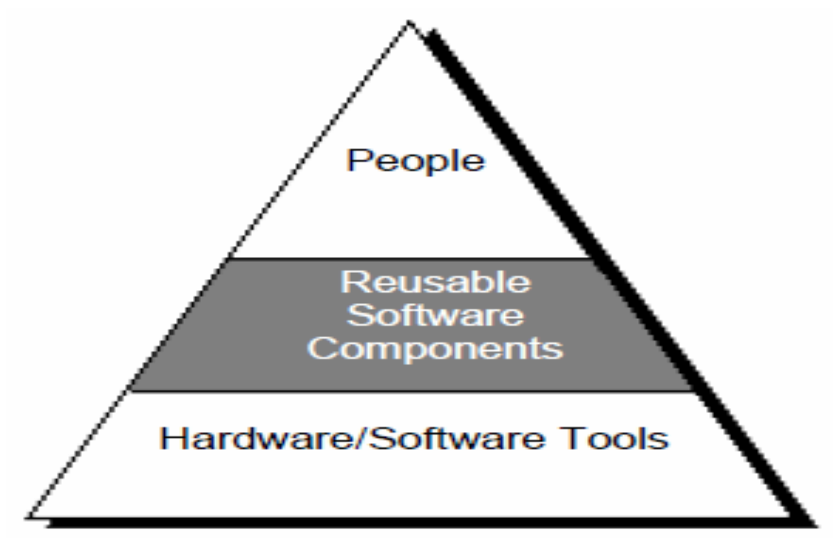


Figure 4.2 Resources

3.6 Human Resources

The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g. manager, senior software engineer, etc) and specialty (e.g. telecommunications, database, client/server) are specified. For relatively small projects (six people per month or less) a single individual may perform all software engineering steps, consulting with specialists as required.

The number of people required for a software project can be determined only after an estimate of development effort (e.g. person-months or person years) is made. Techniques for estimating effort are discussed later in the chapter.

3.7 Reusable Software Resources

Any discussion of the software resource would be incomplete without recognition of *reusability* that is, the creation and reuse of software building blocks [HOO91]. Such building blocks must be catalogued for easy reference, standardized for easy application, and validated for easy integration.

Bennatan suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components - Existing software that can be acquired from a third party or that has been developed internally for a past project. These components are ready for use on the current project and have been fully validated.

Full-experience components - Existing specifications, designs, code, or test date developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full experience components will be relatively low-risk.

Partial-experience components - Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project, but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications, required for partial experience components have a fair degree of risk.

New Components Software components that must be built by the software team specifically for the needs of the current project.

The following guidelines should be considered by the software planner when reusable components are specified as a resource.

(1). If off-the-shelf components meet project requirements, acquire them. The cost for acquisition and integration of off-the-shelf components will almost always be less than the cost to develop equivalent software. In addition, risk is relatively low.

(2). If full experience components are available the risks associated with modification and integration are generally acceptable. The project plan should reflect the use of these components.

(3). If partial-experience components are available, their use for the current project must be analyzed in detail. If extensive modification is required before the components can be properly integrated with other elements of the software, proceed carefully. The cost to modify partial experience components can sometimes be greater than the cost to develop new components.

Ironically, the use of reusable software components is often neglected during planning, only to become a paramount concern during the development phase of the software process. It is far better to specify software resource requirements early. In this way technical evaluation of alternatives can be conducted and timely acquisition can occur.

3.8 Environmental Resources

The environment that supports the software project, often called a software engineering environment, incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.

When a computer based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements begin developed by other engineering teams.

For example, software for a numerical control (NC) used on a class of machine tools may require a specific machine tool (e.g. a NC lathe) as part of the validation test step; a software project for automated typesetting may need a photo-typesetter at some point during development. Each hardware element must be specified by the software project planner.

Self-Assessment Exercise(s)

A. What is software project planning?

Self-Assessment Answer(s)

A. What is software project planning?

Software project planning, describes a series of actions or steps that are needed for the development of work product. In project organization, the functions of the personnel are integrated. It is done in parallel with project planning

4.0 Summary/Conclusion

- The software project planner must estimate three things before a project begins: how long it will take, how much effort will be required and how many people will be involved.
- The planner must predict the resources (hardware and software) that will be required and the risk involved.

5.0 Tutor-Marked Assignments

1. How do you obtain necessary information for scope?
2. Short Note on Resources?
3. Explain about Project Planning Objectives?
4. Describe briefly about Reusable Software Resource?
5. What is Environmental Resource?.

6.0 References/Further readings

[BRO95] Brooks, M., The Mythical Man-Month, Anniversary Edition, Addison Wesley, 1995.

[FLE98] Fleming, Q.W. and J.M. Koppelman, “Earned Value Project Management,” Crosstalk, vol. 11, no. 7, July 1998, p. 19.

[HUM95] Humphrey W., A Discipline for Software Engineering, Addison-Wesley, 1995.

[PAG85] Page-Jones, M., Practical Project Management, Dorset House, 1985, pp. 90–91.

Module 3

Unit 1

Application Programming Interface, Project Scheduling and Tracking

Contents

1.0 Introduction

You often have to rely on others to perform functions that you may not be able or permitted to do by yourself, such as opening a bank safety deposit box. Similarly, virtually all software has to request other software to do some things for it.

To accomplish this, the asking program uses a set of standardized requests, called application programming interfaces (API), that have been defined for the program being called upon. Although there are many reasons why software is delivered late, most can be traced to one or more of the following root causes.

- An unrealistic deadline established by someone outside the software engineering group and forced on managers and practitioners within the group.
- Changing customer requirements that are not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- A failure by project management to recognize that the project is falling behind schedule and a lack of action to correct the problem.

Aggressive deadlines are a fact of life in the software business. Sometimes such deadlines are demanded for reasons that are legitimate from the point of view of the person who sets the deadline, but common sense says that legitimacy must also be perceived by the people doing the work.

An **application programming interface** (API) is a source code interface that a computer system or program library provides to support requests for services to be made of it by a computer program. An API differs from an application binary interface in that it is specified in terms of a programming language that can be compiled when an application is built, rather than an explicit low level description of how data is laid out in memory.

The software that provides the functionality described by an **API** is said to be an *implementation* of the API. The API itself is abstract, in that it specifies an interface and does not get involved with implementation details. A good example of an API would be a web service interface, such as the API provided by Google.

Almost every application depends on the APIs of the underlying operating system to perform such basic functions as accessing the file system. In essence, a program's API defines the proper way for a developer to request services from that program. Developers can make requests by including calls in the code of their applications. The syntax is described in the documentation of the application being called. By providing a means for requesting program services, an API is said to grant access to or open an application.

Building an application with no APIs, says Josh Walker, an analyst at Forrester Research Inc. in Cambridge, Mass., "is basically like building a house with no doors. The API for all computing purposes is how you open the blinds and the doors and exchange information." APIs also exist between applications.

Middleware works by providing a standardized, API-like interface that can allow applications on different platforms or written in different languages to interoperate. Although APIs provide a quick and easy way to tap into an application, they can be constraining for certain power users such as independent software vendors, says Adam Braunstein, an analyst at Robert Frances Group Inc. in Westport, Conn.

Open source code exposes every instruction and operation in an application and therefore offers the most flexibility. But understanding source code can be time-consuming, and it also exposes the author's intellectual property.

When Novell Inc. was rumored to be considering opening up the source code for its Novell Directory Services (NDS) software last year, then-Vice President Chris Stone said most corporate developers didn't want to delve into open source code. Instead, he said, they wanted additional sets of APIs they could work with more quickly. So far, Novell has kept NDS code closed.

Corporate developers should consider including APIs in applications they develop, especially if they expect the applications to last and interact with other applications, Braunstein says. As time goes on, the likelihood that another developer will need to tap an application's services increases. He says having the foresight to include APIs saves subsequent developers from having to find and review the source code.

IMAGINE YOU HAVE THREE NEIGHBORS: Closed Carl, Open Oscar and API Annie. Each of you is like an application. Like any neighbor, you sometimes need to borrow things from your neighbors, such as a lawn mower. This is the equivalent of applications integrating.

CLOSED CARL simply won't provide you with any services. He mows his own lawn behind a high fence. Not only is there no way to ask him for anything, you can't even walk onto his property to try because his fence has no gate. An application like Closed Carl exposes no source code or APIs.

OPEN OSCAR is the opposite. He's so open that he will let you freely enter his yard whenever you'd like and even tinker with his mower so it suits your needs exactly. Of course, once you've changed the design from what's documented in the manual, you've entered the mower maintenance business. An application like Open Oscar has open source code, giving you free reign if you want it.

API ANNIE will let you borrow the mower if you ask in the right way (by calling the "getMower" API in your own application code). You can't enter the gate without that request, and you can't open the mower and tinker with it. But you can get the service of mowing as needed. Applications like Annie, which are closed but have APIs, are the most common in enterprise settings.

1.1 Comments on “Lateness”

The scheduling techniques described in this lecture, are often implemented under the constraint of a defined deadline. If best estimates indicate that the deadline is unrealistic, a competent project manager should “protect his or her team from undue pressure reflect the pressure back to its originators”.

To illustrate, assume that a software development group has been asked to build a real time controller for a medical diagnostic instrument that is to be introduced to the market in 9 months. After careful estimation and risk analysis the software project manager comes to the conclusion that the software as requested, will require 14 calendar months to create with available staff. How does the project manager proceed? It is unrealistic to march into the customer’s office and demand that the delivery date be changed.

External market pressures have dictated the date, and the product must be released. It is equally foolhardy to refuse to undertake the work. So, what to do?

The following steps are recommended in this situation:

1. Perform a detailed estimate using data from past projects. Determine the estimated effort and duration for the project.
2. Using an incremental process model develop a development strategy that will deliver critical functionality by the imposed deadline, but delay other functionality until later. Document the plan.
3. Meet with the customer and explain why the imposed deadline is unrealistic. Be certain to note that all estimates are based on performance on past projects. Also be certain to indicate the percentage improvement that would be required to achieve the deadline as it currently exists. The following comment is appropriate:
4. “I thing we may have a problem with the delivery date for the XYZ controller software. I’ve given each of you an abbreviated breakdown of production rates for past projects and an estimate that we’ve done a number of different ways. You’ll note that I’ve assumed a 20 percent improvement in past production rates, but we still get a delivery date that’s 14 calendar months rather than 9 months away”.

   Offer the incremental development strategy as an alternative. We have a few options, and I’d like you to make a decision based on them. First, we can increase the budget and bring on additional resources so that we’ll have a shot at getting this job done in nine months. But understand that this will increase risk of poor quality due to the tight time line. Second, we can remove a number of the software functions and capabilities that you’re requesting. This will make

> the preliminary version of the product somewhat less functionality and then deliver over the 14 month period. Third, we can dispense with reality and wish the project complete in 9 months. We'll wind up with nothing that can be delivered to a customer. The third option, I hope you will agree, is unacceptable. Past history and our best estimates say that it is unrealistic and a recipe for disaster".

There will be some grumbling but if solid estimates based on good historical data are presented, it's likely that negotiated versions of either option 1 or option 2 will be chosen. The unrealistic deadline evaporates.

1.2 Basic principles

The reality of a technical project is that hundreds of small tasks must occur to accomplish a larger goal. Some of these tasks lie outside the mainstream and may be completed without worry about impact on project completion date. Other tasks lie on the "Critical path." If these "critical" tasks fall behind schedule, the completion date of the entire project is put into jeopardy.

The project manager's objective is to define all project tasks, identify the ones that are critical, and then track their progress to ensure that delay is recognized "one day at a time". To accomplish this, the manager must have a schedule that has been defined at a degree of resolution that enables the manager to monitor progress and control the project.

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major software engineering activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software tasks are identified and scheduled.

Scheduling for software development projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer based system has already been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization. Effort is distributed to make best use of resources, and an end date is defined after careful analysis of the software. Unfortunately the first situation is encountered far more frequently than the second.

Like all other areas of software engineering, a number of basic principles guide software project scheduling:

Compartmentalization The project must compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

Interdependency The interdependencies of each compartmentalized activity or task must be determined. Some tasks must occur in sequence; others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time Allocation Each task to be scheduled must be allocated some number of work units. In addition, each task must assigned a start date and a completion date that are functions of the interdependencies and whether work will be conducted on a full time or part time basis.

Effort validation Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been allocated at any given time. For example, consider a project that has three assigned staff members. On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities: Every task that is scheduled should be assigned to a specific team member.

Defined outcomes: Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product or a part of a work product. Work products are often combined in deliverables.

Defined milestones: Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products have been reviewed for quality and have been approved.

Each of the above principles is applied as the project schedule evolves.

2.0 Learning Outcome

In this unit the following will be learnt:

a. Basic Concepts
b. About The Relationship between People and Effort
c. Defining a task set for the Software Project

3.0 Learning Content

3.1 The Relationship Between People and Effort

In a small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved.

There is a common myth that is still believed by many managers who are responsible for software development effort: “ If we fall the project”, Unfortunately, adding people late in a project often has a disruptive effect, causing schedules to slip even further. People late in a project often has disruptive effect, causing schedules to slip even further. People who are added must learn the system, and the people who teach them are the same people who were doing the work. While they are teaching, no work is done and the project falls further behind.

In addition to the time it takes to learn the system, involving more people increases the number of communication paths and the complexity of communication throughout a project. Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time.

3.1.1 An example

Consider four software engineers, each capable of producing 5000 LOC/year when working on an individual project. When these four engineers are placed on a team project, six potential communication paths are possible. Each communication path requires time that could otherwise be spent developing software. We shall assume that team productivity will associated with communication. Therefore, team productivity will be reduced by 250 LOC/year for each communication path, due to the overhead associated with communication. Therefore, team productivity is 20,000 – (250X6) = 18,500 LOC/year – 7.5 percent less than what we might expect.

The one year project on which the above team is working falls behind schedule and with two months remaining, two additional people are added to the team. The number of communication paths escalates to 14. The productivity input of the new staff is the equivalent of 840 X2 = 1680 LOC for the two months remaining before delivery. Team productivity now is 20,000 + 1680 – (250 X14) = 18,180 LOC / year.

Although the above example is a gross oversimplification of real world circumstances, it does serve to illustrate another key point; the relationship between the number of people working on a software project and overall productivity is not linear.

Based on the people – work relationship, are teams counterproductive? The answer is an emphatic "no," if communication serves to improve software quality. In fact, formal technical reviews conducted by software engineering teams can lead to better analysis and design, and more important, can reduce the number of errors that go undetected until testing. Hence, productivity and quality, when measured by time to project completion and customer satisfaction, can actually improve.

3.1.2 An empirical relationship

Recalling the "software equation "that was introduced in the previous lecture. We can demonstrate the highly nonlinear relationship between chronological time to complete a project and human effort applied to the project. The number of delivered lines of code L is related to effort and development time by the equation:

$$L = P \text{ X } (E/B)^{1/3} \, T^{4/3}$$

Where E is development effort in person months; P is a productivity parameter that reflects a variety of factors that lead to high quality software engineering work B is a special skill factor that ranges between 0.16 and 0.39 and is a function of the size of the software to be produced; and t is the project duration in calendar months. Rearranging the software equation (above), we arrive at an expression for development effort E.

$$D = L^3 / (P^3 T^4)$$

Where E is the effort expended over the entire life cycle for software development and maintenance, and T is the development time in years. The equation for development effort can be related to development cost by the inclusion of a burdened labor rate factor ($/person-year).

This leads to some interesting results. Consider a complex, real time software project estimated at 33,000 LOC, 12 person years of efforts. If eight people are assigned to the project team, the project can be completed in approximately 1.3 years. If, however, we extend the end date to 1.75 years the highly nonlinear nature of the model described in equation 7,.1 yields.

$$E= L^3/ (^{P3}T^4) \sim 3.8 \text{ person years.}$$

This implies that by extending the end date six months, we can reduce the number of people from eight to four! The validity of such results is open to debate, but the implication is clear; Benefit can be gained by using fewer people over a somewhat longer time span to accomplish the same objective.

3.1.3 Effort Distribution

Each of the software project estimation techniques discussed in previous lecture leads to estimates of person months required to complete software development. A recommended distribution of effort across the definition and development phases is often referred to as the 40-20-40 rule. Forty percent or more of all effort is allocated to front end analysis and design tasks. A similar percentage is applied to back end testing. You can correctly infer that coding is de-emphasized.

This effort distribution should be used as a guideline only. The characteristics of each project dictate the distribution of effort. Effort expended on project planning rarely accounts for more than 2 or 3 percent of effort, unless the plan commits and organization to large expenditures with high risk. Requirements analysis may comprise 10 to 25 percent of project effort. Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity.

A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved.

Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages may be considered.

3.2 Defining a Task Set for the Software Project

A number of different process models were described in lecture2. These models offer different paradigms for software development. Regardless of whether a software team chooses a linear sequential paradigm an iterative paradigm an evolutionary paradigm a concurrent paradigm or some permutation, the process model is populated by a set of tasks that enable the software team to define, develop, and ultimately maintain computer software.

There is no single set of tasks that is appropriate for all projects. The set of tasks that would be appropriate for a large complex system would likely be perceived as overkill for a small relatively simple project. Therefore an effective software process should define a collection of task sets each designed to meet the needs of different types of projects.

A task set is a collection of software engineering work tasks, milestone and deliverables that must be accomplished to complete a particular project. The task set to be chosen must provide enough discipline to achieve high software quality. But at the same time, it must not burden the project team with unnecessary work.

Task sets are designed to accommodate different type of projects and different degrees of rigor. Although it is difficult to develop a comprehensive taxonomy, most software organizations encounter projects of the following types:

Concept Development Projects - That are initiated to explore some new business concept or application of some new technology.

New Application Development Projects - That are undertaken as a consequence of a specific customer request

Application Enhancement Projects - That occur when existing software undergoes major modification to function performance or interfaces that are observable by the end user.

Application Maintenance Projects - That correct adapt or extend, existing software in ways that may not be immediately obvious to the end user.

Reengineering Projects - That are undertaken with the intent of rebuilding an existing (legacy) system in whole or in part.

Even within a single project type there are many factors that influence the task set to be chosen. When taken in combination these factors provide an indication of the degree of rigor with which the software process should be applied.

3.2.1 Degree of Rigor

Even for a project of a particular type the degree of rigor with which the software process is applied may vary significantly. The degree of rigor is function of many project characteristics. As an example, small, non-business, critical projects can generally be addressed with somewhat less rigor than large complex baseline critical applications. It should be noted however that all projects must be conducted in manner that results in timely high quality deliverables.

Four different degrees of rigor can be defined:

Casual - All process frame work activities are applied but only a minimum task set is requiring ed. In general umbrella tasks will be minimized and documentation requirements will be reduced. All basic principles of software engineering are still applicable.

Structured - The process framework will be applied for this project. Framework activities and related tasks appropriate to the project type will be applied and umbrella activities necessary to ensure high quality will be applied. SQA, SCM documentation and measurement tasks will be conducted in a streamlined manner.

Strict - The full process will be applied for this project with a degree of discipline that will ensure high quality. All umbrella activities will be applied and robust documentation will be produced.

Quick Reaction - The process framework will be applied for this project but because of an emergency situation only those tasks essential to maintaining good quality will be applied "Back-filling "(i.e., developing a complete set of documentation conducting additional reviews) will be accomplished after the application/product is delivered to the customer.

The project manager must develop a systematic approach for selecting the degree of rigor that is appropriate for a particular project. To accomplish this project adaptation criterion are defined and a task set selector value is computed.

3.2.2 Defining Adaptation Criteria

Adaptation criteria are used to determine the recommended degree of rigor with which the software process should be applied on a project. Eleven adaptation criteria are defined for software projects:

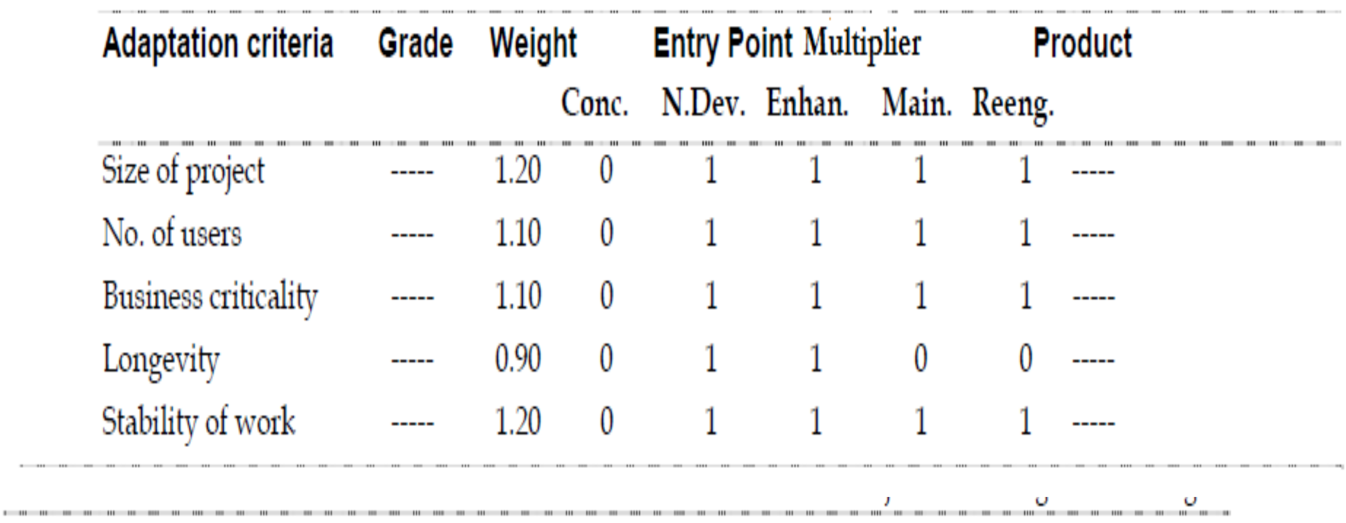
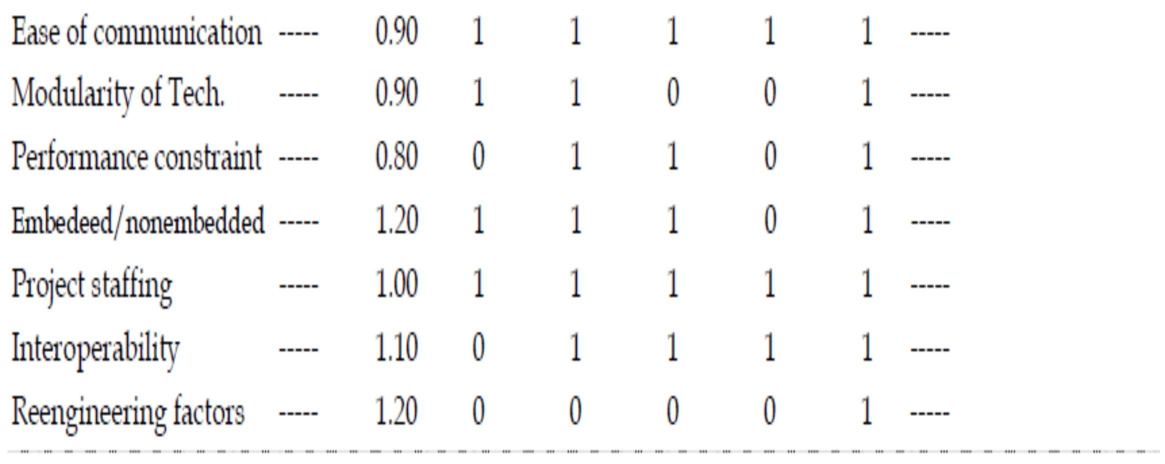
i. Size of the project
ii. Number of potential users
iii. Mission critically
iv. Application longevity
v. Stability of requirements
vi. Ease of customer/developer communication
vii. Maturity of applicable technology
viii. Performance constraints
ix. Embedded /non embedded characteristics
x. Project staffing
xi. Reengineering factors

Each of the adaptation criteria is assigned a grade that ranges between 1 and 5 where 1 represents a project in which a small subset of process tasks are required and over all methodological and documentation of requirements are minima l and 5 represents a project in which a complete set of process tasks should be applied and overall methodological and documentation requirements are substantial

3.2.3 Computing a Task Set Selector Value

1. To select the appropriate task set for a project the following steps should be conducted:
2. Review each of the adaptation criteria sin Section 5.3.2 and assign the appropriate grades (1 to 5) based on the characteristic of the project. These grades should be entered into Table 5.1.
3. Review the weighting factors assigned to each of the criteria. The value of a weighting factor ranges from 0.8 to 5.2 and provides an indication of the relative importance of a particular adaptation criterion to the types of software developed within the local environments. If modifications are required to better reflect local circumstances they should be made.
4. Multiply the grade entered in Table 5.1 by the weighting factor and by the entry point multiplier for the type of project to be undertaken. The entry point multiple user takes on a value of 0 or 1 and indicates that relevance of the adaptation criterion to the project type. The result of the product: ***Grade*weighting factor* entry point multiplier***.

5. Compute the average of all entries in the “Product” column and place the result in the space marked task set selector. This value will be used to help you select the task set that is most appropriate for the project.

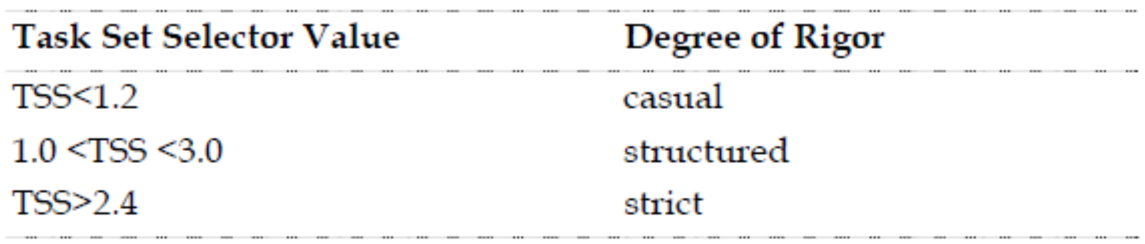



Adaptation criteria	Grade	Weight	Entry Point Multiplier					Product
			Conc.	N.Dev.	Enhan.	Main.	Reeng.	
Size of project	-----	1.20	0	1	1	1	1	-----
No. of users	-----	1.10	0	1	1	1	1	-----
Business criticality	-----	1.10	0	1	1	1	1	-----
Longevity	-----	0.90	0	1	1	0	0	-----
Stability of work	-----	1.20	0	1	1	1	1	-----
Ease of communication	-----	0.90	1	1	1	1	1	-----
Modularity of Tech.	-----	0.90	1	1	0	0	1	-----
Performance constraint	-----	0.80	0	1	1	0	1	-----
Embedeed/nonembedded	-----	1.20	1	1	1	0	1	-----
Project staffing	-----	1.00	1	1	1	1	1	-----
Interoperability	-----	1.10	0	1	1	1	1	-----
Reengineering factors	-----	1.20	0	0	0	0	1	-----

5.1 Computing the Task Set Selector

3.2.4 Interpreting the TSS Value and Selecting the Task Set

Once the task set selector is computed, the following guidelines can be used to select the appropriate task set for a project.



Task Set Selector Value	Degree of Rigor
TSS<1.2	casual
1.0 <TSS <3.0	structured
TSS>2.4	strict

The overlap in TSS values from one recommended task set to another is purposeful and is intended to illustrate that sharp boundaries are impossible to define when making ask set selections. In the final analysis the task set selector value past experience and common sense must all be factored into the choice of the task set for a project.

Adaptation criteria	Grade	Weight	Entry Point Moltiplier					Product
			Conc.	N.Dev.	Enhan.	Main.	Reeng.	
Size of project	2	1.2	-----	1	-----	-----	-----	2.4
No. of users	3	1.1	-----	1	-----	-----	-----	3.3
Business criticality	4	1.1	-----	1	-----	-----	-----	4.4
Longevity	3	0.9	-----	1	-----	-----	-----	2.7
Stability of work	2	1.2	-----	1	-----	-----	-----	2.4
Ease of communication	2	0.9	-----	1	-----	-----	-----	1.8
Modularity of Tech.	2	0.9	-----	1	-----	-----	-----	1.8
Performance constraint	3	0.8	-----	1	-----	-----	-----	2.4
Embedeed/nonembedded	3	1.2	-----	1	-----	-----	-----	3.6
Project staffing	2	1.0	-----	1	-----	-----	-----	2.0
Interoperability	4	1.1	-----	1	-----	-----	-----	4.4
Reengineering factors	0	1.2	-----	0	-----	-----	-----	2.8

Table 5.2 Computing the task set selector – An example

Table 5.2 illustrates how TSS might be computed for a hypothetical project. The project manager selects the grades shown in the "grade" column. The project type is new application development therefore entry point multipliers are selected from the **"N.DEV"** column. The entry in the "Product" column is computed using

Grade*Weight* New Dev Entry Point Multiplier

The value of TSS (computed as the average of all entries in the product column) is 2.8 Using the criteria discussed above, the manager has the option of using either the structured or the strict task set. The final decision is made once all project factors have been considered.

Self-Assessment Exercise(s)

1. What is Degree of Rigor?
2. How do you compute a TSS value?

Self-Assessment Answer(s)

4.0 Summary/Conclusion

- Scheduling is the culmination of a planning activity that is a primary component of software project management.
- When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

5.0 Tutor-Marked Assignments

1. What is Effort Distribution?
2. Define Empirical Relationship?

6.0 References/Further readings

[BEN92] Bennatan, E.M., Software Project Management: A Practitioner's Approach, McGraw-Hill, 1992.

[BOE81] Boehm B., Software Engineering Economics, Prentice-Hall, 1981.

[BOE89] Boehm B., Risk Management, IEEE Computer Society Press, 1989.

[BOE96] Boehm, B., "Anchoring the Software Process," IEEE Software, vol. 13, no. 4, July 1996, pp. 73–82.

Unit 2

Software Quality Assurance

Contents

1.0 Introduction

In this lecture, we focus on the Management issues and the process specific activities that enable a software organization ensure that it does the right thing at the right time in the right way. A quantitative discussion of quality is presented in the lecture Technical Metric for software.

2.0 Learning Outcome:

In this unit the following will be learnt:

a. About Quality Concepts
b. About Quality Movement
c. About Software Reviews

3.0 Learning Content

3.1 Quality Concept

3.1.1 Quality

The American Heritage Dictionary defines quality as “a characteristic or attribute of something. As an attribute of an item, quality refers to measurable characteristics-thing we are able to compare to known standards such as length, color, electrical properties, malleability, and so on. However, software, largely an intellectual entity, is more challenging to characterize than physical objects.

Nevertheless, measures of a program’s characteristics do exist. These properties include complexity, cohesion, number of function points, lines of code and many others discussed in the lecture coming. When we examine an item based on its measurable characteristics, two kinds of quality may be encountered quality of design and quality of conformance.

Quality of design refers to the characteristics that designers specify for an item. Grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-graded materials are used and tighter tolerances and greater levels of performance are specified, the design quality of a product increases, If the product is manufactured according to specifications.

Quality of conformance is the degree to which the design specifications are followed during manufacturing. Again the greater the degree of conformance, the higher the level of quality of conformance.

In software development quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.

3.1.2 Quality control

Variation control may be equated to quality control. But how do we achieve quality control? Quality control is the series of inspections reviews, and tests used throughout the development cycle to ensure that each work product meets the requirements placed upon it, Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications. This approach views quality control as part of the manufacturing process.

Quality control activities may be fully automated, entirely manual or a combination of automated tools and human interaction. A key concept of quality controls is that all work products have defined and measurable specifications. To which we may compare the outputs of each process. The feedback loop is essential to minimize the defects produced.

3.1.3 Quality Assurance

Quality assurance consists of the auditing and reporting functions of management. The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals. Of course, if the data provided through quality assurance identify problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

3.1.4 Cost of Quality

Cost of quality includes all costs incurred in the pursuit of quality or in performing quality related activities. Cost of quality studies are conducted to provide a baseline for the current. Cost of quality, to identify opportunities for reducing the cost of quality and; to provide a normalized basis of comparison.

The basis of normalization is almost always money. Once we have normalized quality costs on a currency basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the effect of changes in dollar-based terms. Quality costs may be divided into costs associated with prevention, appraisal, and failure. Prevention costs include:

i. Quality planning
ii. Formal technical reviews
iii. Test equipment
iv. Training

Appraisal costs include activities to gain insight into product condition "first time through" each process. Examples of appraisal costs include:

i. In-process and inter process inspection
ii. Equipment calibration and maintenance
iii. Testing

Failure costs are costs that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. Internal failure costs are the costs incurred when we detect an error in our product prior to shipment. Internal failure costs include:

i. Rework
ii. Repair
iii. Failure mode analysis

External failure costs are the costs associated with defects found after the product has been shipped to the customer. Examples of external failure s costs are:

i. Complaint resolution
ii. Product return and replacement
iii. Help line support
iv. Warranty work

As expected, the relative costs to find and repair a defect increase dramatically as we go from prevention to detection and from internal failure to external failure. Figure 6.1 based on data collected by Boehm, illustrates this phenomenon.

More recent anecdotal data is reported by Kaplan and his colleagues and is based on work at IBM's Rochester development facility:

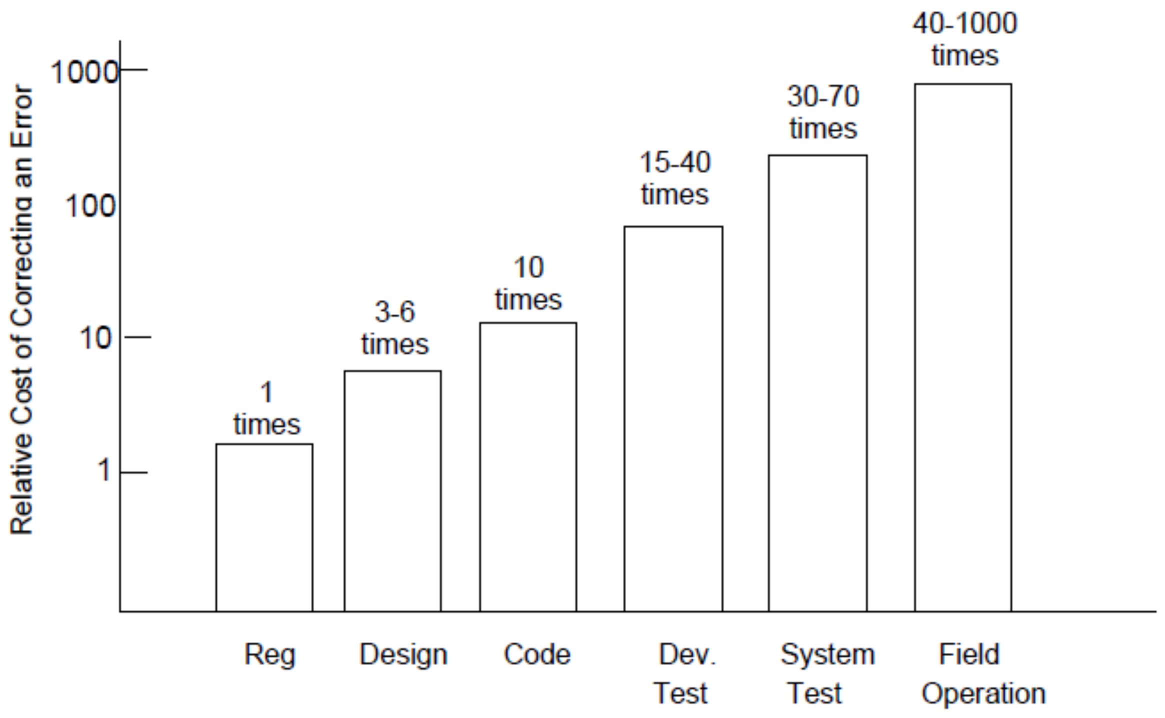


Figure 6.1 Relative cost of correcting an error

A total of 7053 hours was spent inspecting 2, 00,000 lines of code with the result that 3112 potential defects were prevented. Assuming a programmer cost of $40.00 per hour the total cost of preventing 3112 defects was $282,120 or roughly $91.00 per defect.

Compare these numbers to the cost of defect removal once the product has been shipped to the customer. Suppose that there had been no inspections, but that programmers had been extra careful and only one defect per 1000 lines of code [significantly better than industry average] escaped into the shipped product.

That would mean that 200 defects would still have to be fixed in the field. At an estimated cost of $25.000 per field fix, the cost would be $5 million or approximately 18 times more expensive than the total cost of the defect prevention effect.

It is true that IBM produces software that is used by tens of thousands of customers and that their costs for field fixes may be higher than average. This in no way negates the results noted above. Even if the average software organization has field fix costs that are 25 percent of IBM's (most have no idea what their costs are!), the cost savings associated with quality control and assurance activities are compelling.

3.2 The Quality Movement

Today senior managers at companies throughout the industrialized world recognize that high product quality translates to cost savings and an improved bottom line. However, this was not always the case. The quality movement began in the 1940's with the seminal work of W. Edwards Deming [DEM86] and had its first true test in Japan. Using Deming's ideas as a cornerstone, the Japanese have developed a systematic approach to the elimination of the root causes of product defects. Throughout the 1970s and 1980s their work migrated to the Western world and is sometimes called "total quality management (TQM)," Although terminology differs across different companies and authors, a basic four-step progression in normally encountered and forms the foundation of any good TQM program.

The first step is **called kaizen** and *refers to a system of continuous process improvement*. The goal of kaizen is to develop a process (in this case, the software process) that is visible, repeatable, and measurable.

The second step, invoked only after kaizen has been achieved, is **called atarimae hinshitsu** (*The idea that things will work as they are supposed to (e.g. a pen will write.). The functional requirement actually. Example a wall or flooring in a house has functional parts in the house as a product and when the functionality is met atarimae quality requirement is me*). This step examines intangibles that affect the process and works to optimize their impact on the process. For example, the software process may be affected by high staff turnover, which itself is caused by constant reorganizations within a company. It may be that a stable organizational structure could much to improve the quality of software. Atarimae hinshitsu would lead management to suggest changes in the way reorganization occurs.

While the first two steps focus on the process, the next step **called kansei** (translated as "the five senses') *concentrates on the user of the product* (in this case, software). In essence by examining the way the user applies the product kansei leads to improvement in the product itself, and potentially to the process that created it.

Finally, a step called **miryokuteki hinshitsu** broadens management concern beyond the immediate product. This is a business-oriented step that looks for opportunity in related areas that can be identified by observing the use of the product in the marketplace. In the software world, **miryokuteki hinshitsu** might be viewed as an attempt to uncover new and profitable products or applications that are an outgrowth from an existing computer-based system. For most companies kaizen should be of immediate concern. Until a mature software process has been achieved, there is little point in moving to the next steps.

3.3 Software Quality Assurance

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define quality? A wag once said, "Every program does something right, it just may not be the thing that we want it to do"

There have been many definitions of software quality proposed in the literature. For our purposes, software quality is defined as:

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

There is little question that the above definition could be modified or extended. In fact, a definitive definition of software quality could be debated endlessly. For the purposes of this book, the above definition serves to emphasize 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 (e.g., the desire for good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

3.3.1 Background issues

Quality assurance is an essential activity for any business that produces products to be used by others. Prior to the twentieth century, quality assurance was the sole responsibility of the craftsperson who built a product. The first formal quality assurance and control function was introduced at Bell Labs in 1916 and spread rapidly throughout the manufacturing world.

During the early days of computing (the 1950s), quality was the sole responsibility of the programmer. Standards for quality assurance for software were introduced in military contract software development during the 1970s and have spread rapidly into software development in the commercial world.

Extending the definition presented earlier, software quality assurance is a “planned and systematic pattern of actions” that are required to ensure quality in software. Today the implication is that many different constituencies in an organization have software quality assurance responsibility-software engineers, project manager’s customers, salespeople, and the individuals who serve within an

SQA group

The SQA group serves as the customer’s in-house representative, that is the people who perform SQA must look at the software from the customer’s point of view Does the software adequately meet, the quality factors. Has software development been conducted according to pre-established standards? Have technical disciplines properly performed their roles as part of the SQA group attempts to answer these and other questions to ensure that software quality is maintained.

SQA Activities

Software quality assurance is comprised of a variety of tasks associated with two different constituencies the software engineers who do technical work and a SQA group that has responsibility for quality assurance planning, over sight, record keeping, analysis and reporting.

Software engineers address quality (and perform quality assurance) by applying solid technical methods and measures, conducting formal technical reviews, and performing well-planned software testing.

The charter of the SQA group is to assist the software engineering team in achieving a high quality end product. The software Engineering Institute recommends a set of SQA activities that address quality assuring planning, oversight. Record keeping, analysis, and reporting, It is these activities that are performed (or facilitated) by an independent SQA group.

Prepare a SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties Quality assurance activities performed by the software engineering team and the SQA groups are governed by the plan. The plan identifies:

• Evaluations to be performed

• Audits and reviews to be performed

• Standards that is applicable to the project

• Procedures for error reporting and tracking

• Documents to be produced by the SQA group

• Amount of feedback provided to software project team

Participates in the development of the project's software process description. The software engineering team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO 9001), and other parts of the software project plan.

Review software engineering activities to verify compliance with the defined software process. The SQA group identifies documents and tracks deviations from the process and verifies that corrections have been made.

Audits designated software work products to verify compliance with those defined as part of the software process. The SQA group reviews selected work products, identifies, documents and tracks deviations, verifies that corrections have been made and periodically reports the results of its work to the project manager.

Ensures that deviations in software work and work products are documented and handled according to a documented procedure. Deviations may be encountered in the project plan, process description, applicable standards, or technical work products.

Record any noncompliance and reports to senior management. Noncompliance items are tracked until they are resolved.

In addition to these activities the SQA group coordinates the control and management of change and helps to collect and analyze software metrics.

3.4 Software Reviews

Software reviews are a "filter" for the software engineering process. That is reviews are applied at various points during Software development and serve to uncover errors that can then be removed. Software reviews serve to "purify" the software work products that occur as a result of analysis design, and coding.

Freedman and Weinberg discuss the need for reviews this way:

Technical work needs reviewing for the same reason that pencils need erasers: To err is human. The second reason we need technical reviews is that although people are good at catching some of their own errors, large classes of errors escape the originator more easily than they escape anyone else.

A review-any review-is a way of using the diversity of a group of people to:

1. Point out needed improvement in the product of single person or team.
2. Confirm those parts of a product in which improvement is either not desired or not needed; and
3. Achieve technical work of more uniform, or at least more predictable.

   Quality than can be achieved without reviews, in order to make technical work more manageable.

There are many different types of reviews the can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is a form of review. In this book however we focus on the formal technical review sometimes called a walkthrough.

A formal technical review is the most effective filter from a quality assurance standpoint conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

Cost Impact of Software Defects

The IEEE Standard Dictionary of Electrical and Electronics Terms (IEEE Standard 100-1992) defines a defect as “a product anomaly. “The definition for “fault “in the hardware context can be found in IEEE Standard 610.12-1990;

(a) A defect in a hardware device or component; for example, a short circuit or broken wire. (b) An incorrect step process, or data definition in a computer program. Note this definition is used primarily by the fault tolerance discipline. In common usage, the terms “error” and “bug” are used to express this meaning see also data-sensitive fault; program-sensitive fault; equivalent faults; fault masking; intermittent fault.

Within the context of the software process, the terms “defect” and “fault” are synonymous. Both imply a quality problem that is discovered after the software has been released to end users. In earlier chapters we used the term “error” to depict a quality problem that is discovered by software engineers (or others) before the software is released to the end user.

The primary objective of formal technical reviews is to find errors during the process so that they do become defects after release of the software. The obvious benefit of formal technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

A number of industry studies (TRW, Nippon Electric, and Mitre Corp., among others) indicate that design activities introduce between 50 and 65 percent of all errors (and ultimately, all defects) during the software process. However, formal review techniques have been shown to be up to 75 percent effective in uncovering design flaws. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and maintenance phases.

To illustrate the cost impact of an early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects Assume that an error uncovered during design will cost 1.0 monetary unit to correct. Relative to this cost, the same error uncovered just before testing commences will cost 6.5 units during testing 15 units and after release between 60 and 100 units.

Defect Amplification and Removal

A defect amplification model can be used to illustrate the generation and detection of errors during preliminary design, detail design, and coding steps of the software engineering process. The model is illustrated schematically in Figure 6.2. A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through. In some cases, errors, passed through from previous steps are amplified (amplification factor, x) by current work. The box subdivisions represent each of these characteristics and the percent efficiency for detecting errors a function of the thoroughness of review.

Figure 6.3 illustrates a hypothetical example of defect amplification for a software development process in which no reviews are conducted. As shown in the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors (an optimistic assumption). Ten preliminary design errors are amplified to 94 errors before testing commences. Twelve latent defects are released to the field Figure 6.4 considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, 10 initial preliminary design errors are amplified to 24 errors before testing commences. Only three latent defects exist. By recalling the relative costs associated with the discovery and correction of errors overall cost (with and without review for your hypothetical example) can be established. In Table 6.1 it can be seen that total cost for development and maintenance when reviews are conducted is 783 cost units. When no reviews are conducted total cost is 2177 units-nearly three times costlier.

To conduct reviews, a developer must expend time and effort and the development organization must spend money. However, the results of the preceding example leave little doubt that we have encountered a “pay now or pay much more later” syndrome. Formal technical reviews (for design and other technical activities) provide a demonstrable cost benefit. They should be conducted.

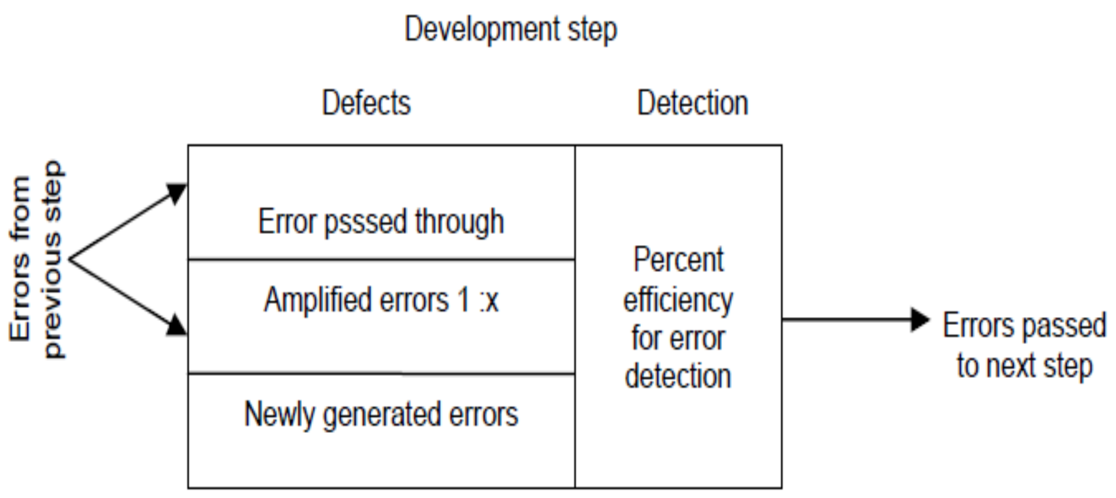


Figure6.2 Defect application model

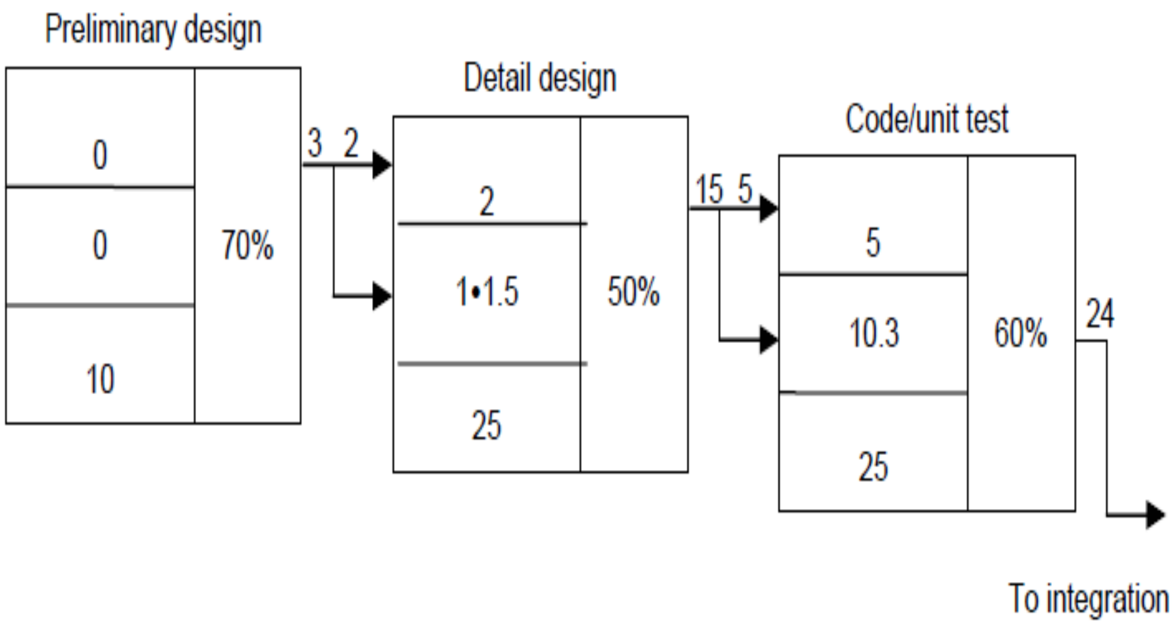


Fig6.3 Defect amplification no reviews

Self-Assessment Exercise(s)

What is 'Software Quality Assurance'?

What's the role of documentation in Quality Assurance?

Self-Assessment Answer(s)

A. What is 'Software Quality Assurance'?

Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon standards and procedures are followed, and ensuring that problems are found and dealt with. It is oriented to 'prevention'.

B. What's the role of documentation in Quality Assurance?

Generally, the larger the team/organization, the more useful it will be to stress documentation, in order to manage and communicate more efficiently. (Note that documentation may be electronic, not necessarily in printable form, and may be embedded in code comments, may be embodied in well-written test cases, user stories, etc.) QA practices may be documented to enhance their repeatability. Specifications, designs, business rules, configurations, code changes, test plans, test cases, bug reports, user manuals, etc. may be documented in some form. There would ideally be a system for easily finding and obtaining information and determining what documentation will have a particular piece of information. Change management for documentation can be used where appropriate. For agile software projects, it should be kept in mind that one of the agile values is "Working software over comprehensive documentation", which does not mean 'no' documentation. Agile projects tend to stress the short term view of project needs; documentation often becomes more important in a project's long-term context.

4.0 Summary/Conclusion

1. Software quality assurance is an “umbrella activity” that is applied at each step in the software process; SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms.
2. SQA is complicated by the complex nature of software quality—an attribute of computer programs that is defined as “conformance to explicitly and implicitly defined requirements.” But when considered more generally, software quality encompasses many different product and process factors and related metrics.

5.0 Tutor-Marked Assignments

1. Explain briefly about Quality Concept?
2. Give short note on Software Reviews?

6.0 References/Further readings

www. Wikipedia.com

[ALV64] Alvin, W.H. von (ed.), Reliability Engineering, Prentice-Hall, 1964.

[ANS87] ANSI/ASQC A3-1987, Quality Systems Terminology, 1987.

[ART92] Arthur, L.J., Improving Software Quality: An Insider's Guide to TQM, Wiley, 1992.

[ART97] Arthur, L.J., "Quantum Improvements in Software System Quality, CACM, vol. 40, no. 6, June 1997, pp. 47–52.

[BOE81] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

[CRO79] Crosby, P., Quality Is Free, McGraw-Hill, 1979.

[DEM86] Deming, W.E., Out of the Crisis, MIT Press, 1986.

[DEM99] DeMarco, T., "Management Can Make Quality (Im) possible," Cutter IT Summit, Boston, April 1999.

[DIJ76] Dijkstra, E., A Discipline of Programming, Prentice-Hall, 1976.

[DUN82] Dunn, R. and R. Ullman, Quality Assurance for Computer Software, McGraw Hill, 1982.

[FRE90] Freedman, D.P. and G.M. Weinberg, Handbook of Walkthroughs, Inspections and Technical Reviews, 3rd ed., Dorset House, 1990.

[GIL93] Gilb, T. and D. Graham, Software Inspections, Addison-Wesley, 1993.

[GLA98] Glass, R., "Defining Quality Intuitively," IEEE Software, May 1998, pp. 103–104, 107.

[HOY98] Hoyle, D., ISO 9000 Quality Systems Development Handbook: A Systems Engineering Approach, Butterworth-Heinemann, 1998.

Module 4

Unit 1: Software Configuration Management

Unit 2: Design Concepts and Principles

Unit 1

Software Configuration Management

Contents

1.0 Introduction

Software configuration management is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change (2) control change, (3) ensure that change is being properly implemented and (4) report change to others who may have an interest.

It is important to make a clear distinction between software maintenance and software configuration management. Maintenance is a set of software engineering activities that occur after software has been delivered to the customer and put into operation. Software configuration management is a set of tracking and control activities that begin when a software project begins and terminate only when the software is taken out of operation.

A primary goal of software engineering is to improve the ease with which changes can be accommodated and reduce the amount of effort expended when changes must be made. In this chapter we discuss the specific activates that enable us to manage change.

2.0 Learning Outcome:

In this unit the following will be learnt:

1. About SCM Process
2. About Version Control
3. About Change Control

3.0 Learning Content

3.1 Software Configuration Management.

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source-level and executable forms) (2) documents that describe the computer programs (targeted at both technical practitioners and users), and (3) data (contained within the program or external to it) The items that comprise all information produced as part of the software process are collectively called a software configuration.

As the software process progresses, the number of software configuration items (SCIs) grows rapidly. A system specification spawns a software project plan and software requirements specification (as well as hardware related documents). These in turn spawn other documents to create a hierarchy of information. If each SCI simply spawned other SCIs little confusion while result. Unfortunately, another variable enters the process—change. Change may occur at any time for any reason. In fact, the First Law of System Engineering states:

"No matter where you are in the system life cycle the system will change and the desire to change it will persist throughout the life cycle."

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of changes:

- New business or market conditions that dictate changes in product requirements or business rules.
- New customer needs that demand modification of data produced by information systems, functionality delivered by products or services delivered by a computer-based system.
- Reorganization and /or business downsizing that causes changes in project priorities of software engineering team structure.
- Budgetary or scheduling constraints management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we examine major SCM tasks and important concepts that help us to manage change.

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, we examine major SCM tasks and important concepts that help us to manage change.

Baselines

Change is a fact of life in software development. Customers want to modify requirements. Developers want to, modify technical approach. Managers want to modify project approach. Why all this modification? The answer is really quite simple. As time passes all constituencies know more (about what they need, which approach would be best and how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: Most changes are justified!

A baseline is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE defines a baseline as:

"A specification or product that has been formally reviewed and agreed upon that thereafter serves as the basis for further development and that can be changed only through formal change control procedures."

One way to describe a baseline is through analogy: Consider the doors to the kitchen of a large restaurant. To eliminate collisions one door is marked OUT and the other is marked IN. The doors have stops that allow them to be opened only in the appropriate direction. If a waiter picks up an order in the kitchen, places it on a tray and then realizes he has selected the wrong dish, he may change to the correct dish quickly and informally before he leaves the kitchen.

If, however he leaves the kitchen gives the customer the dish and then is informed of his error he must follow a set procedure: (1) Look at the check to determine if an error has occurred; (2) Apologize profusely; (3) Return to the kitchen through the indoor. (4) Explain the problem and so forth. A baseline is analogous to a dish as it passes through the kitchen door in the restaurant. Before a software configuration item becomes a baseline, change may be made quickly and informally. However once a baseline change may be made quickly and informally. However once a baseline is established we figuratively pass through a swinging one-way door. Changes can be made but a specific formal procedure must be applied to evaluate and verity each change.

In the context of software engineering a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review. For example, the elements of a design specification have been documented and reviewed. Errors are found and corrected. Once all parts of the specification have been reviewed corrected and then approved the design specification becomes a baseline. Further changes to the later, has each been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure11.1.

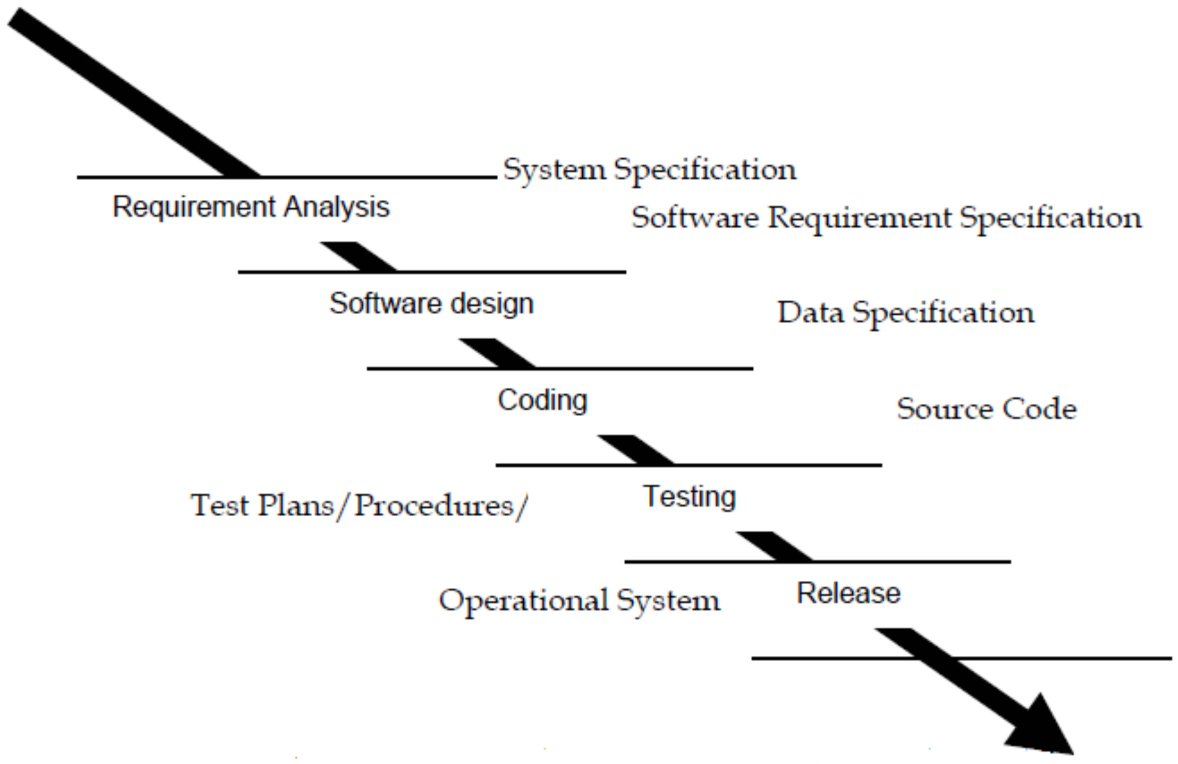


Figure 7.1 Baselines

The progressing of events that lead to a baseline is illustrated in **Figure7.2**. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a project database (also called a project library or software repository). When a member of a software engineering team wants to make a modification to a baseline SCI it is copied from the project database into the engineer's private work space. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The dashed arrows noted in **Figure7.2** illustrate the modification path for a baselined SCI.

Software Configuration Items

We have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme an SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically an SCI is a document an entire suite of test cases or a named program component (e.g., a C++ function or an Ada95 package).

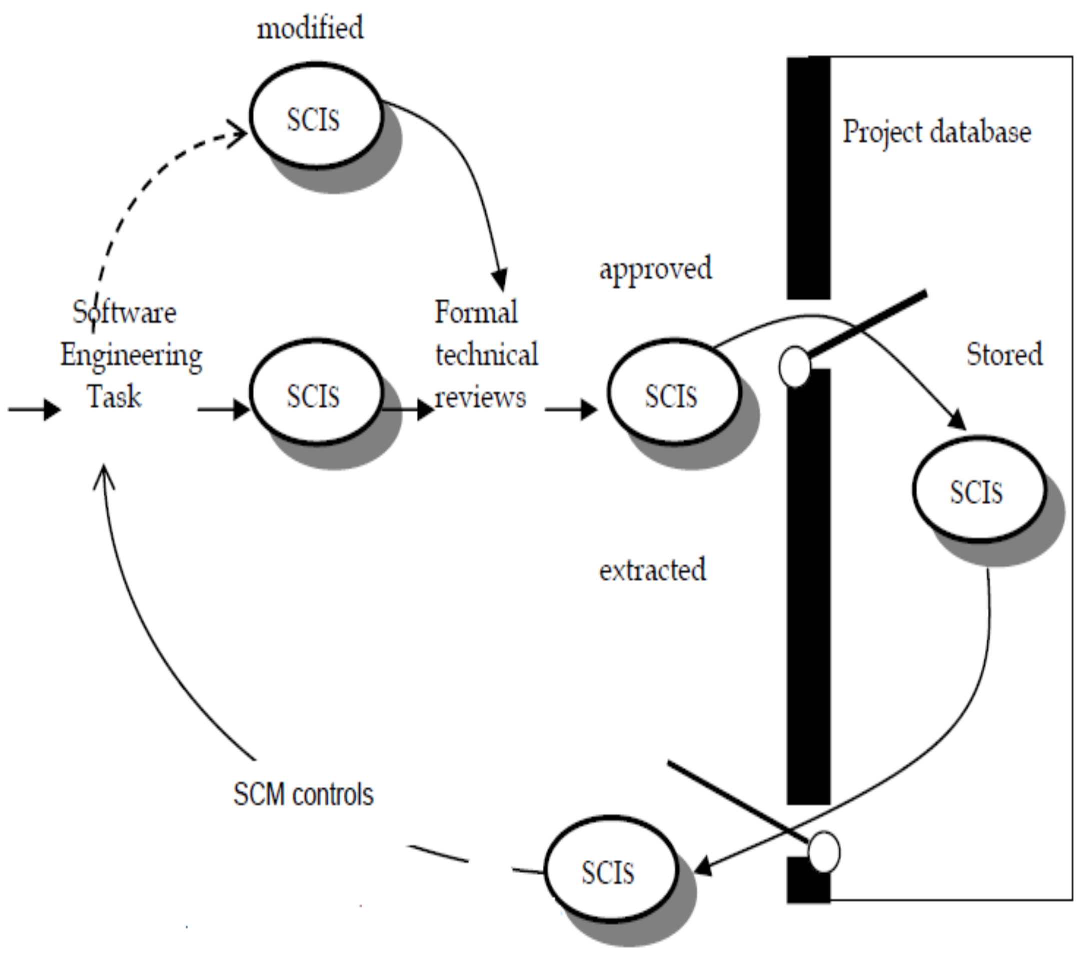


Figure 7.2 Baselined SCIs and the Project Database

The following SCIs become the target for configuration management techniques and form a set of baselines:

1. System Specification
2. Software Project Plan
3. Software Requirements Specification
   a. Graphical analysis models
   b. Process specifications
   c. Prototype

    d. Mathematical specification
4. Preliminary User Manual
5. Design Specification
    a. Data design description
    b. Architectural design description
    c. Module design descriptions
    d. Interface design descriptions
    e. Object descriptions (if object-oriented techniques are used)
6. Source Code Listing
7. Test Specification
    a. Test plan and procedure
    b. Test cases and recorded results
8. Operation and Installation Manuals
9. Executable Program
    a. Module executable code
    b. Linked modules
10. Database Description
    a. Schema and file structure
    b. Initial Content
11. As-built User Manual
12. Maintenance Documents
    a. Software problem reports
    b. Maintenance requests
    c. Engineering change orders
13. Standards and Procedures for Software Engineering

In addition to the SCIs noted above, many software engineering organizations also place software tools under configuration control. That is Specific versions of editors, compilers and other CASE tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselines as part of a comprehensive configuration management process.

In reality, SCIs are organized to form configuration objects that may be catalogued in the project database with a single name. A configuration object has a name, attributes and is "connected" to other objects by relationships. In **Figure 7.3** the configuration objects **design specification, data model, module N, source code,** and **test specification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is **data model** and Software Configuration

module N are part of the object **design specification.** A double headed straight arrow indicates an interrelationship. If a change were made to the **source code** object interrelationships enable a software engineer to determine what other objects (and SCIs) might be affected.

3.2 The SCM Process

Software configuration management is an important element of software quality assurance. Its primary responsibility is the control of change. However, SCM is also responsible for the identification of individual SCIs and various versions of the software the auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration.

Any discussion of SCM introduces a set of complex questions:

- How does an organization identify and manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How an organization controls changes before and after software is released to a customer?
- Who has responsibility for approving and prioritizing changes?
- How can we assure that changes have been made properly?
- What mechanism is used to apprise others of changes that are made?

These questions lead us to the definition of five SCM tasks: identification, version control, change control, configuration auditing and reporting.

3.3 Identification of Objects in the Software Configuration

To control and manage software configuration items each must be separately named and then organized using an object-oriented approach. Two types of objects can be identified: basic objects and aggregate objects. A basic objects air a “unit of text” that has been created by a software engineer during analysis, design, coding of testing. For example, a basic object might be a section of a requirements specification, a source listing for a module or a suite of test cases that are used to exercise the code. An aggregate object is a collection of basic objects and other aggregate objects. In **Figure 7.3 design specification** is an aggregate object. Conceptually it can be viewed as a named list of pointers that specify basic objects such as **data model** and **module N.**

Each object has a set of distinct features that identify it uniquely: a name. A description, a list of resources and a “realization”; the object name is a character string that identifies the object unambiguously. The object description is a list of data items that identify:

- The SCI type (e.g., document, program, data) that is represented by the object;
- A project identifier; and change and /or version information.

Resources are “entities that are provided, processed, referenced or otherwise required by the object” [CHO 89]. For example, data types, specific functions or even variable names may be considered to be object and null for an aggregate object.

Configuration object identification must also consider the relationships that exist between named objects. An object can be identified as **<part-of>** defines a hierarchy of objects. For example, using the simple notation.

E-R diagram 1-4 <part-of> data model;

Data model <part-of> Design Specification;

We create a hierarchy of SCIs.

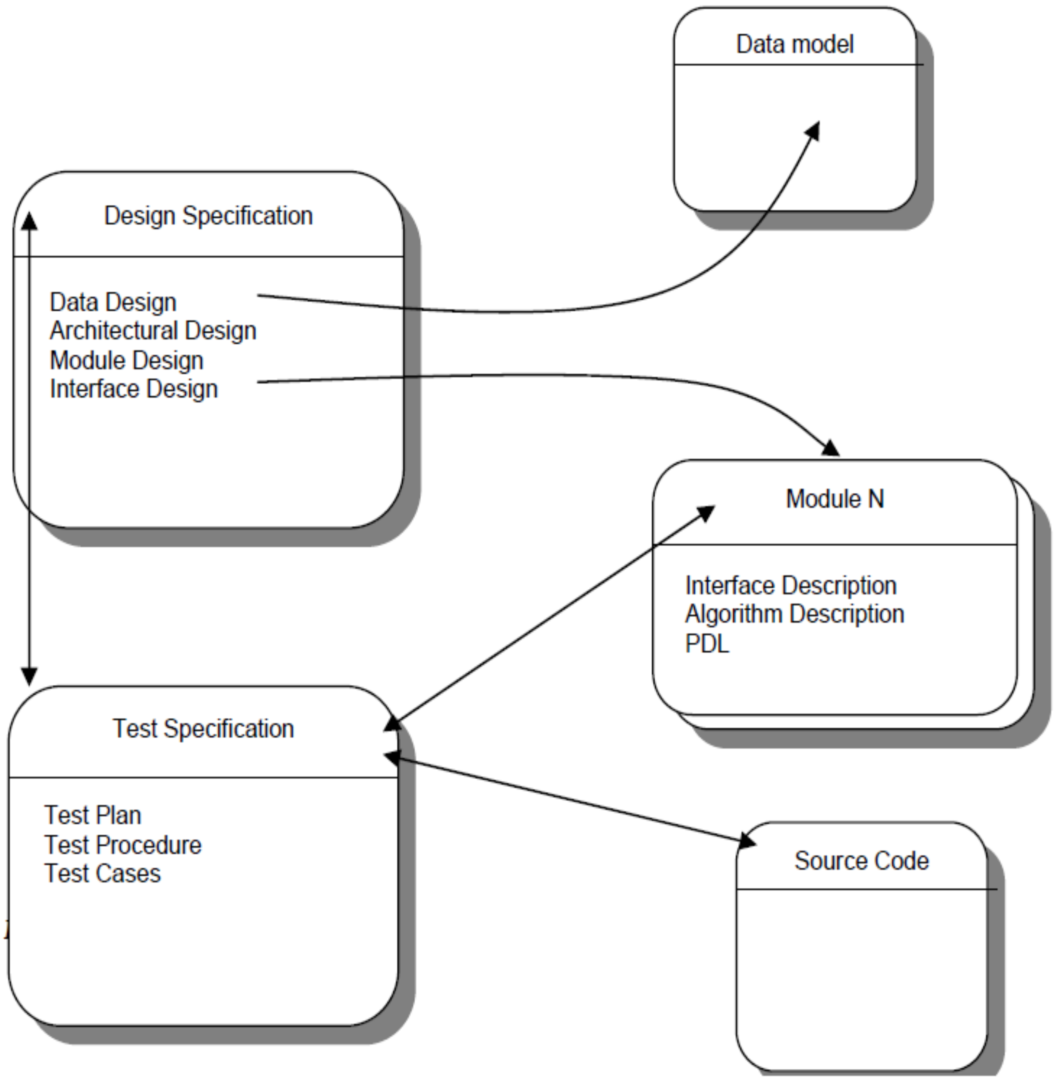


Figure 7.3 Configuration objects

It is unrealistic to assume that the only relationship among objects in an object hierarchy are align direct paths of the hierarchical tree. In many cases, objects are interrelated across branches of the objects hierarchy. For example, data model is interrelated to data flow diagrams (assuming the use of structured analysis) and also interrelated to a set of test cases for a specific equivalence class. These cross-structural relationships can be represented in the following manner:

Data model <interrelated> data flow model;

Data model <interrelated> test case class m;

In the first case, the interrelationship is between a composite object while the second relationship is between an aggregate object (**data model)**, and a basic object (**test case class m)**.

The interrelationship between configuration objects can be represented with a module interconnection language (MIL). A MIL describes interdependencies among configuration objects and enables any version system to be constructed automatically.

The identification scheme for software objects must recognize that objects evolve throughout the software process. Before an object is base lined, it is change many times and even after a baseline has been established, change may be quite frequent. It is possible to create an evolution graph any object.

The evolution graph describes the change history of the objects and comes object 1.1. Minor corrections and changes result in versions1.1.1 and 1.1.2 which is followed by a major update that is object1.2. The evolution or object1.0 continues through 1.3 and 1.4 but at the same time a major modification to the object results in a new evolutionary path version 2.0 Both versions are currently supported.

It is possible that changes may be made to any version, but not necessarily to all versions. How does the developer reference all modules, documents and test cases for version1.4? How does the marketing department know what customers currently have version 2.1? How can we be sure that changes to version2.1 source code are properly reflected in corresponding design documentation? A key element in the answer to all of the above questions is identification.

A variety of automated SCM tools (e.g., CCC, RCS, SCCS, Aide-de-Camp) have been developed to aid in identification (and other SCM) tasks. In some cases, a tool is designed to maintain full copies of only the most recent version. To achieve earlier versions (of documents of programs) changes (catalogued by the tool) are “subtracted” from the most recent version. This scheme makes the current configuration immediately available and other versions easily available.

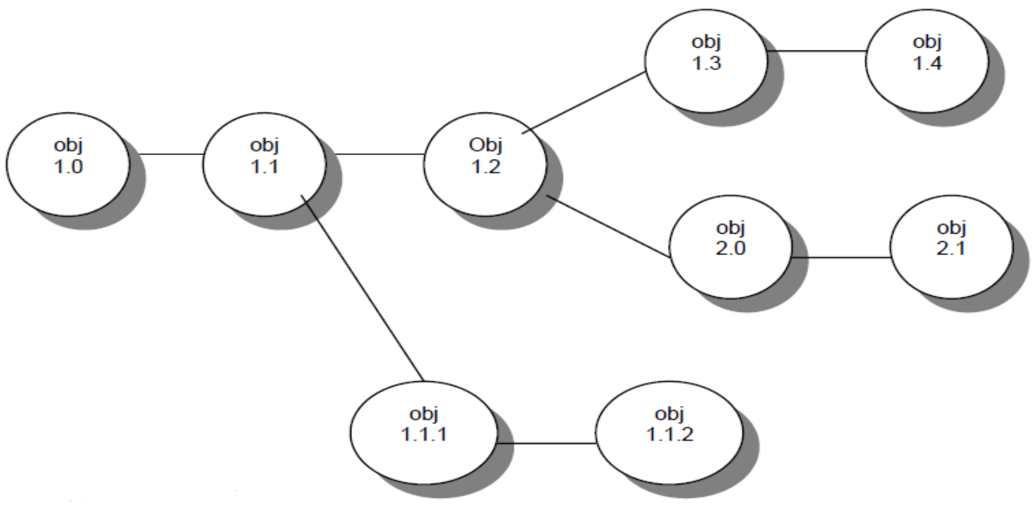


Figure 7.4 Evolution Graph

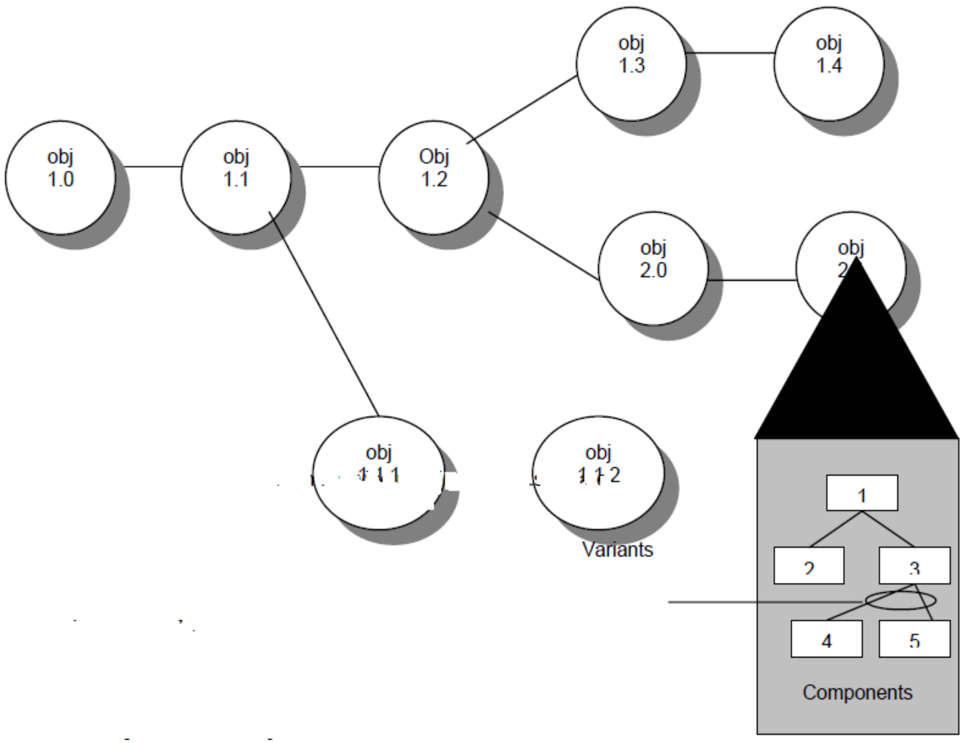


Figure 7.5 Versions and Variants

3.4 Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software engineering process Clemm describes version control in the context of SCM:

Configuration managements allows a user to specify alternative configuration of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified [and constructed] by describing the set of desired attributes.

The “attributes” mentioned above can be as simple as a specific version number that is attached to each object or as complex as a string of Boolean variables (switches) that indicate specific types of functional changes that have been applied to the system.

One representation of the different versions of a system is the evolution graph presented in **Figure 7.4,** each node on the graph is an aggregate object that is a complete version of the software. Each versions of the software are a collection of SCIs (source code, documents, data) and each version may be composed of different variants. To illustrate this concept, consider a versions f a simple program that is composed of components 1,2,3,4 and 5 (**Figure 7.5)** Component 4 is used only when the software is implemented using color displays. Component 5 is implemented when monochrome displays are available. Therefore, two variants of the version can be defined: (1) Components 1, 2, 3, 4; and (2) Components1, 2, 3 and 5.

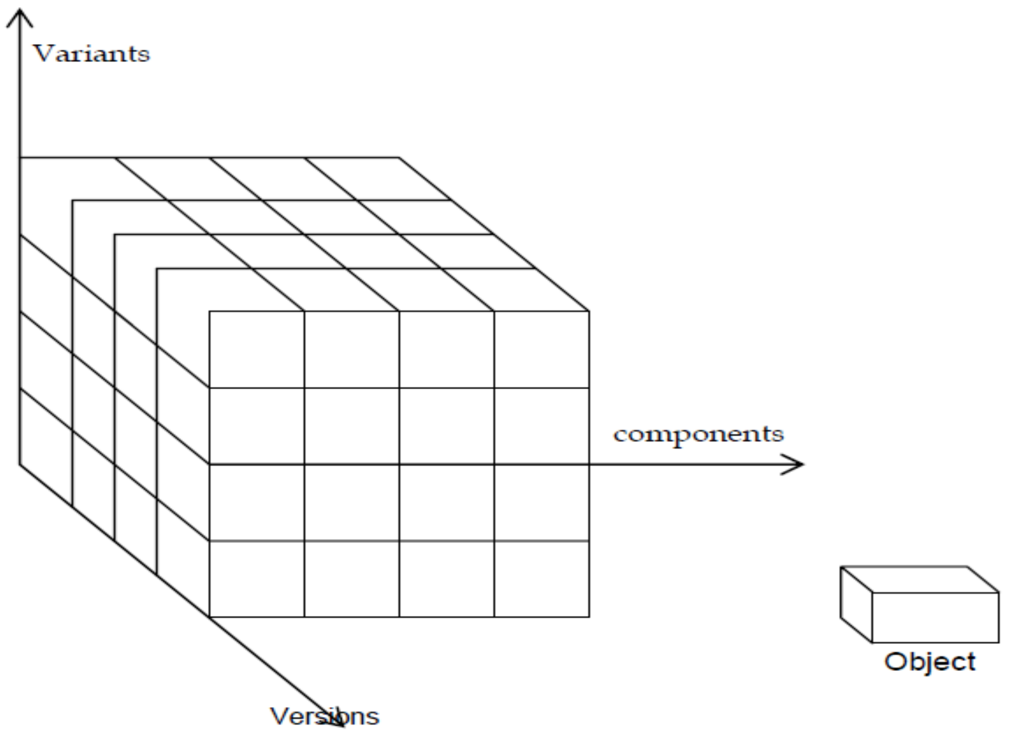


Figure 7.6 Object pool representations of components, variants and versions

To construct the appropriate variant of a given version of a program, each component can be assigned an “attribute–topple “a list of features that will define whether the component should be used when a particular variant of a software version is to be constructed. One or more attributes is assigned of each variant. For example, a color displays are to be supported.

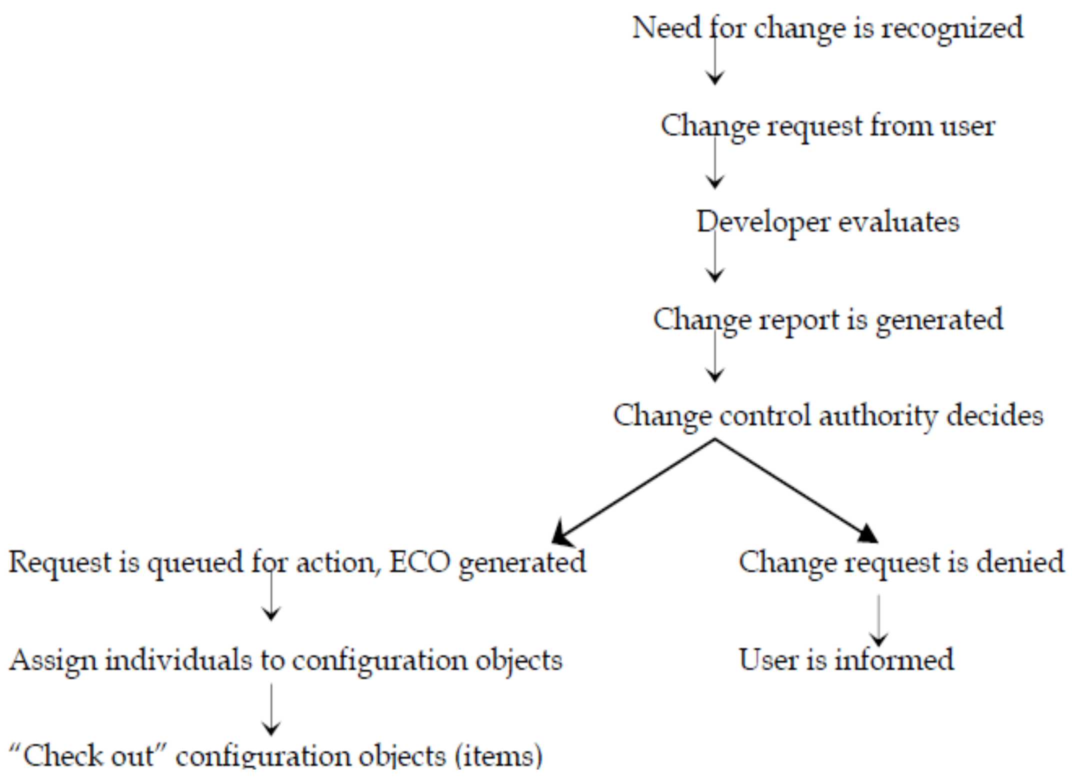
Another way to conceptualize the relationship between components, variants and version is to represent them as an object pool. As 7**.6** shows the relationship between configuration objects and components, variants and versions can be represented as a three-dimensional space. A component is composed of a collection of objects at the same revision level. A variant is a different collection of objects at the same revision level and therefore coexists in parallel with other variants. A new version is defined when major changes are made to one or more objects.

A number of different automated approaches to version control have been proposed over the past decade. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process of construction. In early systems such as SCCS, attributes took on numeric values. In later systems such as RCS, symbolic revision keys were used, Modern systems such as NSE or DSEE create version specifications that can be used to construct variants or new versions. These systems also support the base lining concept, thereby precluding uncontrolled modification (or deletion) of a particular version.

3.5 Change Control

For a large software development project, uncontrolled change rapidly leads to chaos. 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 7.7.** A change request is submitted and evaluated to assess technical merit potential side effects, overall impact on other configuration objects and system functions and the projected cost of the change. The results of the evaluation are presented as a change report that is used by a change control authority (CCA) a person or group who makes a final decision on the status and priority of the change. An engineering change order (ECO) is generated for each approved change. The ECO describes the change to be made; the constraints that must be respected, and the criteria for review and audit, the object to be changed is “checked out” of the project database the change is made and appropriate SQA activities are applied. The object is then “checked in” to the database and appropriate version control mechanism is used to create the next version of the software.

The “Check in” and” Check out” processed implement two important elements of change control access control and synchronization control. Access control governs which software engineers have the authority to access and modify a particular configuration object. Synchronization control helps to ensure that parallel changes performed by two different people don’t overwrite one another.


Need for change is recognized
Change request from user
Developer evaluates
Change report is generated
Change control authority decides
Request is queued for action, ECO generated
Change request is denied
Assign individuals to configuration objects
User is informed
“Check out” configuration objects (items)

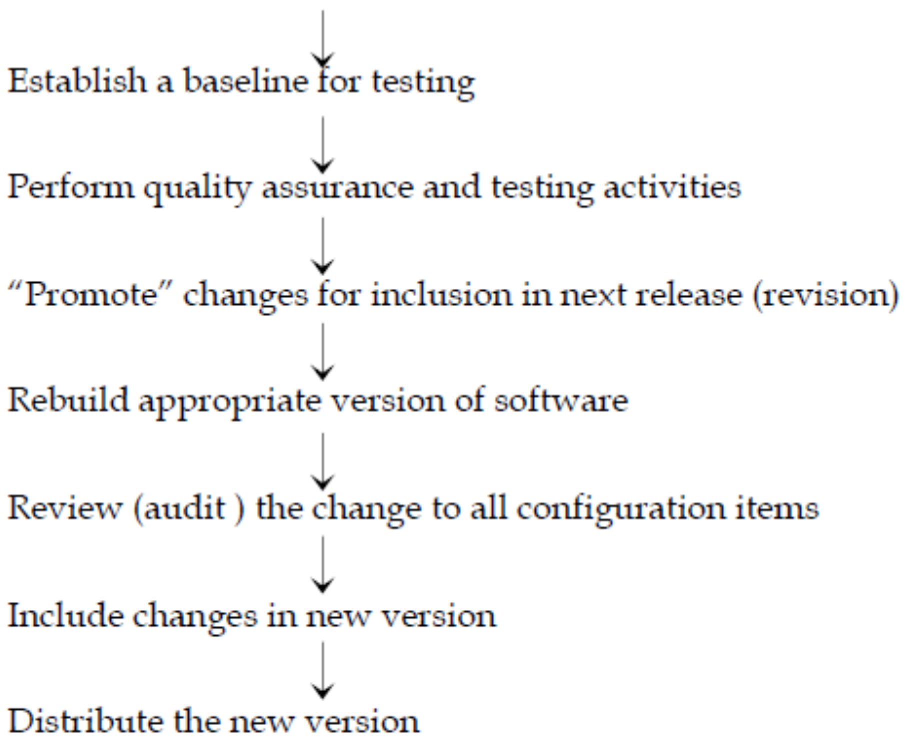

Establish a baseline for testing
Perform quality assurance and testing activities
"Promote" changes for inclusion in next release (revision)
Rebuild appropriate version of software
Review (audit) the change to all configuration items
Include changes in new version
Distribute the new version

Figure 7.7 The Change Control Process

Access and synchronization control flow is illustrated schematically in **Figure 7.8** based on approved change request and ECO, a software engineer checks out a configuration object. An access control function ensures that the software engineer has authority to check out the object, and synchronization control locks the object in the project database so that no updates can be made to it until the version currently checked out has been replaced. Note that other copies can be checked out, but other updates cannot be made. A copy of the base lined object called the “extracted version” is modified by the software engineer. After appropriate SQA and testing the modified version of the object is checked in and the new baseline object is unlocked.

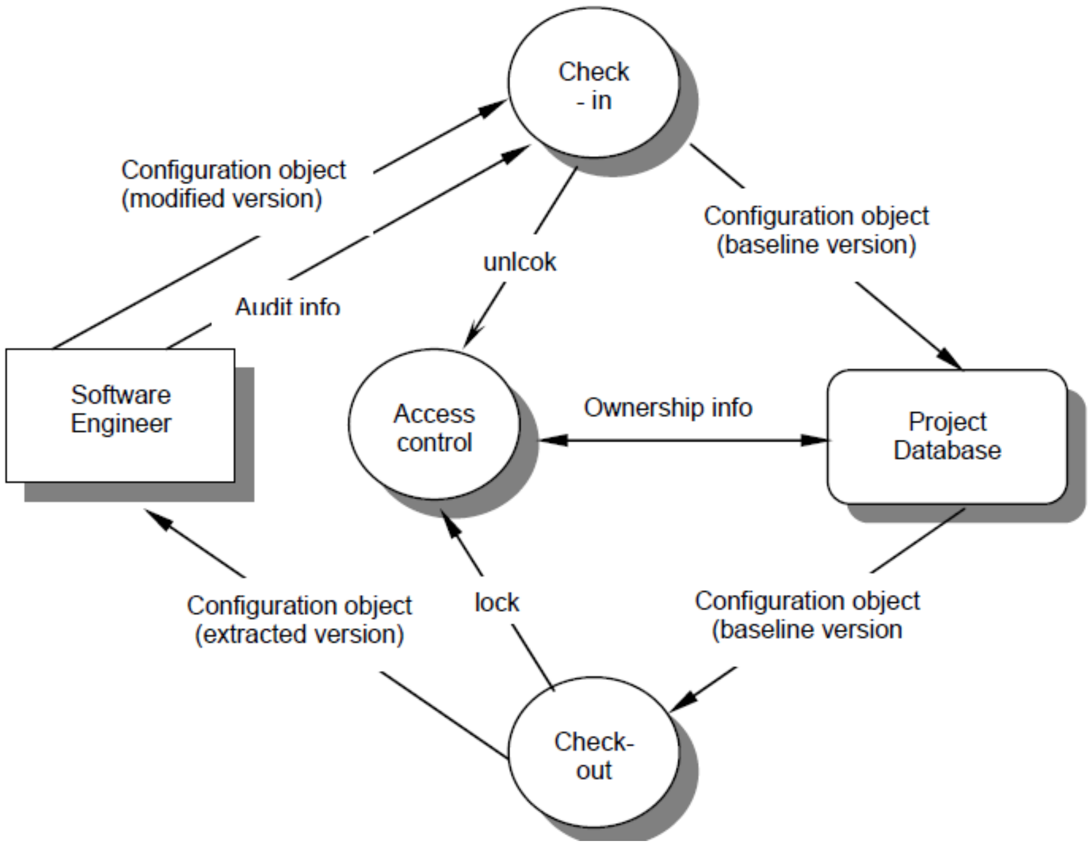


Figure 7.8 Accesses and Synchronization Control

Prior to an SCI becoming a baseline, only informal change control need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not impact broader system requirements that lie outside the developer's scope of work). Once the object has undergone formal technical review and had been approved, a baseline is created. Once an SCI becomes a baseline project level change control is implemented. Now, to make a change the developer must gain approval from the project manager (if the change is "local") or from the CCA if the change impacts other SCIs. In some cases, normal generation of change requests change reports, and ECOs is dispensed with. However, assessment of each change is conducted and all changes are tracked and reviewed.

When the software product is release to customer's formal change control is instituted. The formal change control procedure has been outlined in **figure 7.7.**

The change control authority (CCA) plays an active role in the second and third layer of control. Depending on the size and character of a software project, the CCA may be comprised of one person – the project manager – or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing, etc.). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change impact hardware? How will the change impact performance? How will the change modify the customer's perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

3.6 Configuration Audit

Identification, version control, and change control help the software developer to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can we ensure that the change has been properly implemented? The answer is twofold: (1) formal technical reviews and (2) the software configuration audit.

The formal technical review focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, and potential side effects. A formal technical review should be conducted for all but the most trivial changes.

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:

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a formal technical review been conducted to assess technical correctness?
3. Have software engineering standards been properly followed?
4. 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?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a formal technical review. However, when SCM is a formal activity, the SCM audit is conducted separately by the quality assurance group.

3.7 Status Reporting

Configuration status reporting (sometimes called status accounting) is an SCM task that answers the following questions; (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information of configuration status reporting is illustrated in **Figure 7.7.** Each time is SCI is assigned a new 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 change is made. Each time a configuration audit is conducted the results are reported as part of the CSR task. Output from CSR may be placed in an on-line database so that software developers or maintainers can access change information by keyword category. In addition, a SCR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

Configuration status reporting plays a vital role in the success of a large software development project. When many people dare involved it is likely that "the left hand not knowing what the right hand is doing" syndrome will occur. Two developers may attempt to modify the same SCI with different and conflicting intent. A software engineering team may spend months of effort building software to a obsolete for a proposed change is not aware that the change is being made. CSR helps to eliminate these problems by improving communication among all people involved.

3.8 SCM Standards

Over the past two decades a number of software configuration management standards have been proposed. Many early SCM standards such as MIL-STD-483, DOD-STD-480A and MIL-STD-1512A, focused on software developed for military applications. However more recent ANSI/IEEE standards such as ANSI/IEEE Std. No. 828-1983, Std, No. 1042-1987 and Std, No.1028-1988 [IEEE94] are applicable for commercial software and are recommended for both large and small software engineering organizations.

Self-Assessment Exercise(s)

1. Explain Software configuration management?

Self-Assessment Answer(s)

Explain Software configuration management?

Software Configuration Management is the discipline that enables us to keep evolving software products under control, and thus contributes to satisfying quality and delay constraints.
SCM emerged as a discipline soon after "software crisis" was identified, i.e. when it was understood that programming does not cover everything in Software Engineering (SE), and that other issues were hampering SE development, like architecture, building, evolution and so on.

SCM emerged, during the late 70s and early 80s, as an attempt to address some of these issues; this is why there is no clear boundary to SCM topic coverage. In the early 80s SCM focused in programming in the large (versioning, rebuilding, composition), in the 90s in programming in the many (process support, concurrent engineering), late 90s in programming in the wide (web remote engineering).
Currently, a typical SCM system tries to provide services in the following areas:
a. <u>Managing a repository of components:</u> There is a need for storing the different components of a software product and all their versions safely. This topic includes version management, product modeling and complex object management.
b. <u>Help engineers in their usual activities:</u> SE involves applying tools to objects (files). SCM products try to provide engineers with the right objects, in the right location. This is often referred as workspace control.
c. <u>Compilation and derived object control is a major issue:</u> Process control and support. Later (end 80s), it became clear that, if not the, major issue is related to people. Traditionally, change control is an integral part of an SCM product; currently the tendency is to extend process support capability beyond these aspects.

4.0 Summary/Conclusion

- Software configuration management is an umbrella activity that is applied throughout the software process. SCM identifies controls, audits and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All information produced as part of the software process becomes part of a software configuration. The configuration is organized in manner that enables orderly control of change.

- The software configuration is composed of a set of interrelated objects also called software configuration items that are produced as a result of some software engineering activity. In addition to documents programs and data the development environment that is used to create software can also be placed
- Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a base lined object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Basic and aggregate objects forms object pool from which variants and versions are created. Versions control is set of procedures and tools for managing the use of these objects.
- Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change and culminates with a controlled update of the SCI that is to be changed.
- The configuration audit is a SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

5.0 Tutor-Marked Assignments

1. Give a Short Note on SCM Process?
2. State and explain 5 software configuration items

6.0 References/Further readings

[BAB86] Babich, W.A., Software Configuration Management, Addison-Wesley, 1986.

[BAC98] Bach, J., “The Highs and Lows of Change Control,” Computer, vol. 31, no. 8, August 1998, pp. 113–115.

[BER80] Bersoff, E.H., V.D. Henderson, and S.G. Siegel, Software Configuration Manageent, Prentice-Hall, 1980.

[CHO89] Choi, S.C. and W. Scacchi, "Assuring the correctness of a Configured Software Description," Proc. 2nd Intl. Workshop on Software Configuration Management, ACM,Princeton, NJ,October 1989, pp. 66–75.

[CLE89] Clemm, G.M., "Replacing Version Control with Job Control," Proc. 2nd Intl.Workshop on Software Configuration Management, ACM, Princeton, NJ, October 1989, pp. 162–169.

[GUS89] Gustavsson, A., "Maintaining the Evaluation of Software Objects in an Integrated Environment,"Proc.2nd Intl. Workshop on Software Configuration Management, ACM, Princeton, NJ, October 1989, pp. 114–117.

[HAR89] Harter, R., "Configuration Management," HP Professional, vol. 3, no. 6, June 1989.

[IEE94] Software Engineering Standards, 1994 edition, IEEE Computer Society, 1994.

[LIE89] Lie, A. et al., "Change Oriented Versioning in a Software Engineering Database, "Proc. 2nd Intl. Workshop on Software Configuration Management, ACM, Princeton, NJ, October,1989, pp. 56–65.

[NAR87] Narayanaswamy, K. and W. Scacchi, "Maintaining Configurations of Evolving Software Systems," IEEE Trans. Software Engineering, vol. SE-13, no. 3, March 1987, pp. 324–334.

[REI89] Reichenberger, C., "Orthogonal Version Management," Proc. 2nd Intl. Workshop on Software Configuration Management, ACM, Princeton, NJ, October 1989, pp.137–140.

Unit 2

Design Concepts and Principles

Contents

1.0 Introduction

Design is the first step in the development phase for any engineering product or system. The designer's goal is to produce a model or representation of an entity that will later be built. This lecture gives you an idea about Software Design and Software Engineering, Design principles, process and concepts and also about Software Architecture.

2.0 Learning Outcome:

In this unit the following will be learnt:

1. About Software Design
2. About Design Process
3. About Design Principles
4. About Design Concepts
5. Design Documentation

3.0 Learn Content

3.1 Software Design and Software Engineering

Software design sits at the technical kernel of the software engineering process and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities - design, code generation, and testing - that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

Each of the elements of the analysis model provides information that is required to create a design model. The flow of information during software design is illustrated in following figure 8.1. Software requirements, manifested by the data, functional, and behavioral models, feed the design step. Using one of a number of design methods (discussed in later chapters), the design step produces a data design, an architectural design, an interface design, and a procedural design.

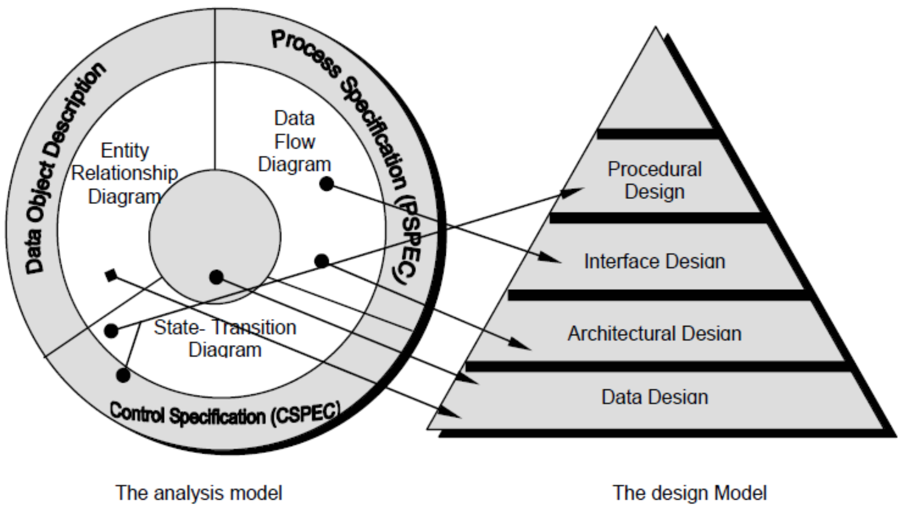


Figure 8.1 Translating the analysis model into a software design

The data design transforms the information domain created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity.

The architectural design defines the relationship among major structural elements of the program. This design representation - the modular framework of a computer program - can be derived from the analysis model(s) and the interaction of subsystems defined within the analysis model.

The interface design describes how the software communicates within itself, to systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control). Therefore, the data and control flow diagrams provide the information required for interface design.

The procedural design transforms structural elements of the program architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, CSPEC, and STD serve as the basis for procedural design.

During design we make decisions that will ultimately affect the success of software construction, and as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word-quality. Design is the place where quality is fostered in software development. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all software engineering and software maintenance steps that follow. Without design, we risk building an unstable system-one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in. the software engineering process, when time is short and a lot (in terms of money) have already been spent

3.2 The Design Process

Software design is an interactive process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction - a level that can be directly traced to specific data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is subtler.

Design and Software Quality

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs discussed in following chapters. Three characteristics that serve as a guide for the evaluation of a good design:

• The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

• The design must be a readable, understandable guide for those who generate code and for those who test and subsequently maintain the software.

• The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

The following are guidelines on design:

1. A design should exhibit a hierarchical organization that makes intelligent use of control among elements of software.
2. A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and sub functions.
3. A design should contain both data and procedural abstractions.
4. A design should lead to modules (e.g., subroutines or procedures) that exhibit independent functional characteristics.
5. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
6. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

The Evolution of Software Design

The evolution of software design is a continuing process that has spanned the past three decades. Early design work concentrated on criteria for the development of modular programs and methods for refining software architecture in a top-down manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches propose an object-oriented approach to design derivation.

Many design methods, growing out of the work noted above, are being applied throughout the industry.

Like the analysis methods each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes lead to design quality. Yet, each of these methods have a number of common characteristics: (1) a mechanism for the translation of an analysis model into a design representation, (2) a notation of representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, a software engineer should apply a set of fundamental principles and basic concepts to data, architectural, interface, and procedural design. These principles and concepts are considered in the sections that follow.

3.3 Design Principles

Software design is both a process and a model. The design process is a set of iterative steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software and an overall commitment to quality are critical success factors for a competent design.

The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three dimensional rendering of the house) and slowly refining it to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer program.

Basic design principles enable the software engineer to navigate the design process.

The following are sets of principles for software design:

- The design process should not suffer from "tunnel vision." A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts.
- The design should be traceable to the analysis model. Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- The design should not reinvent the wheel. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns, often called reusable design components, should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
- The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world. That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- The design should exhibit uniformity and integration. A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
- The design should be structured to accommodate change. Many of the design concepts discussed in the next section enable a design to achieve this principle.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. A well-designed computer program should never "bomb". It should be designed to accommodate unusual circumstances, and it if must terminate processing, do so in a graceful manner.

- Design is not coding, coding is not design. Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
- The design should be assessed for quality as it is being created, into after the fact. A variety of design concepts and design measures are available to assist the designer in assessing quality.
- The design should be reviewed to minimize conceptual errors. There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A designer should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

When the design principles described above are properly applied, the software engineer creates a design that exhibits both external and internal quality factors. External quality factors are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability). Internal quality factors are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.

3.4 Design Concepts

A set of fundamental software design concepts has evolved over the past three decades. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

Each helps the software engineer to answer the following questions:

- What criteria can be used to partition software into individual components?
- How is a function or data structure detail separated from a conceptual representation of the software?
- Are there uniform criteria that define the technical quality of a software design?

M. A. Jackson once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right". Fundamental software design concepts provide the necessary framework for "getting it right".

Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. The architecture of a program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a step wise fashion until programming language statements are reached.

An overview of the concept is provided by Wirth:

In each step (of the refinement), one or several instructions of the given program are decomposed into more detailed instructions. This successive

decomposition or refinement of specifications terminates when all instructions are expressed in terms of any underlying computer or programming language... As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine the program and the data specifications in parallel.

Every refinement step implies some design decisions. It is important that... the programmer be aware of the underlying criteria (for design decisions) and of the existence of alternative solutions.

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach.

Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is the statement function or information conceptually, but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

Modularity

The concept of modularity in computer software has been espoused for almost four decades. Software architecture embodies modularity; that is, software is divided into separately named and addressable components, called modules that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program comprised of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. To illustrate this point, consider the following argument based on observations of human problem solving.

Let $C(x)$ be a function that defines the perceived complexity of a problem x, and $E(x)$ be a function that defines the effort (in time) required to solve a problem x.

For two problems, p1 and p2, if

$C(p1) > C(p2)$ (1.1a)

It follows that

$E(p1) > E(p2)$ (1.1b)

As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem. Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

$C(p1+p2) > C(p1) + C(p2)$ (1.2)

Equation (1.2) implies that the perceived complexity of a problem that combines p1 and p2 is greater than the perceived complexity when each problem is considered separately. Considering equation (1.2) and the condition implied by equations (1.1), it follows that

E (p1+p2) > E (p1) +E (p2) (13.3)

This leads to a "divide and conquer" conclusion - it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in inequality (1.3) has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

It is possible to conclude from inequality (1.3) that if we subdivide software indefinitely, the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. As **figure 8.2** shows, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules mean smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grow. These characteristics lead to a total cost or effort curve shown in the **figure 8.2.** There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

The curves shown in **Figure8.2** do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of M. Under modularity or over modularity should be avoided. But how do we know "the vicinity of M"? How modular should we make software?

The answers to these questions require an understanding of other design concepts considered later in this lecture

. Another important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system.

Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

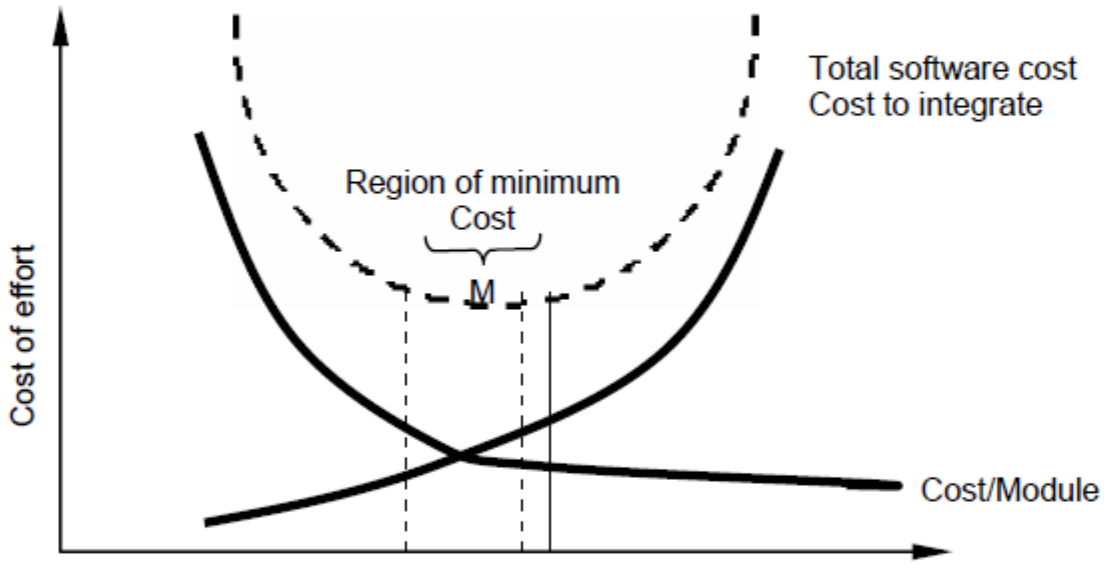


Figure 8.2 Modularity and software cost

Modular decomposability: If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

Modular composability: If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understandability: If a module can be understood as a standalone unit (without reference to other modules) it will be easier to build and easier to change.

Modular continuity: If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of change-induced side effects will be minimized.

Modular protection: If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Finally, it is important to note that a system may be designed modularly, even if its implementation must be "monolithic". There are situations (e.g., real time software, embedded software) in which relatively minimal speed and memory overhead introduced by subprograms (i.e., subroutines, procedures) is unacceptable. In such situations software can and should be designed with modularity as an overriding philosophy. Code may be developed "in-line." Although the program source code may not look modular at first glance the philosophy has been maintained, and the program will provide the benefits a modular system.

Software Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact, and the structure of the data that are used by the components. In a broader sense, however, "components" can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system.

This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

Shaw and Garland describe a set of properties that should be specified as part of an architectural design:

Structural properties: This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.

For example, objects are packaged to encapsulate both data and the processing that manipulates the data, and to interact via the invocation of methods.

Extra-functional properties: The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems: The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models. Structural models represent architecture as an organized collection of program components. Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. Process models focus on the design of the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system.

A number of different architectural description languages (ADLs) have been developed to represent the models noted above. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

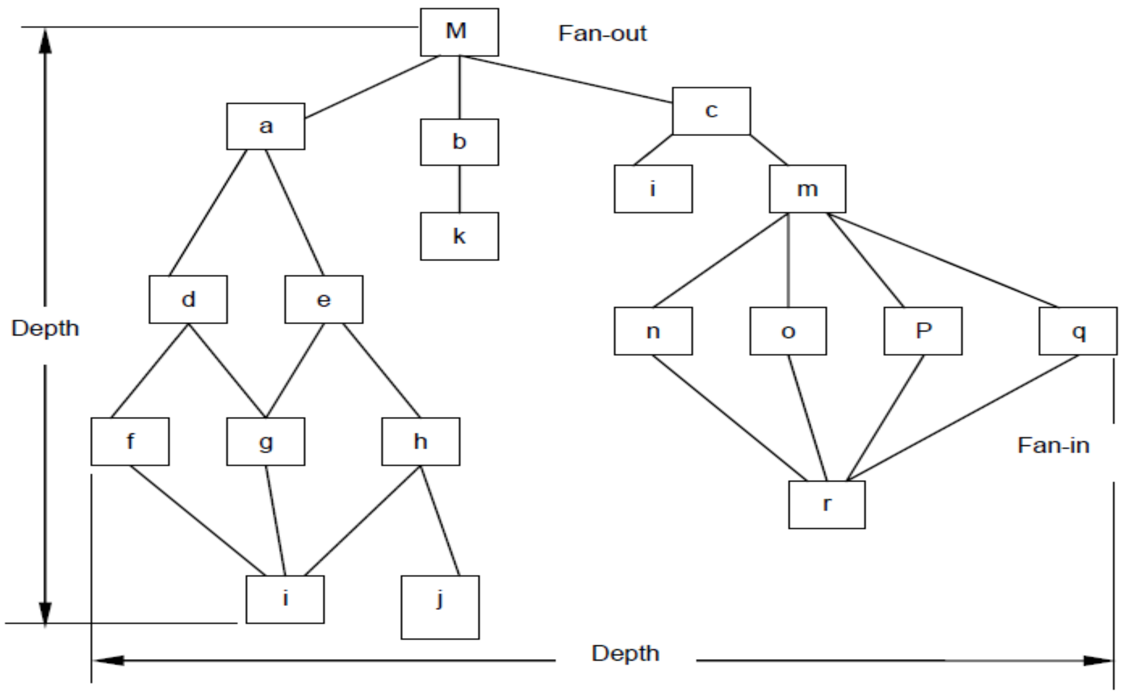


Figure 8.3 Structure Terminology

Control Hierarchy

Control hierarchy, also called program structure, represents the organization of program components and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence/order of decisions, or repetition of operations. Many different notations are used to represent control hierarchy. The most common is the tree-like diagram that represents the hierarchy. However, other notations, such as Warnier-Orr and Jackson diagrams may also be used with equal effectiveness. In order to facilitate later discussions of structure, we define a few simple measures and terms. In **Figure 8.3,** depth and width provide an indication of the number of levels of control and overall span of control, respectively. Fan-out is a measure of the number of modules that are directly controlled by another module. Fan-in indicates how many modules directly control a given module.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be super ordinate to it; conversely, a module controlled by another is said to be subordinate to the controller. For example, as shown in **Figure 18.3,** module M is super ordinate to modules a, b and c. Module h is subordinate to module e and is ultimately subordinate to module M.

Width-oriented relationships (e.g., between modules d and e), although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity. Visibility indicates the set of programs components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented, but only makes use of a small number of these data attributes. All of the attributes are visible to the module. Connectivity indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution in connected to it.

Structural Partitioning

The program structure should be partitioned both horizontally and vertically. As shown in **Figure 8.4 a,** horizontal partitioning defines separate branches of the modular hierarchy of reach major program function. Control modules, represented in a darker shade, are used to coordinate communication between and execution of program functions. The simplest approach to horizontal partitioning defines three partitions - input, data transformation (often called processing), and output. Partitioning the architecture horizontally provides a number of distinct benefits:

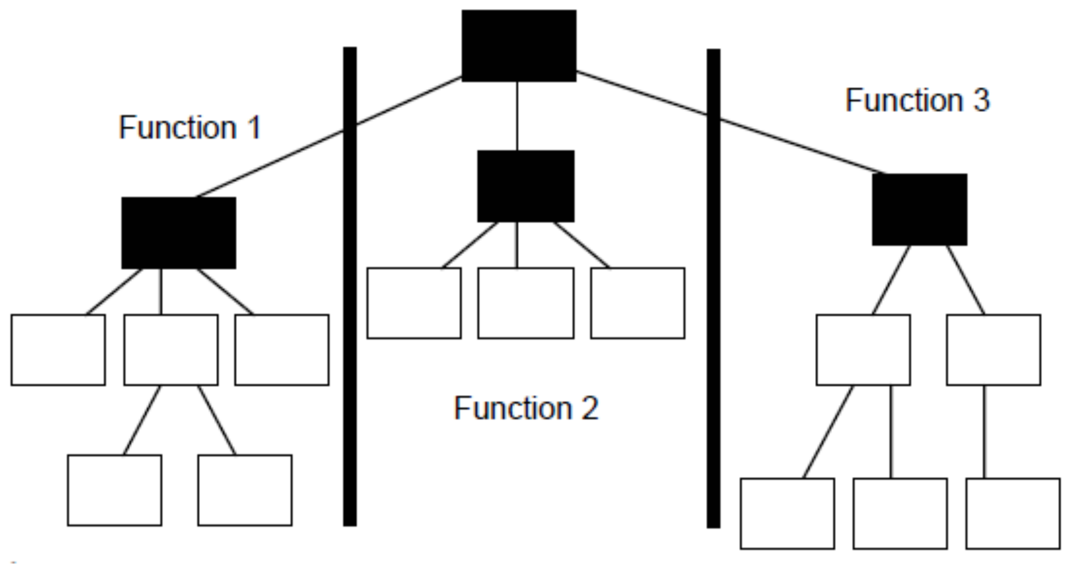
• Results in software that is easier to test

• Leads to software that is easier to maintain

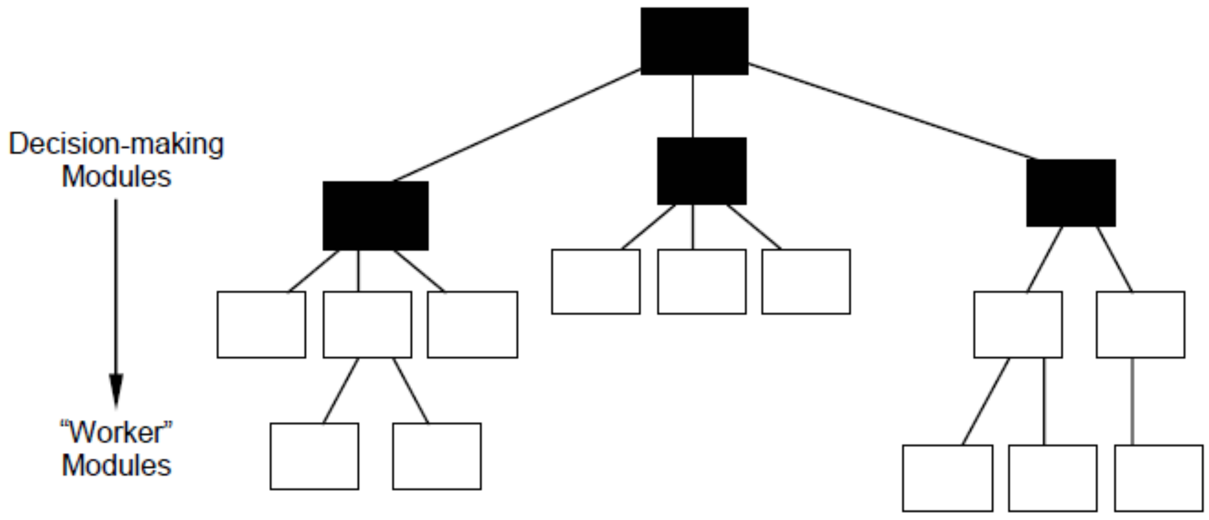
• Results in propagation of fewer side effects

• Results in software that is easier to extend

Because major functions are decoupled from one another, change tends to be less complex and extensions



(a) Horizontal partitioning



(b) Vertical partitioning

Figure 8.4 Architectural partitioning

to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

Vertical partitioning, often called factoring, suggests that control and work should be distributed top down in the program architecture. Top-level modules should perform control functions and do little actual processing work. Modules that reside low in the architecture should be the workers, performing all input, computational, and output tasks.

The nature of change in program architectures justifies the need for vertical partitioning. A change in a control module (high in the architecture) will have a higher probability of propagating side effects to modules that are subordinate to it. A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects. In general, changes to computer programs revolve around changes to input, computation or transformation, and output. The overall control structure of the program (i.e., its basic behavior) is far less likely to change. For this reason, vertically partitioned architectures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable – a key quality factor.

Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariable affect the final procedural design, data structure is as important as program structure the representation of software architecture. Data structure dictates the organization, methods of access, degree of association, and processing alternatives for information. Entire texts have been dedicated to these topics, and a complete discussion is beyond the scope of this book. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.

The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A scalar item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in storage.

When scalar items are organized as a list or contiguous group, a sequential vector is formed. Vectors are the most common of all data structures and open the door to variable indexing of information.

When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an n-dimensional space is created. The most common n-dimensional space is the two-dimensional matrix. In most programming languages, an n-dimensional space is called an array. Items, vectors, and spaces may be organized in a variety of formats.

A linked list is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner that enables them to be processed as a list. Each node contains the appropriate data organization and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

Other data structures incorporate or are constructed using the fundamental data structures described above. For example, a hierarchical data structure is implemented using multilinked lists that contain scalar items, vectors, and possible n-dimensional spaces. A hierarchical structure is commonly encountered in applications that require information categorization and association.

Categorization implies a grouping of information by some generic category (e.g., all subcompact automobiles or all 64-bit microprocessors).

Association implies the ability to associate information from different categories; for example, find all entries in the microprocessor category that cost less than $100.00 (cost subcategory), run at 100 MHz (cycle time subcategory), and are made by U.S. vendors (vendor subcategory).

It is important to note that data structures, like program structures, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector or a linked list. Depending on the level of design detail, the internal workings of stack may or may not be specified.

Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization / structure.

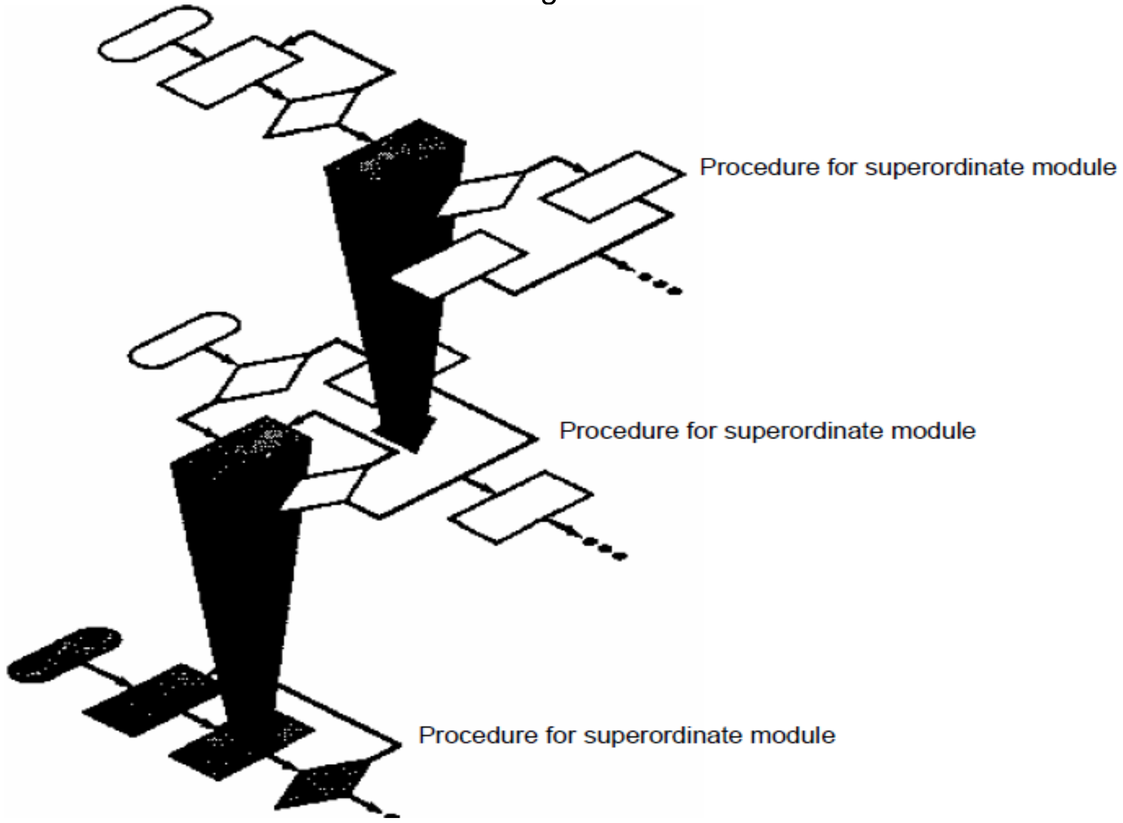

Procedure for superordinate module
Procedure for superordinate module
Procedure for superordinate module

Figure 8.5 Procedure is layered

There is, of course, a relationship between structure and procedure. Processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered.

Information Hiding

The concept of modularity leads every software designer to a fundamental question: “How do we decompose a software solution to obtain the best set of modules?” The principle of information hiding suggests that modules be “characterized by design decisions that (each) hides from all others.” In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information that is necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that comprise the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. The use of information hiding as a design criterion for modular systems provides its greatest benefits when modifications are required during testing and later, during software maintenance.

Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

3.5 The Design Model

The design principles and concepts discussed in this chapter establish a foundation for the creation of the design model that encompasses representation of data, architecture, interfaces, and procedures. Like the analysis model before it, in the design model each of these design representation is tied to the others and all can be traces back to software requirements. In figure 8.5, the design model was represented as pyramid. The symbolism of this shape is important.

A pyramid is an extremely stable object with a wide base and a low center of gravity. Like the pyramid, we want to create a software design that is stable. By establishing a broad foundation using data design, a stable mid-region with architectural and interface design, and a sharp point by applying procedural design, we create a design model hat is not easily “tipped over” by the winds of change.

It is interesting to note that some programmers continue to design implicitly, conducting procedural design as they code. This is akin to taking the design pyramid and standing it on its point an extremely unstable design results. The smallest change may cause the pyramid (and the program) to topple.

The methods that lead to the creation of the design model are presented in lectures. Each method enables the designer to create a stable design that conforms to the fundamental concepts that lead to high quality software.

Design Documentation

The document outlined in figure 8.6 can be used as template for a design specification. Each numbered paragraph addresses different aspects of the design model. The numbered sections of the design specification are completed as the designer refines his or her representation of the software.

The overall scope of the design effort is described in section I (section numbers refer to design specification outline) Much of the information contained in this section is derived from the system specification and the analysis model (software requirements specification).

I. Scope

A. System objectives

B. Major software requirements

C. Design constraints, limitations

II. Data Design

A. Data objects and resultant data structures

B. File and database structures

1. External file structure

a. Logical structure

b. Logical record description

c. Access method

2. Global data

3. File and data cross reference

III. Architectural Design

A. Review of data and control flow

B. Derived program structure

IV. Interface Design

A. Human-machine interface specification

B. Human-machine interface design rules

C. External interface design

1. Interfaces to external data

2. Interfaces to external systems or devices

D. Internal interface design rules

V. Procedural Design

For each module:

A. Processing narrative

B. Interface description

C. Design language (or other) description

D. Modules used

E. Internal data structures

F. Comments / restrictions / limitations

Requirements Cross-Reference

VII. Test Provisions

1. Test guidelines

2. Integration strategy

3. Special considerations

VIII. Special Notes

IX. Appendices

Figure 8.6 Design specification outline

Section II presents the data design, describing external file structures, internal data structures and a cross reference that connects data objects to specific files. Section III, the architectural design, indicates how the program architecture has been derived from the analysis model. Structure charts (a representation of program structure) are used to represent the module hierarchy.

Sections IV and V evolve as interface and procedural design commence. External and internal program interfaces are represented and a detailed design of the human-machine interface is described. Modules – separately addressable elements of software such as subroutines, functions, or procedures – are initially described with an English-language processing narrative. The processing narrative explains the procedural function of a module. Later, a procedural design tool is used to translate the narrative into a structured description.

Section VI of the design specification contains a requirements cross-reference. The purpose of this cross-reference matrix is (1) to establish that all requirements are satisfied by the software design, and (2) to indicate which modules are critical to the implementation of specific requirements.

The first stage in the development of test documentation is contained in section VII of the design document. Once software structure and interfaces have been established, we can develop guidelines for testing of individual modules and integration of the entire package. In some cases, a detailed specification of test procedure occurs in parallel with design. In such cases, this section may be deleted from the design specification.

Design constraints, such as physical memory limitations or the necessity for a specialized external interface, may dictate special requirements for assembling or packaging of software. Special considerations caused by the necessity for program overlay, virtual memory management, high-speed processing, or other factors may cause modification in design derived from information flow or structure.

Requirements and considerations for software packaging are presented in section VII. Secondarily, this section describes the approach that will be used to transfer software to a customer site.

Section IX of the design specification contains supplementary data. Algorithm descriptions, alternative procedures, tabular data, excerpts from other documents and other relevant information are presented as a special not or as a separate appendix. It may be advisable to develop a preliminary operations / installation manual and include it as appendix to the design document.

Self-Assessment Exercise(s)

1. What do you understand by 'good software design'?

Self-Assessment Answer(s)

'Design' could refer to many things, but often refers to 'functional design' or 'internal design'. Good internal design is indicated by software code whose overall structure is clear, understandable, easily modifiable, and maintainable; is robust with sufficient error-handling and status logging capability; and works correctly when implemented. Good functional design is indicated by an application whose functionality can be traced back to customer and end-user requirements. For programs that have a user interface, it's often a good idea to assume that the end user will have little computer knowledge and may not read a user manual or even the on-line help; some common rules-of-thumb include:

- The program should act in a way that least surprises the user
- It should always be evident to the user what can be done next and how to exit
- The program shouldn't let the users do something stupid without warning them.

4.0 Summary/Conclusion

Modularity (in both program and data) and the concept of abstraction enable the designer to simplify and reuse software components.

Refinement provides a mechanism for representing successive layers of functional detail.

Program and data structure contribute to an overall view of software architecture, while procedure provides the detail necessary for algorithm implementation. Information hiding and functional independence provide heuristics for achieving effective modularity.

Design is the technical kernel of software engineering. During design, progressive refinements of data structure, program architecture, interfaces, and procedural detail are developed, reviewed, and documented.

Design results in representations of software that can be assessed for quality.

A number of fundamental software design principles and concepts have been proposed over the past three decades.

Design principles guide the software engineer as the design process proceeds.

Design concepts provide basic criteria for design quality.

The discussion on design fundamentals can be concluded with the words of Glenford Myers:

We try to solve the problem by rushing through the design process so that enough time will be left at the end of the project to uncover errors that were made because we rushed through the design

The moral is, do not rush through it! Design is worth the effort.

5.0 Tutor-Marked Assignments

1. Explain briefly about Design principles and Design Concepts?
2. Explain briefly about Design Model and Design Documentation?

6.0 References/Further readings

[AHO83] Aho, A.V., J. Hopcroft, and J. Ullmann, Data Structures and Algorithms, Addison-Wesley, 1983.

[BAS98] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice, Addi-son-Wesley, 1998.

[BEL81] Belady, L., Foreword to Software Design: Methods and Techniques (L.J. Peters, author), Yourdon Press, 1981.

[BRO98] Brown, W.J., et al., Anti-Patterns, Wiley, 1998.

[BUS96] Buschmann, F. et al., Pattern-Oriented Software Architecture, Wiley, 1996.

[DAH72] Dahl, O., E. Dijkstra, and C. Hoare, Structured Programming, Academic Press,1972.

DAV95] Davis, A., 201 Principles of Software Development, McGraw-Hill, 1995.

[DEN73] Dennis, J., "Modularity," in Advanced Course on Software Engineering (F. L. Bauer, ed.), Springer-Verlag, New York, 1973, pp. 128–182.

Module 5

Unit 1

Design Methods

Contents

1.0 Introduction

In this lecture, we focus on Architectural Design, Interface Design, General Interaction and Procedural Design.

2.0 Learning Outcome:

In this unit the following will be learnt:

1. About Architectural Design Optimization
2. About API Design
3. About Procedural Design

3.0 Learning Content

3.1 Architectural Design Optimization

Any discussion of design optimization should be prefaced with the following comment:" Remember that an 'optimal design' that doesn't work has questionable merit." The software designer should be concerned with developing a representation of software that will meet all functional and performance requirement and merit acceptance based on design quality measures.

Refinement of program structure during the early stages of design is to be encouraged. Alternative representation may be derived refined and evaluated for the best approach. This approach optimization is one of the true benefits derived from developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design purpose should strive for the smallest number of modules that is consistent with effective modularity and the least comes data structure that adequately serves information requirements. For performance critical applications it may be necessary to "optimize" during later design iterations and possibly during coding. The software engineer should note however that a relatively small percentage of a program often accounts for a large percentage of all processing time.

It is not unreasonable to propose the following approach for performance critical software:

1. Develop and refine program structure without concern for performance critical optimization.
2. Use CASE tools that simulate run time performance to isolate areas of in efficiency.
3. During late design iterations select modules that are suspect "time hogs" and carefully develop procedures for time efficiency.
4. Code in an appropriate programming language.
5. Instrument the software to isolate modules that account for heavy processor utilization.
6. If necessary, redesign or recode in machine-dependent language to improve efficiency.

3.2 Interface Design

The architectural design provides a software engineer with a picture of the program structure like the blue print for a house the overall design is not complete without a representation of doors windows and utility connections for water electricity and telephone. The "doors, windows, and utility connections" for computer software comprise the interface design of a system.

Interface design focuses on three areas of concern :(1) the design of interfaces between softwa4re modules; (2) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities); and (3) the design of the interface between a human (i.e., the user) and the computer.

Internal and External Interface Design

The design of internal program interfaces sometimes called inter-modular interface dewing is driven by the data that must flow between modules and the characteristics of the programming language in which the software is to be implemented. In general, the analysis model contains much of the information required for inter-modular interface design. The data flow diagram describes how data objects are transformed as they move through a system. The transforms of the DFD are mapped into modules within the program structure. Therefore, the arrows flowing into and out of each DFD transform must be mapped into a design for the interface of the module that corresponds to that DFD transform.

External interface design begins with an evaluation of each external entity represented in the DFDs of the analysis model. The data and control requirements of the external entity are determined and appropriate external interfaces are designed.

Both internal and external interface designs must be coupled with data validation and error handling algorithms within a module. Because side effects propagate across program interfaces it is essential to check all data flowing from module to module to ensure that the data conform to bounds established during requirement analysis.

User Interface Design

According to, Ben Shneiderman states:

"Frustration and anxiety are part of daily life for many users of computerized information systems. They struggle to learn command language or menu selecting systems that are supposed to help them to do their job."

Some people encounter such serious cases of computer shock terminal or network neurosis that they avoid using computerized systems.

The problems to which Shneiderman alludes are real. We have all encountered "interfaces" that are difficult to learn difficult to use confusing unforgiving and in many causes, totally frustrating. Yet someone spent time and energy building each of these interfaces and it is likely that the builder did not create these problems purposely.

User interface design has as much to do with the study of people as it does with technology issues. Who is the user? How does the user learn to interface with a new computer-based system? How does the user interpret information produced by the system? What will the user expect of the system? These are only a few of the many questions that must be asked and answered as part of user interface design.

3.3 Human-Computer Interface Design

The overall process for designing a user interface design with the creation of different models of system function. The human and computer-oriented tasks that are required to achieve system function are then delimited; design issues that apply to all interface designs model and the result is evaluated for quality.

Interface Design Models

Four different models come into play when a human computer interfaces is to be designed. The software engineer creates a design model, a human engineer establishes a user model the end user develops a mental image that is often called the user's model or the system perception and the implementers of the system create a system image. Unfortunately, these models may differ significantly. The role of interface designer is to reconcile these differences and derive a consistent representation of the interface.

A design model of the entire system incorporates data architectural interface and procedural representations of the software. The requirements specification may establish certain constraints that help to define the user of the system but the interface design is often only incidental to the design model.

The user model depicts the profile of end users of the system. To build an effective user interface "all design should begin with an understanding of the intended user including profile of their age, sex, physical abilities, education, cultural, or ethnic background motivation goals and personality"

In addition, users can be categorized as:

- Novices --no syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general;
- Knowledgeable intermittent users-- reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface; and
- Knowledgeable, frequent users -- good semantic and syntactic knowledge that often leads to the "power-user syndrome" that is individuals who look of shortcuts and abbreviated modes of interaction.

The system perception is the image of the system that an end user carries in his or her head. For example, if the user of a particular word processor were asked to describe its operation the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain.

A user, who understands word processors fully but has only worked with the specific work processor once, might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

The system image couples the outward manifestation of the computer based system (the look and feel of the interface) with all supporting information (books, manuals, video tapes) that describe system syntax and semantics. When the system image and the system image and the system perception are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models the design model must have been developed to accommodate the information contained in the user model ant he system image must accurately reflect syntactic and semantic information about the interface. The interrelationship among the models is shown if Figure 9.1.

The models described in this section are "abstractions of what the user is doing or thinks he is doing or what somebody else thinks he ought to be doing when he uses an interactive style" In essence these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "Know the user, know the tasks."

Task Analysis and Modeling

Task analysis and modeling can be applied to understand the tasks that people currently perform (when using a manual or semi-automated approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the HCI. This can be accomplished by observation or by studying an existing specification of a computer- based solution and deriving a set of user tasks that will accommodate the user model the design model and the system perception.

Regardless of the overall approach to task analysis the human engineer must first define and classify tasks. One approach is stepwise elaboration. For example, assume that a small software s company wants to build a computer aided design system explicitly for interior designers. By observing a designer at work the engineer notices that the interior design is comprised of a number of major activities: furniture layout fabric and material selection wall and window covering selection presentation (to the customer), costing and shopping. Each of these major tasks can be elaborated into subtasks. For example, furniture

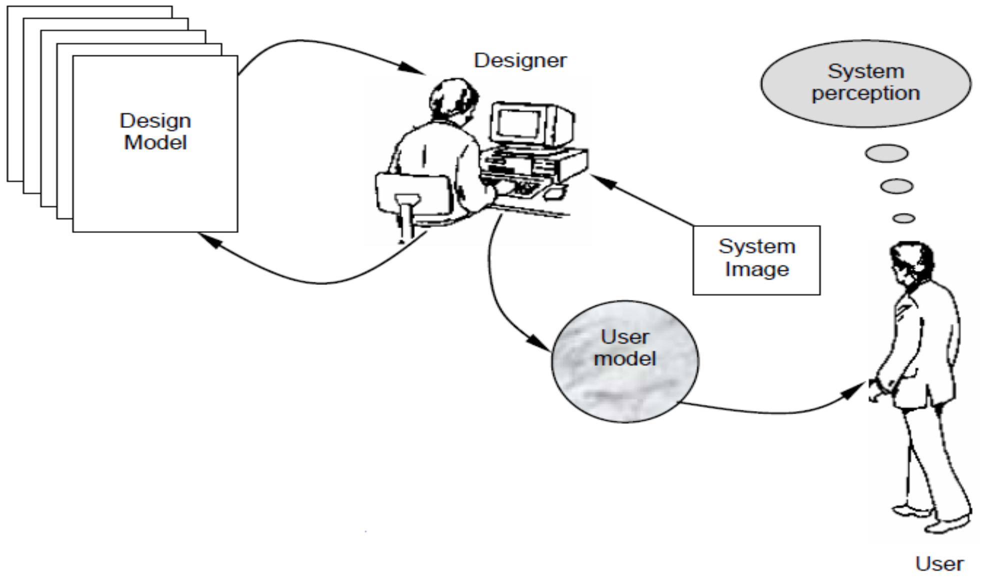


Figure 9.1 Relating interface design models

layout can be refined into the following tasks: (1) Draw floor plan based on room dimensions; (2) Place windows and doors at appropriate locations; (3) Use furniture templates to draw scaled furniture outlines on floor plan:(4) Move furniture outline to get best placement; (5) Label all furniture outline; (6) Draw dimensions to show location and (7) Draw perspective view for customer. A similar approach could be used for each f the other major tasks.

Subtasks 1 to 7 can each be refined further. Subtasks 1 to 6 will be performed by manipulation information and performing actions with the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction. The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a Design Model User Model User System Perception System Image Designer "typical" interior designer) and system perception (what the interior designer expects from an automated system).

An alternative approach to task analysis takes an object-oriented point of view. The human engineer observes the physical objects that are used by the interior designer and the actions that are applied to each object. For example, interior the furniture template would be an object in this approach to task analysis. The interior designer would select the appropriate template move it to a position on the floor plan trace the furniture outline and so forth. The design model for the interface would not describe implementation details for each of these actions but it would define user tasks that accomplish the end result (drawing furniture outline on the floor plan).

Once each task or action has been defined interface design begins. The first steps in the interface design process can be accomplished using the following approach:

1. Establish the goals and intentions for the task

2. Map each goals intention to a sequence of specific actions

3. Specify the action sequence s it will be executed at the interface level.

4. Define control mechanism i.e., the devices and action available to the user to alter the system state.

5. Show how control mechanism affects the state of the system.

6. Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues

As the design of a user interface evolves four common design issues almost always surface system response time user help facilities error information handling and command labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first linking of a problem doesn't occur until an operational prototype is available). Unnecessary interaction project delays and customer frustration is almost always the result. It is far better to establish was a design issue to be considered at the beginning of software design when changes are easy and costs are low.

System response time is the primary complaint for many interactive systems. In general system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.

System response time has two important characteristic: length and variability. If the length of time for system response time is too long user frustration and stress is the inevitable result. However, a very brief response time can also be detrimental if the user is being paced by the interface. A rapid response may force the user to rush and therefore make mistakes.

Variability refers to the deviation from average response time and in many ways it is the more important or the response time characteristics. Low variability enables the user to establish a rhythm even if response time is relatively long. For example, one second response to a command is preferable to a response that varies from 0.1 to 2.5 seconds. The user is always off balance always wondering whether something "different" has occurred behind the scenes. Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick

Two different types of help facilities are encountered integrated and add on. An integrated help facility is designed into the software from the beginning. It is often context sensitive enabling the user to select from those topics that are relevant to the actions currently being performed. Obviously, this reduces the time required for the user to obtain help and increases the "friendliness" of the interface. An add-on help facility is added to the software after the system has been built. In many ways, it is really an on-line user's manual with limited query capability. The user may have search through a list of hundreds of topics to find appropriate guidance often making many false starts and revenging much irrelevant information.

There is little doubt that the integrated help facility is preferable to the add-on approach.

A number of design issues must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction? Options include help only for a subset of all functions and actions and help for all functions.
- How will the user request help? Options include data help menu a special function key and a HELP command.
- How will help be represented? Options include a separate window a reference to a printed document and a one or two-line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include are turn button displayed on the screen and a function key or control sequence.
- How will help information be structured? Options include a "flat" structure in which all information is accessed through a keyword a layered hierarchy of information that provides increasing detail as the user proceeds into the structure and the use of hypertext.

Error messages and warning are "bad news" delivered to users of interactive systems when something has gone awry. At their worst error messages and warning impact useless or misleading information and serve only to increase user frustration. Few computer users have not encountered an error of the form;

Severe System Failure – 14A

Somewhere an explanation for error 14A must exist; otherwise why would the designers have added the identification? Yet the error message provides no real indication of what is wrong or where to look to get additional information. An error message presented in the manner shown above does nothing to assuage user anxiety or to help correct the problem.

In general, every error message or warning produced by an interactive system should have the following characteristics:

- The message should describe the problem in jargon that the user can understand.
- The messages should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).
- The message should be accompanied by an audible or visual cue. That is a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color".
- The message should be “nonjudgmental." That is the wording should never place blame on the user.

Because no one really likes bad news few users will like an error message no matter how well it is designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

The typed command was once the most common mode of interacting between user and system software and was commonly used for application of every type. Today the use of window-oriented point and pick interfaces has reduced reliance on typed commands, but many power-users continue to prefer a command-oriented mode of interacting. In many situations the user can be provided with an option software functions can be selected from a static or pull down menu or invoked through some keyboard command sequence.

A number of design issues arise when commands are provided as a mode of interaction:

1. Will every menu option have a corresponding command?
2. What form will commands take? Options include a control sequence (e.g., ^P), functions keys and a typed word.
3. How difficult will it be to learn and remember the command? What can be done if a command is forgotten?
4. Can commands be customized or abbreviated by the user?

In a growing number of applications interface designers provide a command under a user-defined name. Instead of each command being typed individually the command macro is typed and all commands implied by it are executed in sequence.

In an ideal setting convention for command usage should be established across all applications. It is confusing and often error-prone for a user to type ^D when a graphics object is to be duplicated in one application and ^D when a graphics object is to be deleted in another. The potential for error is obvious.

Implementation Tools

The process of user interface design is interactive. That is a design model is created implemented as a prototype examined by users (who fit the user model described earlier) and modified based on their comments. To accommodate this interactive design, approach a broad class of interface design and prototyping tools has evolved. Called user interface toolkits or user interface development systems (UIDS) these tools provide routines or objects that facilitate creation of windows, menus, device interaction, error messages, commands and many other elements of an interactive environment.

Using prepackaged software that can be used directly by the designers and implementer or a user interface, a UIDS provides built in mechanism for:

- Managing input devices (such as the mouse or keyboard);
- Validating user input;
- Handling errors and displaying error messages;
- Providing feedback (e.g., automatic input echo);
- Providing help and prompts;
- Handling windows and fields, scrolling within windows;
- Establishing connections between application software and the interface;
- Insulating the application from interface management functions; and
- Allowing the user to customize the interface

The functions described above can be implemented using either a language based or a graphical approach.

Design Evaluation

Once an operational user interface prototype has been created it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive" in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.

The user interface evaluation cycle takes the form shown in Figure 9.2. After the preliminary design has been completed a first level prototype is created. The prototype is evaluated by the user who provides the designer with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires rating sheets) the designer may extract information from this information (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modification is made based on user input and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. But is it possible to evaluate the quality of a user interface before a prototype is built? If potential problems can be uncovered and corrected early the number of loops through the evaluation cycle will be reduced and development time will shorten.

When a design model of the interface has been created a number of evaluation criteria can be applied during early design reviews:

1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of commands or actions specified and the average number of arguments per command or individual operations per action provides an indication of interaction time and the overall efficiency of the system.
3. The number of actions commands and system states indicated byte design model indicate the memory load on users of the system.

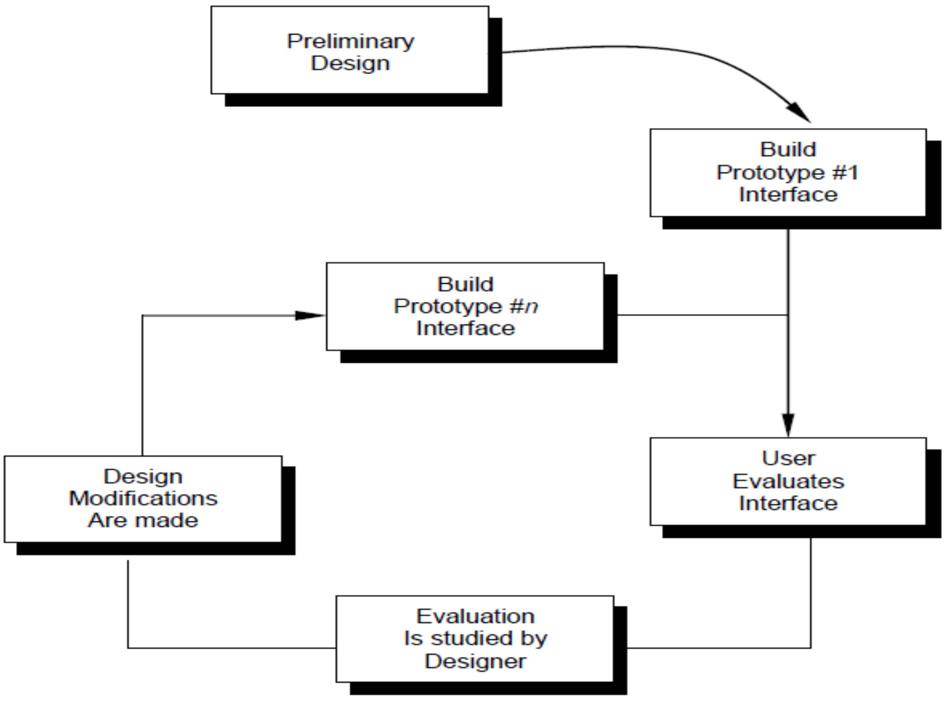

Preliminary Design
Build Prototype #1 Interface
Build Prototype #n Interface
User Evaluates Interface
Design Modifications Are made
Evaluation Is studied by Designer

Figure 9.2 The interface design evaluation design

1. Interface style help facilities and error handling protocols provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data questionnaires can be distributed to user of the prototype can be (1) simple yes/no (2) numeric (3) scaled (subjective), (4) percentage (subjective).

Examples are:

1. Were the commands easy to remember?
2. How many different commands did you use?
3. How easy was it to learn basic system operations?
4. Compared to other interfaces you've used, how would this rate? (top 1%, top10%, top 25%, top 50%, bottom 50%)

If qualitative data are desired a form of time study analysis can be conducted. Users are observed during interaction and data such as number of tasks correctly completed over a standard time period, frequency of command use, sequence of commands, time spent "looking " at eh display, number of errors types of error and error recovery time, time spent using help and number of help references per standard time period are collected and used as a guide for interface modification.

A complete discussion of user interface evaluation methods is beyond the scope of this course.

3.4 General Interaction

Guidelines for general interaction often cross the boundary into information display data entry and overall system control. They are therefore all-encompassing and are ignored at great risk.

The following guidelines focus on general interaction:

Be Consistent - Use a consistent format for menu selection command input data display and the myriad other functions that occur in a HCI.

Offer meaningful feedback - Provide the user with visual and auditory feedback to ensure that two-way communication (between user and interface) is established.

Ask for verification 0f any nontrivial destructive action If a user requests the deletion of a file indicates that substantial information's to be overwritten or asks for the terminating of a program an "Are you sure...?" the message should appear.

Permit easy reversal of most actions. UNDO or REVERSE functions have saved tens of thousands of end users from millions of hours of frustration. Reversal should be available in every interactive application.

Reduce the amount of information that must be memorized between actions - The user should not be expected to remember a list of numbers or names so that he or she can reuse them in a subsequent function. Memory load should be minimized.

Seek efficiency in dialog, motion, and thought - Keystroke should be minimized the distance a mouse must trace between picks should be considered in designing screen layout, the user should rarely encounter a situation where he or she asks, " Now what does this mean?"

Forgive mistakes - The system should protect itself from errors that might cause it to fail.

Categorize activities by function and organize screen geography accord singly One of the key benefits of the pull down menu is the ability to organize command by type. In essence the designer should strive for "cohesive" placement of commands and actions.

Provide help facilities that are context sensitive use simple auctioneers or short verb phrases to name commands - A lengthy command name is more difficult to recognize and recall. It may also take up unnecessary space in menu lists.

Information Display

If information presented by the HCI is incomplete, ambiguous or unintelligible, the application will fail to satisfy the needs of a user, Information is "displayed” in many different ways: with text pictures and sound; by placement, motion, and size; and using color resolution and even omission. The following guidelines focus on information display:

Display only that information that is relevant to the current context - The user should not have to wade through extraneous data, menus and graphics to obtain information relevant to a specific system function.

Don't bury the user with data use a presentation format that enables rapid assimilation of information*.* Graphs or charts should replace voluminous tables.

Use consistent labels, standard abbreviations and predictable colors - The meaning of a display should be obvious without reference to some outside source of information.

Allow the user to maintain visual context - If graphical representations are scaled up and down, the original image should be displayed constantly (in reduced form at the corner of the display) so that the user understands the relative location of the portion of the image that is currently being viewed.

Produce meaningful error messages Use upper and lower case, identification and text grouping to aid in understanding- Much of the information imparted by a HCI is textual and the layout and form of the text has a significant impact on the ease with which information is assimilated by the user.

Use windows to compartmentalize different types of information - Windows enable the user to "keep” many different types of information within easy reach.

Use 'analog' displays to represent information that is more easily assimilated with this form of representation- For example a display of holding tank pressure in an oil refinery would have little impact if a numeric representation were used. However, if a thermometer like display were used vertical motion a color changes could be used to indicate dangerous pressure conditions. This would provide the user with both absolute and relative information.

Consider the available geography of the display screen and use it efficiently- When multiple windows are to be user, space should be available to show at least some portion of each. In addition, screen size (a system engineering issue) should be selected to accommodate the type of application that is to be implemented.

Data Input

Much of the user's time is spent picking commands typing data and otherwise providing system input. In many applications the keyboard remains in the primary input medium, but the mouse, digitizer and even voice recognition systems are rapidly becoming effective alternatives.

The following guidelines focus on data input:

Minimize the number of input actions required of the user - Above all reduce the amount of typing that is required. This can be accomplished by using the mouse to select from predefined sets of input using "sliding scale" to specify input data across a range of values and using macros that enable a single keystroke to be transformed into a more complex collection of input data.

Maintain consistency between information display and data input - The visual characteristics of the display (e.g., text size, color, and placement) should be carried over to the input domain.

Allow the user to customize input - An expert user might decide to create custom commands or dispense with some types of warning messages and action verification. The HCI should allow this.

Interaction should be flexible but also tuned to the user's preferred mode of input - The user model will assist in determining which mode of input is preferred. A clerical worker might be very happy with keyboard input while a mange might be more comfortable using a point and pick device such as a mouse.

Deactivate commands that are inappropriate in the context of current actions - This protects the user from attempting same action that could result in an error.

Let the user control the interactive flow - The user should be able to jump unnecessary actions, change the order of required actions (when possible in the context of an application) and recover from error conditions without exiting from the program.

Provide help to assist with all input actions Eliminate "Mickey mouse" input - Do not require the user to specify units for engineering input (unless there may be ambiguity). Do not require the user to type .00 for whole number dollar amounts, provide default values whenever possible and never require the user to enter information that can be acquired automatically or computed within the program.

3.5 Procedural Design

Procedural design occurs after data, architectural and interface design have been established. In an ideal world the procedural specification required to define algorithmic details would be stated in a natural language such as English. After all members of a software development organization all speak a natural language people outside the software domain could move readily understand the specification and no new learning would be required.

Unfortunately, there is one small problem. Procedural design must specify procedural detail unambiguously and a lack of ambiguity in a natural language in not natural. Using a natural language, we can write a set of procedural steps in too many different ways. We frequently rely on context to get a point across. We often write as if a dialog with the reader were possible. For these and many other reasons a more constrained mode for representing procedural detail must be used.

Structured Programming

The foundation of procedural design was formed in the early 1960s and were solidified with the work of Edgar Dijkstra and his colleagues. In the late 1960s Dijkstra and others proposed the use of a set of existing logical constructs from which any program could be formed. The constructs emphasized “maintenance of functional domain" That is each construct has a predictable logical structure was entered at the top and exited at the bottom enabling a reader to a follow procedural flow more easily.

The constructs are sequence, condition and repetition. Sequence implements processing steps that are essential in the specification of any algorithm condition provides the facility for selected processing based on some logical occurrence and repetition provides for looping. These three constructs are fundamental to structured programming an important procedural design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable operations. Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call chunking. To understand this process, consider how this page is being read by you. You do not read individual letters; rather you recognize patterns or chunks of letters that form words or phrases. The structured constructs are logical chunks that allow reader to recognize procedural elements of a module rather than read the design or code line by line. Understanding is enhanced when readily recognizable logical forms are encountered. Any program regardless of application area or technical complexity can be designed and implemented using only the three structured constructs. It should be noted however that dogmatic use of only these constructs can sometimes cause practical difficulties.

Graphical Design Notation

"A picture is worth a thousand words" but it's rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the flowchart or box diagram, provide excellent pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.

The flowchart was once the most widely used graphical representation for procedural design. Unfortunately, it was the most widely abused method as well.

The flowchart is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition and arrows show the flow of control. Figure 9.3 illustrates the three structured constructs. Sequence is represented as two processing boxes connected by a line of control. Condition also called if-then-else is depicted as a decision diamond which if true causes then part processing to occur and if false invokes else-part processing. Repetition is represented using two slightly different forms. The do-while tests a condition and executes a loop task repetitively as long as the condition holds true. A repeat-until executes the look task first then tests a condition and repeats the task until the condition fails. The selection construct shown in the figure is actually an extension of the if-then else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

The structured constructs may be nested within one another as shown in Figure 9.4. In the figure a repeat-until forms the then part of an if-then0else (shown enclosed by the outer dashed boundary). Another if-then-else forms the else-part of the larger condition. Finally, the condition itself becomes a second block in a sequence. By nesting constructs in this manner, a complex logical schema may be developed. It should be noted that any one of the blocks in Figure 9.5 could reference another module, thereby accomplishing procedural layering implied by program structure. In general, the dogmatic use of only the structured constructs can introduce inefficiency when an escape from a set of nested loops or nested conditions is required. More important, additional complication of all logical tests along the path of escape can cloud software control flow, increase the possibility of error and have negatives impact on readability and maintainability. What can we do?

The designers left with two options: (1) The procedural representation is redesigned so that the "escape branch" is not required at a nested location in the flow of control; or (2) the structured constructs are violated in a controlled manner; that is a constrained branch out of the nested flow is designed

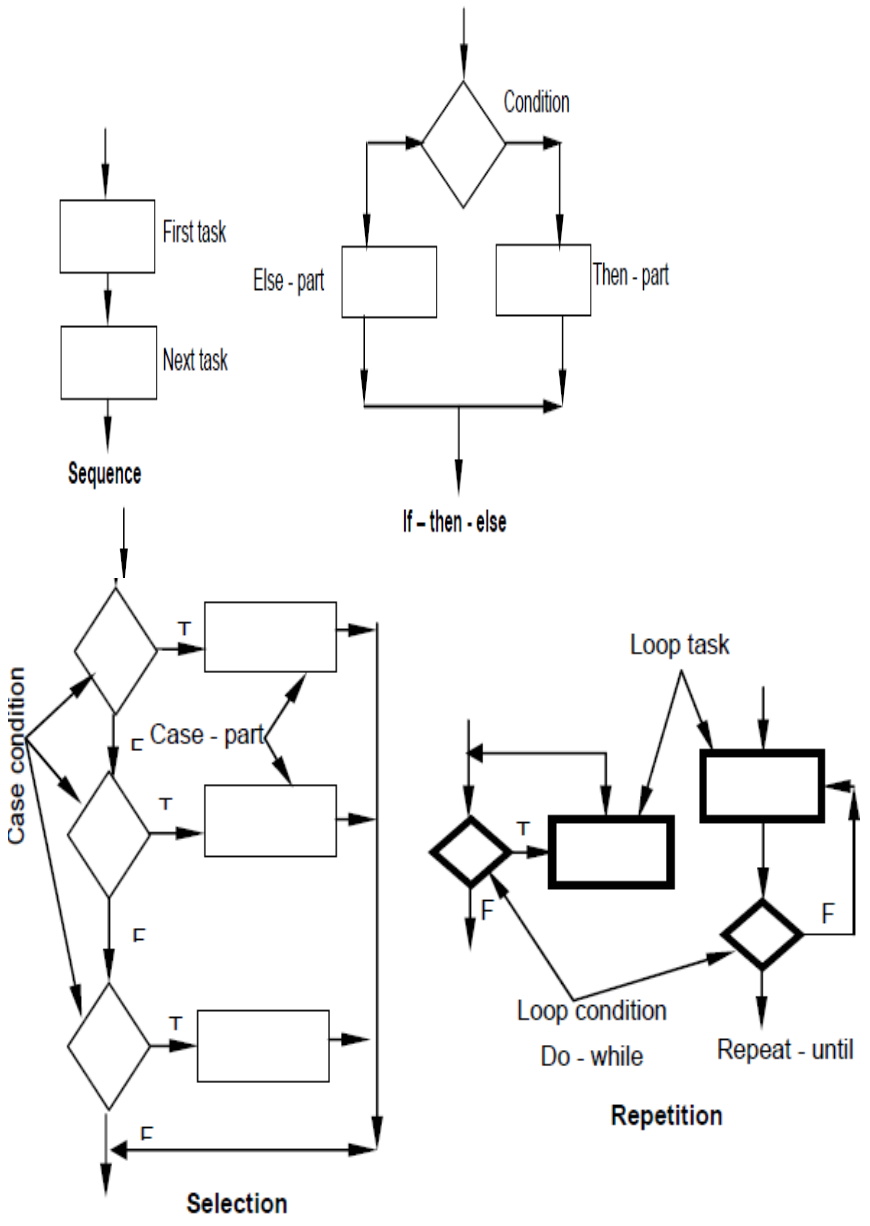


Figure 9.3

Option 1 is obviously the ideal approach but option can be accommodated without violating the spirit of structured programme

Another graphical design tool, the box diagram evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs. Developed by Nassi and Shneiderman and extended by Chapin the diagrams (also called Nassi-Shneiderman charts, N-S charts or Chapin charts) have the following characteristics:(1) functional domain (that is the scope of repetition or an if-then-else) is well defined and clearly visible as a pictorial representation; (2) arbitrary transfer of control is impossible; (3) the scope of local and/or global data can be easily determined and (4) recursion is easy to represent.

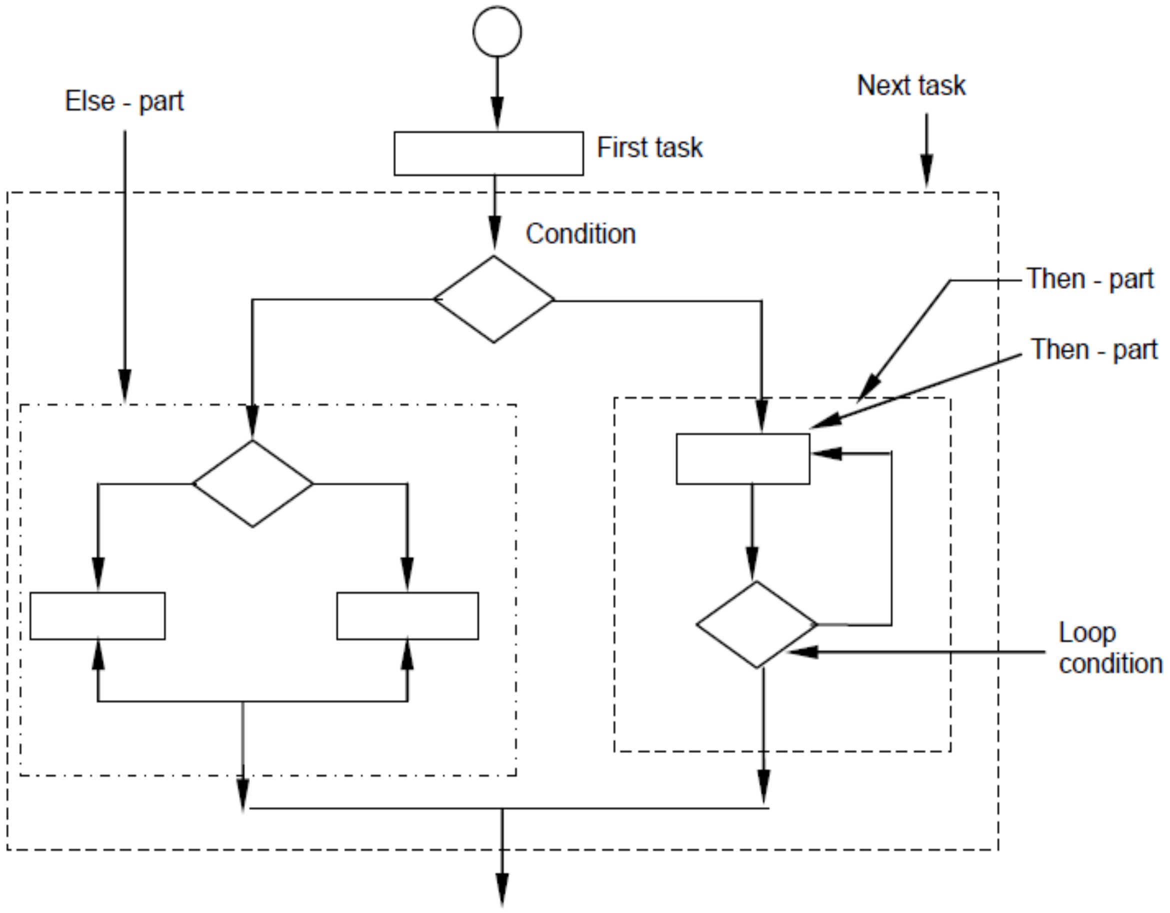


Figure 9.4 Nesting constructs

The graphical representation of structured constructs using the box diagram is illustrated in Figure 9.5. The fundamental element of the diagram is a box. To represent sequence two boxes are connected bottom to top. To represent an if-then-else a condition box is followed by a then-part box and else-part box.

Repetition is depicted with a bounding pattern that encloses the process to be repeated. Finally, selections represented using the graphical form shown at the bottom right of the figure.

Like flowcharts a box diagram is layered on multiple pages a processing elements of a module are refined. A "call" to a subordinate module can be represented by a box with the module name enclosed by an oval.

In many software applications a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions into a tabular form. The table is difficult to misinterpret and may even be used as a machine readable input to a table driven algorithm. In a comprehensive treatment of this design tool, Ned Chapin states:

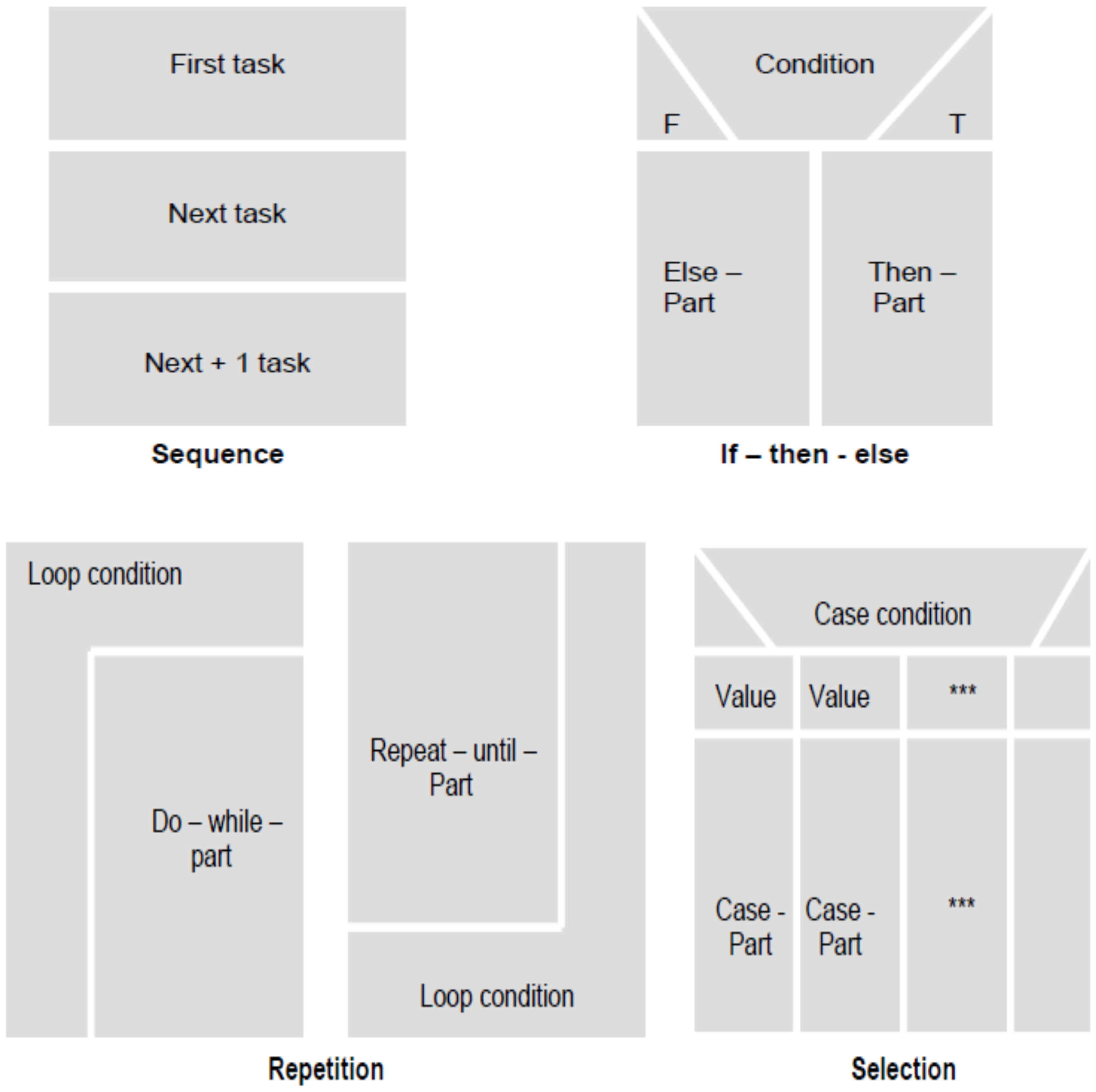


Figure 9.5 Box diagram constructs

Some old software tools and techniques mesh well with new tools and techniques of software engineering. Decision tables are an excellent example Decision tables preceded software engineering by nearly a decade but fit so well with software engineering that they might have been designed for that purpose.

Decision table organization is illustrated in Figure 9.6. The table is divided into four sections. The upper left hand quadrant contains a list of all conditions. The lower left hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right hand quadrants form a matrix that indicates conditions, combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing rule.

The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or module).
2. List all condi6tins (or decisions made) during execution of the procedure.
3. Associate specific sets conditions with specific actions, elimination impossible combinations of conditions; alternatively develop every possible permutation of conditions.
4. Define rules by indicating what action or actions occur for a set of conditions.

To illustrate the use of a decision table, consider the following excerpt from a processing narrative for a public utility billing system:

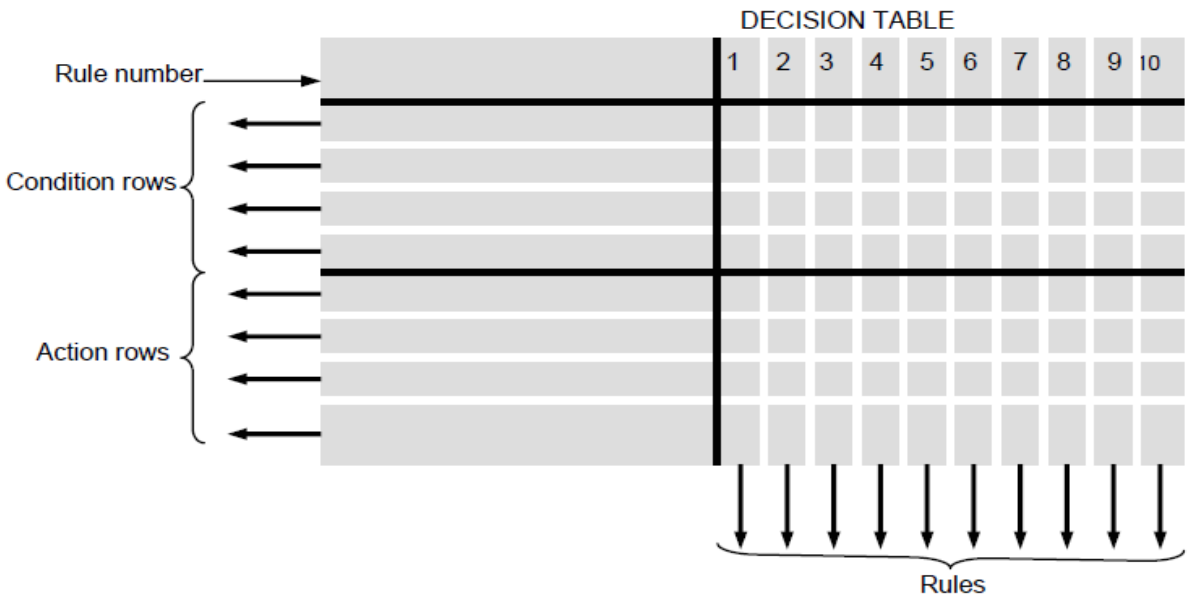


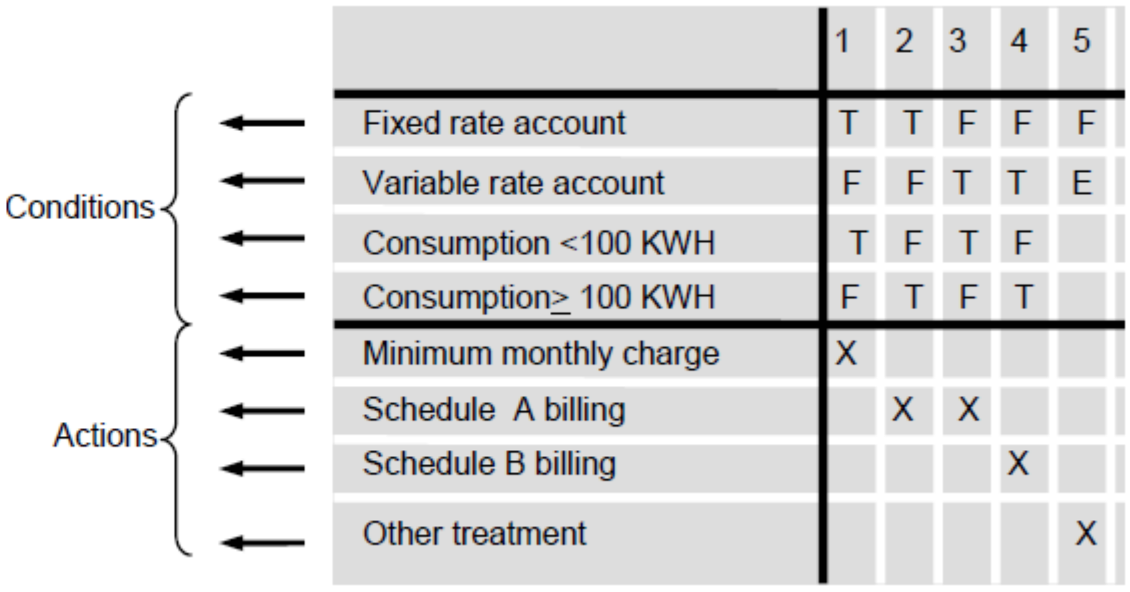
Figure 9.6 Decision table nomenclature

If the customer account is billed using a fixed rate method, a minimum monthly charge is assessed for consumption of less than 100 kWh. Otherwise computer billing applies a Schedule a rate structure. However, if the account is billed using a variable rate method, a Schedule A rate structure will apply to consumption below 100 kWh with additional consumption billed according to Schedule B. Figure 9.7 illustrates a decision table representation of the preceding narrative. Each of the five rules indicates one of five viable conditions (e.g., a "T" (true) in both fixed rate and variable rate account makes no sense in the context of this procedure) As a general rule the decision table can be effectively used to supplement other procedural design notation.

Program Design Language

Program Design Language (PDL) also called structured English or pseudo code, is "a pidgin language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)" In this chapter PDL is used as a generic reference for a design language.

At first glance PDL looks something like any modern programming language. The difference between PDL and a modern programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements. Because narrative text is embedded directly into a syntactical structure, PDL cannot be compiled. However, PDL "processors" currently exist to translate PDL into a graphical representation (e.g., a flowchart) of design and produce nesting maps a design operation index cross reference tables and a variety of other information.


Conditions
Actions

	1	2	3	4	5
Fixed rate account	T	T	F	F	F
Variable rate account	F	F	T	T	E
Consumption <100 KWH	T	F	T	F	
Consumption≥ 100 KWH	F	T	F	T	
Minimum monthly charge	X				
Schedule A billing		X	X		
Schedule B billing				X	
Other treatment					X

Figure 9.7 Resultant decision table

A program design language may be a simple transposition of a language such as Ada or C. Alternatively it may be a product purchased specifically for procedural design.

Regardless of origin a 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 processing features;
- Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures; and
- Subprogram definition and calling techniques that support various modes of interface description. Today a high order programming language is often used as the basis for a PDL. For example, Ada-PDL is widely used in the Ada community as a design definition tool. Ada language constructs and format are "mixed" with English narrative to form the design language.

A basic PDL syntax should include constructs for subprogram definition interface description and data declaration; and techniques for block structuring condition constructs repetition constrictors and I/O constructs. The format and semantics for some of these PDL constructs are presented in the section that follows.

It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling inter process, synchronization and many other features. The application design for which PDL is to be used should dictate the final form for the design language.

Self-Assessment Exercise(s)

1. What is interface design?

Self-Assessment Answer(s)

User interface design or user interface engineering is the design of computers, appliances, machines, mobile communication devices, software applications, and websites with the focus on the user's experience and interaction. The goal of user interface design is to make the user's interaction as simple and efficient as possible

4.0 Summary/Conclusion

Interface design encompasses internal and external program interfaces and the design of the user interface. Internal and external interface design are guided by information obtained from the analysis model.

The user interface design process begins with task analysis and modeling, a design activity that defines user tasks and actions using either a elaborative or object-oriented approach.

Design issues such as response time, command structure, error handling, and help facilities are considered, and a design model for the system is refined.

A variety of implementation tools are used to build a prototype for evaluation by the user.

A set of generic design guidelines govern general interaction information display, and data entry.

Design notation, coupled with structured programming concepts, enables the designer to represent procedural detail in a manner that facilitates translation to code. Graphical, tabular, and textual notations are available.

Data structure is developed, program architecture is established, modules are defined, and interfaces are established. This blueprint for implementation forms the basis for all subsequent software engineering work.

5.0 Tutor-Marked Assignments

1. What is Design Evaluation?
2. Explain briefly about Procedural Design?
3. Give a Short Note on Program Design Language?

6.0 References/Further readings

[LEA88] Lea, M., "Evaluating User Interface Designs," User Interface Design for Computer Systems, Halstead Press (Wiley), 1988.

[MAN97] Mandel, T., The Elements of User Interface Design, Wiley, 1997.

[MON84] Monk, A. (ed.), Fundamentals of Human-Computer Interaction, Academic Press, 1984.

[MOR81] Moran, T.P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," Intl. Journal of Man-Machine Studies, vol. 15, pp. 3–50.

[MYE89] Myers, B.A., "User Interface Tools: Introduction and Survey, IEEE Software, January 1989, pp. 15–23.

[NOR86] Norman, D.A., "Cognitive Engineering," in User Centered Systems Design, Earlbaum Associates, 1986.

[RUB88] Rubin, T., User Interface Design for Computer Systems, Halstead Press (Wiley), 1988.

Unit 2

Programming

Contents

1.0 Introduction

Programming is instructing a computer to do something for you with the help of a programming language. The role of a programming language can be described in two ways:

1. Technical: It is a means for instructing a Computer to perform Tasks
2. Conceptual: It is a framework within which we organize our ideas about things and processes.

According to the last statement, in programming we deal with two kinds of things:

- Data, representing ``objects'' we want to manipulate
- Procedures, i.e. ``descriptions'' or ``rules'' that define how to manipulate data.

According to Abelson and Sussman ([ABELSON, 1985, 4,])

``.... we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every language has three mechanisms for accomplishing this:

primitive expressions, which represent the simplest entities with which the language is concerned means of combination, by which compound expressions are built from simple ones, and means of abstraction, by which compound objects can be named and manipulated as units.''

A programming language should both provide means to describe primitive data and procedures and means to combine and abstract those into more complex ones.

The distinction between data and procedures is not that clear cut. In many programming languages, procedures can be passed as data (to be applied to ``real'' data) and sometimes processed like ``ordinary'' data. Conversely ``ordinary'' data can be turned into procedures by an evaluation mechanism.

2.0 Learning Outcome

In this unit the following will be learnt:

1. About Programming
2. About C++ (Brief Introduction)

3.0 Learning Content

3.1 A program = data + instructions

Data:

- there are several ***data types*** (numbers, characters, etc.)
- each individual data item must be ***declared*** and ***named***
- each individual data item must have a ***value*** before use
- *initial values* come from
    - program instructions

        - user input
        - disk files
    - program instructions can alter these *values*
    - original or newly computed *values* can go to
        - screen
        - printer
        - disk

Instructions:

- for data *input* (from keyboard, disk)
- for data *output* (to screen, printer, disk)
- *computation* of new values
- program control (decisions, repetition)
- modularization (putting a sequence of instructions into a package called a function)

Sample C++ program

```
#include <iostream>       // a.
using namespace std;      // b.
int main()               // c.
 {
 int num1, num2;         // d.
 num1 = 10;              // e.
 num2 = 15;
 num1 = num1 + num2;     // f.
 cout << "Sum is: " << num1;  // g.
 return 0;               // h.
 }
```

Notes:

- a. so we can use *cout* for output
- b. so we can use the shortcut name *cout* instead of the full name *std.cout*
- c. program name is always *main* with *int* (which stands for integer) before it
- d. *num1* and *num2* are the data items (which we often call variables)
- e. they get initial values from instructions via *assignment* (the = is the *assignment* command)
- f. then the value in *num1* is changed by adding *num1* and *num2* and putting the result back into *num1*
- g. the result is displayed via *cout*: "Sum is: 25"

- h. returns an integer: *main* "promised" to provide an integer "answer". In this program (and usually) the "answer" doesn't matter. But we have to keep the promise, so we just return 0. We will discuss this further later.
- on any line, typing // turns the rest of the line into a *comment*. A comment is not "seen" by the computer - it is not part of the program. It is used by human readers of the program to explain something in the program.
- there are no program control or modularization statements
- this program has no input
- notice the semicolons - where they are and aren't

3.2 Program Error Types

1. **Compile error** - invalid C++ statements; so your code cannot be translated to machine language. Examples:

    - integer x, y;
    - num1 = num1 + num2

In example 1 you must use int, not integer.
In example 2 the; at the end of the line is missing

2**. Link error**: one of the *functions* used in the program is not found when it's time to run the program. In the program above, there are no such functions, but we could encounter this problem later.

3. **Run-time error** - program does not run to completion. Example:

- divide by zero: set x = 10 and y = 0 and then try to compute x/y

4. **Logic error** - program runs to completion, but the results are wrong. Example:

- you really wanted x - y but coded x + y
- in calculating an average, you use the wrong number to divide

Fixing 3 and 4 is called debugging.

You must learn to understand that the computer did **what you said**, not necessarily **what you meant**.

3.3 The C++ Language - Introduction

Note: C++ is a *superset* of the C programming language. Any C code is also C++ code, but some C++ code is not C. Much of what you learn in this course could also be used in a C program. There are lots of new features and capabilities in C++. We will learn some of them (but far from all of them) in this course.

The C++ Language is made up of

- keywords/reserved words (if, while, int, etc.)

- symbols: { } = | <= ! [ ] * & (and more)
- programmer-defined names for variables and functions

These programmer-defined names:

- 1 - 31 chars long; use letters, digits, _ (underscore)
- start with a letter or _
- are case-sensitive: *Num* is ***different*** than *num*
- should be meaningful: *studentCount* is better than *s* or sc

3.4 C++ Data Types

Each data item has a **type** and a **name** chosen by the programmer. The **type** determines the **range** of possible **values** it can hold as well as the **operations** that can be used on it. For example, you can add a number to a numeric data type, but you cannot add a number to someone's name. (What would "Joe" + 1 mean?)

Common data types

The commonly-used data types

1. **int** (integer numbers)

- no decimal point, comma, or $
- leading + - allowed
- range (on our system): about plus or minus 2 billion
- int **constants** are written as: 1234 -3 43

To make a variable that your program can use, you must *declare* it.

Declaration Examples:

```
int x;                  // gives type and name; no value yet
 int x, y; // declares 2 ints; no values yet
 int population = 16000;      // declares & sets init value
```

So you could picture the results of the last two lines as:

x	y	Population
??	??	16000

Note: It is critically important that variables have values before they are used in a program. In the examples above, x and y have no values even though they have been declared. More accurately, they do have values but those values could be anything between -2 billion and +2 billion. Before you **use** x or y you would have to give them values (perhaps as shown in the first program example.) 2 and 3. **Float and double** (floating point or real numbers). These data types are commonly referred to as ***real*** variables (following mathematics usage) when it is not important which one we mean and we don't want to always have to say "float or double".

- has decimal point
- leading + - allowed
- no comma, $
- range (float) plus or minus 10 to 38th (limited to ~ 6 significant digits)
- range (double) plus or minus 10 to 308th (limited to ~12 significant digits)
- values are rarely exact
- floating point constants are written as 3.08 -32.0 0.15 9.3e7 (9.3 times 10 to the 7th power)

Declaration Examples

```
float pi = 3.1416;   //declares, names, sets init value
 double x = 3.5,        //note comma here
       y = 1245.6543; //can use > 6 digits
OR
double x   = 3.5;
double y   = 1245.6543;
 float big = 5.28e3;  // exponential notation: 5280
```

4. **char** (single characters)

- can hold just one character
- char constants are written with single quotes: 'a'

Declaration examples:

```
char ch;
char choice = 'q';
```

5. **string** (character "strings" or sequences)

- can hold 0 to many characters
- string constants are written with double quotes: "Hello, world"

Declaration Examples:

```
string s;
 string MyName = "Jim";
```

3.5 Arithmetic Operations and Examples

The arithmetic operators are:

- + addition

- - subtraction or unary negation (-5)
- * multiplication
- / division (see special notes on division below)
- % integer remainder of integer division
- Note: there is no exponentiation operator.

Notes on Division:

```
    3  R1
  _______     4 is divisor
4| 13           13 is dividend
   12           3 is quotient
    1           1 is remainder
```

In C++, a division with *2 int operands* has an *int* resulting value:

```
  13/4 --> 3        // int quotient
  13%4 --> 1        // int remainder
```

But with 1 or 2 float/double operands, the resulting value is a float or double:

```
So: 13.0 / 4 --> 3.25
    13 / 4.0 --> 3.25
```

Be aware. Forgetting this can easily cause a Logic Error.

What is the % operator good for?

Examples:

1. we can use the % operator to find if one number evenly divides another:

Is 1966 a leap year? Yes, if *1966 % 4* is 0

Is the value in the int variable num even or not? It is even if the value of *num % 2* is 0.

Arithmetic Expressions are formed by *operands* and *operators*. Operands are the values used, operators are the things done to the numbers. Parts of arithmetic expressions are often grouped by () for clarity or to affect the meaning of the expression:

```
// declare:
int x, y;
double realnum;
 // initialize
```

```
x       = 11;
y       = 2;
real num = 2.0;
```

expression	value	Notes
x + y	13	
x * y	22	
x * y + x	33	
x – y	9	
-x + y	-9	unary negation: "minus x"
x / y	5	int since both ops are int
x % y	1	rem when x divided by y
x / realnum	5.5	one op is real so result is real

Note that the expressions above *by themselves* are not valid C++ statements - they would be *part* of a statement. But each expression (if it is *in* a valid statement) has a *value*.

Also - the spaces written between operands and operators are not required by C++, but do improve legibility.
More complex expressions are evaluated by C++ using **Rules of Precedence** (like algebra)

1. sub-expressions in ()
2. unary -
3. * and / - left to right
4. + and - - left to right

In the examples below, the red numbers show what operation is done as C++ evaluates and expression:

So: 3 + 5 * 4 / 2 =
3 + 20 / 2 =
3 + 10 =
13

But () can be used to change this (or to make expression more readable)

(3 + 5) * 4 / 2 =
8 * 4 / 2 =
32 / 2 =
16

Example 1:

Suppose you accumulate 5 int test scores in *sum*.
And suppose that *sum* is 104.

Then to write an expression that is the average, you could write:

sum/5

and the value of that expression is 20 (a bit wrong - we lost the decimal fraction .8)

but if you write:

sum/5.0

then the value of the expression is 20.8 (correct)

Example 2:

Suppose you don't know how many numbers there are when you write the program. So you can't use 5 or 5.0 because there might be 21 or 3 or 97 numbers. You could use a variable to hold the count of numbers: declare *int count* and write some instructions that will give it the proper value before you want to calculate the average. We'll see how to do this soon.

You can't write "Count.0" to force it to be real (like we wrote 5.0 above). C++ simply does not allow this.

You must use a type cast - to tell C++ that it should *temporarily* use the variable as if it were some other data type. To do this, put the name of the data type you want in parenthesis before the variable name. It will not change the variable type or value permanently, but it will force C++ to *temporarily* treat the variable as a different type:

(double) sum/count   // or

sum /(double)count

(It is true that we could avoid the need to type cast here by declaring count to be double instead of int. But there are other situations where you will either want or need to do this.)

You cannot do just *any* type cast: *(char) sum* would not make sense and would not compile.

3.6 Assignment Statement/Operation

The symbol "=" is a **command**.

It means: "*evaluate* the **expression** on the right and *store* this **value** in the **variable** on the left"

Memorize that sentence.

In general:

```
someVariable = some expression;
```

Examples:

```
x = y * 3;
x = x + y;
```

Notice that there is *always exactly one variable on the left* to receive the value of the expression on the left.

This is **NOT** like algebra; it is **NOT** an equation.

Algebra: x = 1   (T or F for a given x)

C++:   x = 1; // store 1 into x

Algebra: x = x + 1 (false for all x)

C++:   x = x + 1; //find x+1; result in x

More Assignment Examples

```
//-- declare vars
double dAns;
int iAns;
int i, j;
double x, y;
 //-- assign values via numeric literals
i = 5;
j = 8;
x = 4.0;
y = 1.5;
 //-- assign values from complex expressions
i = i + 2; // 7
dAns = x / y;     // 2.666..
iAns = j/2*2;     // 8/2*2 = 4*2 = 8
iAns = 8/5;       // 1
iAns = 8 % 5;     // 3
iAns = 5 / 8;     // 0
iAns = 5 % 8;     // 5
i = x/y;   // 2
```

The last needs an explanation. The value of the expression *x/y* is 2.666.. but *i* is an *int* variable and so cannot hold the .666 part. C++ simply drops it (truncates it) and so only the whole number part of the value is stored into *i.*

Notes:
1. All variables in expressions on the right must have defined values or you get random results. This is an example of what I meant before when I said variables must have values before they are used.

```
int x, y;       // declared; no init values
x = y + 1;      // ??? in x
int x;          // declared; no init value
int y = 2;      // declared; given a value
x = y + 1;      // x has value 3.
                // It is Ok if x has no value before this
```

Inadvertant use of uninitialized variables is one of the most common causes of program errors. Learn to be very careful about this.

2. You can do multiple assignments on one line:

```
x = y = z = 3;  //all get value 3
```

3. Naming Variables: use meaningful names that explain themselves to reader of the program. (You, me, the maintainer of the program after you go to your next job...)

No	Yes		
s	score		
cs	class_size	ClassSize	classSize
cav	class_average	ClassAvg	classAvg
num	student_count	NumStudents	numStudents
stnum	student_id	StudentID	studentID

3.7 Input

We will use the standard C++ library input and output features.
For now we will use *cin* for input. *cin*'s task is to accept user input and store it into variables.

Note that

- the C++ *language* itself has no defined I/O functions or commands; just a *standard library* that can accomplish I/O. We could use other libraries, including those from the C language or those developed by others.
- most often when we want user input, we will need to do two things:
    1. tell the user what to enter and then
    2. get the value entered.

So typically we will do something like this:

```
int i;                          // a.
...
cout << "Enter the temperature: ";     // b.
cin >> i;                        // c.
```

Notes:

- a. we declare an int to hold the value to be typed in
- b. we tell (*prompt*) the user to enter something
- c. we get the value that is entered by the user
- when a *cout* occurs in a program, the characters are displayed immediately on the screen
- when *cin* occurs in a program, the program ***waits*** for user to type in a value and press ***<Enter>***.
- the << and >> are meant to suggest the direction the data is going:
  - for *cout*, the prompt is going from the string constant through *cout* to the screen
  - for *cin* the data entered is coming in from the keyboard, through *cin*, to the variable i
- for numeric input with *cin*, if the user types a non-number like "w" or "23e" then results are unpredictable. (For a double or float, "2e5" is ok - it's the same as 200000)
- the variable receiving the value from *cin* must be declared (of course) but need not have an initial value. Any previous value in that variable is lost.
- details about formatting numeric output (number of decimal places, etc.) will be covered below.

cin "understands" the different data types we have covered. So you can do *cin* with int, float, double, char, and string. But you **must** ensure that the data item getting the value is compatible with what the user is asked to enter. You can't store "jim" into an int, for example. You can store the value 4 into a double, however.

3.8 Output

We will use *cout* for output. *cout*'s task is to display information on the screen.

As shown above, the simplest form of *cout* is:

```
cout << "Hello";
```

You can output (for display) several values, *including the values calculated and stored into variables*, by stringing them together connected by "<<":

```
cout << "The average is: " << avg;    // avg is a variable

cout << "The temperature is: "
    << temp
    << " and the rainfall today was: "
    << rainAmt;
```

Note that I have put each item to display on a separate line for clarity. This is both legal and is usually desirable if you have more than 2 values.

There are a couple of special techniques to position cursor before or after you display characters, using an **escape sequence**. (A \ followed by a char. For example, \n is the *newline* escape sequence.) Note that this is a **back**slash.

```
cout << "\nhello";    //moves to next line, shows hello
cout << "hello\n";    //shows hello, then goes to new line
cout << "hel\nlo";    //shows hel, then lo on next line
```

Note: to *display* a single \, you must code two: \\

```
cout << "the \\ char is special";
```

This shows: "the \ char is special"

Also, there is an alternate way to move the cursor to the beginning of a new line: the special value "endl":

```
cout << endl << "Some stuff" << endl;
cout << "\nSome stuff\n";
```

Both will first move the cursor to a new line, then display "Some stuff" and then move the cursor to the next line.

Note that if you don't include endl or \n, all your output will be on one line:

```
cout << "The average is: " << avg;       // avg is a variable
cout << "The temperature is: "
    << temp
    << " and the rainfall today was: "     << rainAmt;
```

This will display something like this:

The average is 77.5The temperature is: 78 and the rainfall today was: 1.32362

Self-Assessment Exercise(s)

A. A program to enter two numbers and find their average
B. Program to enter the sale value and print the agent's commission

Self-Assessment Answer(s)

A. A program to enter two numbers and find their average

```
#include <iostream.h>
#include <iostream.h>
#include <conio.h>

void main()
{
clrscr();
int x,y,sum;
float average;
cout << "Enter 2 integers : " << endl;
cin>>x>>y;
sum=x+y;
average=sum/2;
cout << "The sum of " << x << " and " << y << " is " << sum << "." << endl;
cout << "The average of " << x <<  " and " << y << " is " << average << "." <<
endl;
getch();
}
```

This program takes in two integers x and y as a screen input from the user.
The sum and average of these two integers are calculated and outputted using the 'cout' command.

Sample: if input is 8 and 6
Output will be **The sum of 8 and 6 is 14.**
The average of 8 and 6 is 7.

Program to enter the sale value and print the agent's commission

```
#include <iostream.h>
#include <conio.h>

void main()
{
clrscr();
long int svalue;
float commission;
cout << "Enter the total sale value : " << endl;
cin>>svalue;
if(svalue<=10000)
{
commission=svalue*5/100;
cout << "For a total sale value of $" << svalue << ", ";
cout << "the agent's commission is $" << commission;
}
else if(svalue<=25000)
{
commission=svalue*10/100;
cout << "For a total sale value of $" << svalue << ", ";
cout << "the agent's commission is $" << commission;
}
else if(svalue>25000)
{
commission=svalue*20/100;
cout << "For a total sale value of $" << svalue << ", ";
cout << "the agent's commission is $" << commission;
}
getch();
}
```

This program takes in the total sale value as a screen input from the user.
The program then calculates the agent's commission with the help of the 'IF-ELSE' command as follows :

5% if the total sale value is less than or equal to $10000.
10% if the total sale value is less than or equal to $25000.
20% if the total sale value is greater than $25000.

It then outputs the agent's commission using the 'cout' command.

4.0 Summary/Conclusion

- A **computer program** (also **software**, or just a **program**) is a sequence of instructions written to perform a specified task with a computer
- Various data types in programming include integer, float, char, double etc.
- Programming languages include C, C++, Java etc
- C++ is a *superset* of the C programming language
- It is made up of keywords/reserved words.
- Header declaration is important for a programme to work
- A program should be written with the output insight.

5.0 Tutor-Marked Assignments

1. Write a Program to find if a certain number is odd or even
2. Write a program to find the area of a circle
3. Write a program to find the simple interest

6.0 References/Further readings

http://www.Glearn.net/programming simplified

http://tecfa.unige.ch/moo/book2/node74.html

Module 6

Unit 1

Client/Server Computing

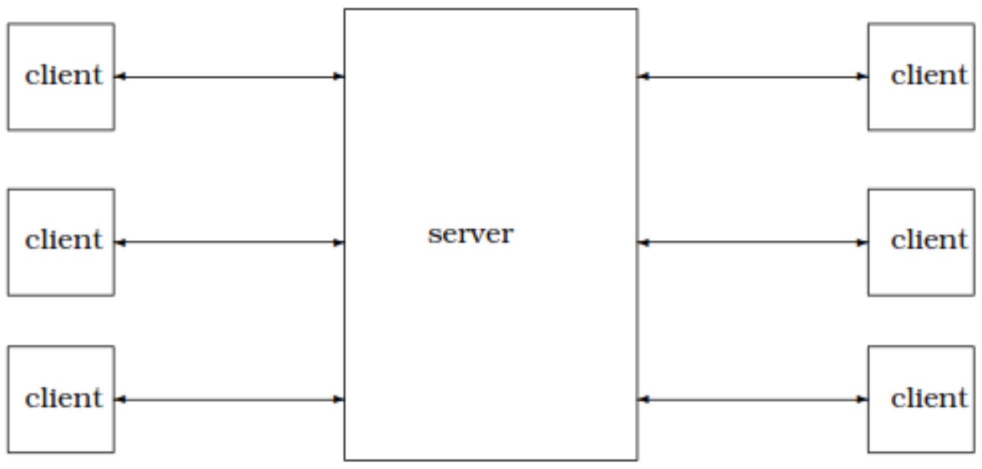
Contents

1.0 Introduction

A modern computing environment consists of not just one computer, but several. When designing such an arrangement of computers it might at first seem that the fairest approach is to make all of the individual computers in the collection equally powerful, with processors of equal speed and the same amount of memory. In this way it would seem that the users of this computer system would be equally well resourced and that the resources are shared out in the fairest possible way.

In practice, such an equal distribution of resources typically does not provide the best service to users. It is frequently the case that users will have occasional tasks which are more time-consuming or memory-hungry than others. Tasks such as these will then be delayed, possibly at a time when many of the other available computers are under-used.

An alternative distribution of resources which helps with this problem is to buy at least one machine which is more powerful than then rest (with a faster processor and with more memory) and have the other machines arranged so that users may connect to the more powerful machine when they need to. With this arrangement the users of the system are provided with occasional access to a more powerful computer than they otherwise would have had. The more powerful machine is called the server and the less powerful machines used to connect to the server are called the clients.



The client/server design provides users with a means to issue commands which are sent across a network to be received by a server which executes their commands for them. The results are then sent back to the client machine which sent the request in order that the user may see the results. Sending the request and receiving over the network will take time but this expense is often offset by the fact that the server will be able to execute the command faster than the client itself could do, if it would even be able to do it at all, given its lesser memory resources.

In order for client and server to communicate successfully, they must do so according to a set of rules which regulate the form which the communication must take. This set of rules is called a protocol and the communications protocol which is most widely used today is TCP/IP (Transport Control Protocol/Internet Protocol). (Another is UDP, the Unreliable Datagram Protocol). TCP/IP provides an addressing mechanism which allows clients to set up connections with servers. The addressing system makes use of host names and port numbers.

2.0 Learning Outcome:

In this unit the following will be learnt:

a. About Client/Server Computing
b. About various types of servers

3.0 Learning Content

3.1 Hosts and ports

A host is a machine which is connected to an IP network. A host can be identified by a host name (such as scar.inf.ed.ac.uk) which translates into a numeric IP address (such as 129.215.216.18). The textual form of the name is much easier to remember than the numeric address and can be translated into the numeric address using a service called the Domain Name Service (DNS).

A port number is used to communicate with a process which is running on a host.

Particular services provided by a host will be associated with a particular port number. For example, port 25 is used for electronic mail and port 80 is used for Web communication using HTTP. On UNIX systems (including Linux) the meanings of the assigned ports can be found by examining the file /etc/services.

Together, a host name (or IP address) and a port number uniquely identify a particular process running on a particular machine on the network. An analogy is that the IP address is rather like a telephone number, connecting you to a particular location, and the port number is rather like a telephone extension number, connecting you to a particular telephone receiver at that location. Together a host name and a port number allow us to create a socket.

3.2 Sockets

A socket is a connection to another machine. A server will listen for connections on a particular port. When a client tries to connect to the same port a socket connection is established. The socket behaves as two pairs of an input stream and an output stream. Viewed from the server side, the input stream is read to get commands from the client and the output stream is used to write results back to the client. Viewed from the client side, the output stream is used to write commands to the server and the input stream is used to read the results from the server. Of course these pairs of streams match up to make a bi-directional communication channel between client and server.

Keeping in mind that sockets are made up of input and output streams is useful because we are always aware that input and output operations can fail. Communication between hosts can fail also, perhaps because we do not have permission to connect to a particular host or because we do not have permission to use a particular port.

3.3 Client/Server computing in Java

Having met the concepts of client/server computing, we now progress to discovering how to implement these systems in Java. We will make use of classes from the java.io package, as we might expect, but we will also make use of classes from Java's networking package java.net. As an example here we will consider constructing a simple messaging service, allowing the client to send messages to be printed on the console of the server. A reply is sent back to acknowledge that the message has been received.

The system is structured as two separate Java programs, each with a main() method which is invoked from the Java interpreter, initiated with the java command. Thus there are two separate instances of a Java virtual machine executing at the same time. One runs on the server. The other runs on the client machine. No memory is shared between the client and the server and so information cannot be communicated via shared variables which both the client and the server can access. The only option is that information must be sent as 'messages'. This method of communication between co-operating programs is known as message passing.

Starting the server

We begin by initiating the Server program on the server, which for the sake of definiteness; we will take to be ssh.inf.ed.ac.uk, using port 5055, which is not needed for any other purpose.

```
[ssh]stg: java Server
 Listening on port 5055
```

Starting the client

At this point we can run the client program on another machine (perhaps using the machine bleaurgh.inf.ed.ac.uk). The Client program assumes that it is communicating with ssh on port 5055 but it could be easily made more general by accepting the host name and port number as command line arguments. After an initial pause while connection is established we eventually receive acknowledgements to our messages as shown below. Simultaneously, the messages are received by the server.

```
[ssh]stg: java Server

[bleaurgh]stg: java Client

Listening on port 5055 Message received

First message sent Message received

Second message sent
```

The program could easily be turned into a simple talk application, allowing sentences typed at one side to be sent to the other and replies to be sent back similarly.

3.4 The Server Program

The following program is run on the server. Lines 7 to 13 try to create a server socket, and exit the program with an explanatory message if this socket could not be created. Lines 15 to 21 establish a client connection to this socket, where an error will again cause the server to exit with an explanatory error message. In lines 31 to 36 lines of text are read repeatedly and acknowledged until a null String object is received.

```
import java.io.*;
import java.net.*;

class Server {
   public static void main (String[] args)
         throws IOException {
      ServerSocket sock = null;
      try {
         sock = new ServerSocket(5055);
      } catch (IOException e) {
         System.out.println ("Could not listen on port "+ e);
         System.exit(1);    // Exit the program
      }
      System.out.println ("Listening on port 5055");
      // Try to accept a connection from a client
      Socket clientSocket = null;
      try {
         clientSocket = sock.accept();
      } catch (IOException e) {
         System.out.println
               ("Accept failed: " + e);
         // Exit the program
         System.exit(1);
      }
      // Communication has been established
      InputStreamReader is = new InputStreamReader
         (clientSocket.getInputStream());
      BufferedReader input = new BufferedReader(is);
      PrintWriter client = new PrintWriter
         (clientSocket.getOutputStream());
      String line = input.readLine();
      while (line != null) {
         System.out.println ("Client said: " + line);
         client.println ("Message received");
         client.flush();
         line = input.readLine();
      }
   }
}
```

3.5 The Client Program

The following program is run on the client. It sends only two message strings to the server, which it assumes to be running on ssh.inf.ed.ac.uk, listening on port 5055. Lines 9 to 11 establish a buffered input stream which allows replies from the server to be read by invoking the readLine() method of the input object. This object is of the Buffered Reader class and has been created by getting the input stream object from the socket. This object, which is seen as the input stream from the client side, is seen as the output stream from the server side. The strings which are returned as the result of the readLine () method invocations are printed on the console of the client machine by using the println() method provided by the System. out object.

```
import java.io.*;
import java.net.*;

class Client {
   public static void main (String[] args)
           throws IOException {
       Socket sock =
           new Socket ("localhost", 5055);
       // Communication has been established
       InputStreamReader is = new InputStreamReader
          (sock.getInputStream());
       BufferedReader input = new BufferedReader(is);
       PrintWriter server = new PrintWriter
          (sock.getOutputStream());
       server.println("First message sent");
       server.flush();
       System.out.println("Server replied: " +
            input.readLine());
       server.println("Second message sent");
       server.flush();
       System.out.println("Server replied: " +
            input.readLine());
   }
}
```

Careful thought is required to ensure that a pair of communicating programs such as these will work together. The messages written by one side need to be read by the other for the communication to succeed. One the server side, the program first reads a message and then writes an acknowledgement. On the client side, the program first writes a message and then reads an acknowledgement. Symmetric communication such as this will succeed, but it is easy to miss-match messages, in which case both parties can be left waiting to read a message sent by the other. Such deadlocks are often prevented by limiting the time spent waiting for a communication to be received. This technique also deals with the case that one or other machine suffers a software or hardware crash. A planned interrupt like this is called a time-out.

3.6 Communicating Objects

Message passing is a simple form of communication between programs which run on different computers. In particular, it does not exploit the fact that both of the programs which are used here are Java programs, with a common means of representing objects in memory. Because of this, we cannot communicate by message passing any objects more complex than strings. Java does provide a means of communicating general objects between instances of Java virtual machines running on different hosts where the objects to be communicated are serialized on the sender's side and sent across the network to be de-serialized on the receiving side. This form of communication is called remote method invocation (RMI).

3.7 Server Structures

Web Server

A web server is a specialized type of file server. It is a piece of software that enables a website to be viewed using HTTP. HTTP (Hyper Text Transfer Protocol) is the key protocol for the transfer of data on the web, for example when using HTTP, the website URL begins with "http://" (for example, "http://www.futminna.edu.ng"). Its job is to retrieve files from the server's hard drive, format the files for the Web browser (for example internet explorer, Mozilla etc), and send them out via the network.

Servers are designed to do a great job of sending content out to a large number of users. The pages delivered by the server are expected to be the same for everyone who visits the server.

Characteristics of Web Servers

- Create one or more websites. (No I don't mean build a set of web pages. What I mean is, set up the website in the web server, so that the website can be viewed via HTTP)
- Configure log file settings, including where the log files are saved, what data to include on the log files etc. (Log files can be used to analyze traffic etc)
- Configure website/directory security. For example, which user accounts are/aren't allowed to view the website, which IP addresses are/aren't allowed to view the website etc.
- Create an FTP site. An FTP site allows users to transfer files to and from the site.
- Create virtual directories, and map them to physical directories
- Configure/nominate custom error pages. This allows you to build and display user friendly error messages on your website. For example, you can specify which page is displayed when a user tries to access a page that doesn't exist (i.e. a "404 error").
- Specify default documents. Default documents are those that are displayed when no file name is specified. For example, if you open "http://localhost", which file should be displayed? This is typically "index.html" or similar but it doesn't need to be. You could nominate "index.cfm" if your website is using ColdFusion. You could also nominate a 2nd choice (in case there is no index.cfm file), and a 3rd choice, and so on.

3.8 Tools for Client Server Development

Choosing client server development tools can be a complicated issue, involving factors such as the intended audience of the program, the program's intended use, the operating system, and the programming language you want to use. Cost is also a factor, but fortunately there are many integrated development environments (IDE) provided for free by companies such as Microsoft, Apple and Oracle. The following are list of development tools:

Visual Studio 2010 Express

- Microsoft's Visual Studio Express series is both free and a complete IDE. There are different versions available depending on the type of development you're working on and all are available with SQL Server Express for database development. The available languages and technologies include ASP.NET, Visual Basic, Visual C#, Visual C++, .NET 3.5/4, and Silverlight 3/4. Possible target platforms include web applications, internal client server applications and Windows Phone applications. The Windows Phone version also includes a simulator to test and debug applications during the development process.

Oracle Solaris Studio

- Oracle Solaris Studio is another free IDE provided by a major software manufacturer. It runs on both Solaris and Linux, and supports development in C, C++ and FORTRAN. Not surprisingly, this IDE integrates well in systems and client server applications that make use of Oracle databases and is compatible with SPARC and x86 based systems.

Oracle JDeveloper

- Oracle JDeveloper is also freely available, but is focused toward Java development. It can be installed and run on Windows, Linux and Mac, and supports integration with Oracle, MySQL, JDK and GlassFish.

NetBeans IDE

- NetBeans IDE makes use of a plugin system to provide for a wide variety of development languages and needs, all in a free IDE. The IDE itself is compatible with Windows, Mac, Linux and Solaris on x86 and SPARC systems. It's compatible with web and database servers such as MySQL, PostgreSQL, Oracle, GlassFish Server, JDK, and Apache Tomcat. Available languages include Java, C, C++, Ruby, Groovy and PHP. There is also a plugin portal where users can submit and download community created plugins.

Xcode IDE

Xcode IDE is Apple's product for the free IDE market. It's a complete IDE that can be used to create stand-alone, client/server and iOS apps. It only runs on Mac OS X and supports C, C++ and Objective-C. The IDE also integrates with the development tools included with OS X, which include support for Java, AppleScript, Perl, Python, and Ruby. Xcode includes an iPhone simulator that can be used to test applications, it even tests touch gestures and phone rotation response

How a Web server Works

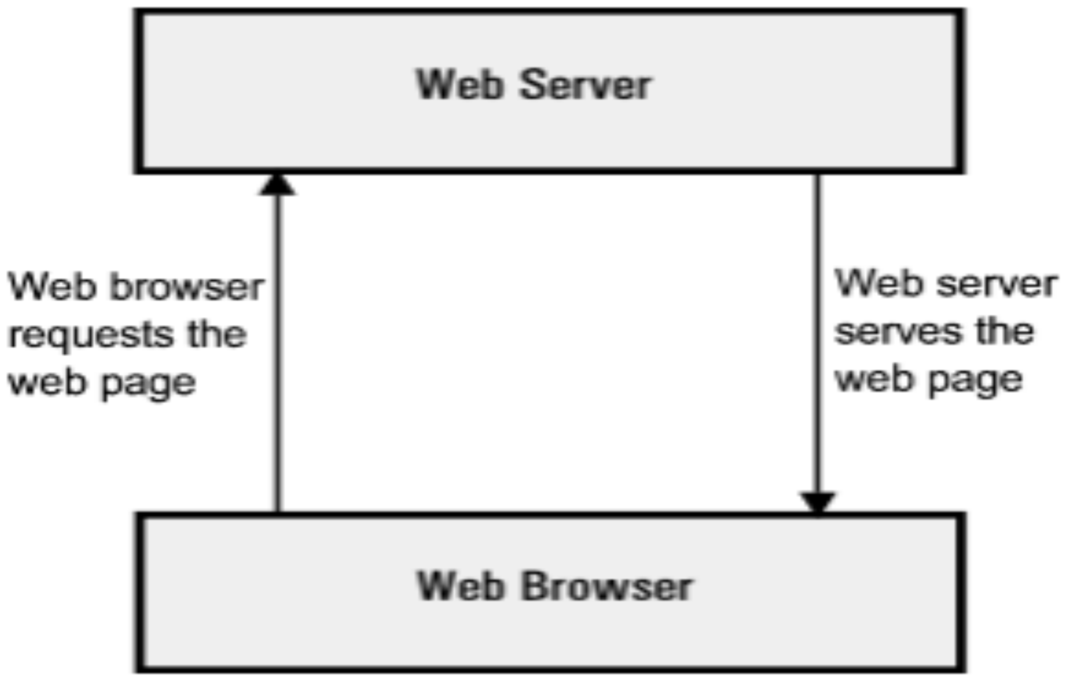

Web Server
Web browser requests the web page
Web server serves the web page
Web Browser

Figure 11.9 Simplistic Structure of a Web Server

The above diagram is a simplistic version of what occurs. The details are explained below:

1. Your web browser first needs to know which IP address the website "www.fut.com" resolves to. If it doesn't already have this information stored in its cache, it requests the information from one or more DNS servers (via the internet). The DNS server tells the browser which IP address the website is located at. Note that the IP address was assigned when the website was first created on the web server.
2. Now that the web browser knows which IP address the website is located at, it can request the full URL from the web server.
3. The web server responds by sending back the requested page. If the page doesn't exist (or another error occurs), it will send back the appropriate error message.
4. Your web browser receives the page and renders it as required.

   When referring to web browsers and web servers in this manner, we usually refer to them as a *client* (web browser) and a *server* (web server).

   Examples of Web Servers are: Apache HTTP Server, Sun Java System Web Server.

3.9 Domain Name System Server

The Domain Name System (DNS) is a standard technology for managing the names of Web sites and other Internet domains. DNS technology allows you to type names into your Web browser like *fut.about.com* and your computer, automatically finds that address on the Internet. A key element of the DNS is a worldwide collection of *DNS servers*. **What, then, is a DNS server**?

A **DNS server** is any computer registered to join the Domain Name System. A DNS server runs special-purpose networking software, features a public IP address, and contains a database of network names and addresses for other Internet hosts.

DNS servers communicate with each other using private network protocols. All DNS servers are organized in a hierarchy. At the top level of the hierarchy, is a ***root server.*** This stores the complete database of Internet domain names and their corresponding IP addresses. The Internet may employ root servers that have become somewhat famous for their special role. Maintained by various independent agencies, these servers may reside in the United States, Japan, UK, Africa and so on.

How a DNS Server Works

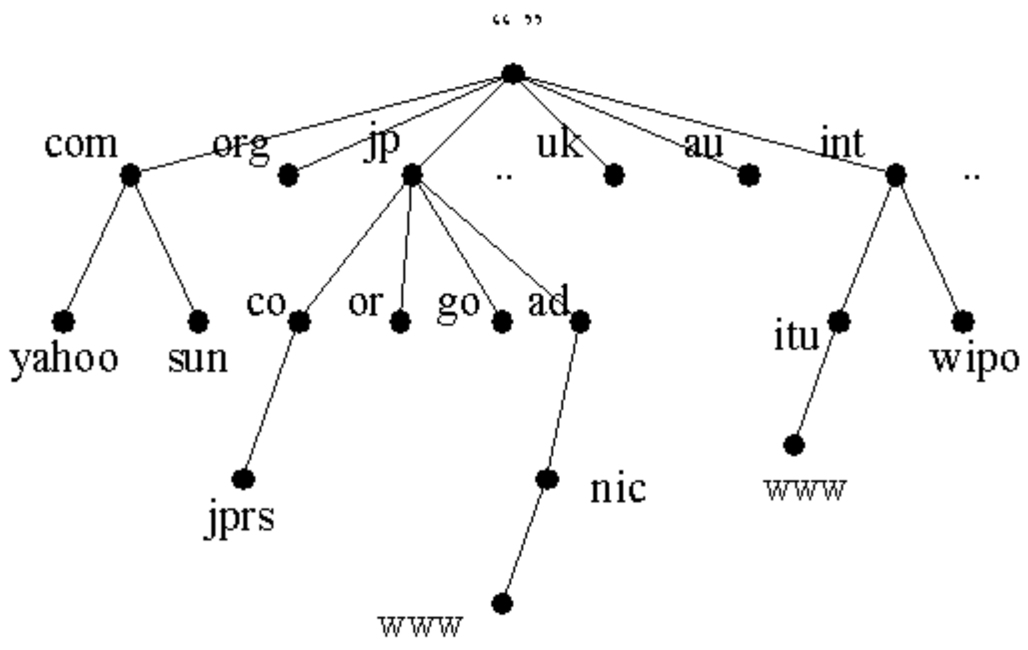


Figure 11.10 DNS Server Structure

The Figure above shows how an entity named by a domain name is identified on the Internet. Each node of the DNS structure can be considered as a table, called a name server, maintaining pairs of the node labels directly underneath the node and the corresponding IP addresses. Name servers correspond to organizations or units that are *authoritative* to manage the domain name corresponding to the node. For example, the root server is the authoritative source for the .int or .com names; the name servers for .int are the authoritative source for the.itu.int and wipo.int names, and the name servers for .itu.int are authoritative for www.itu.int. The DNS is therefore, in effect, a large globally distributed database from both an engineering and management viewpoint.

3.10 Terminal Servers

A Terminal server (also referred to as a serial server or network access server) enables organizations to connect devices with serial interface(Such as :RS-232, RS-422 or RS-485) to a local area network (LAN). Products marketed as terminal servers can be very simple devices that do not offer any security functionality, such as data encryption and user authentication.

The primary application scenario is to enable serial devices to access network server applications, or vice versa, where security of the data on the LAN is not generally an issue. There are also many terminal servers on the market that have highly advanced security functionality to ensure that only qualified personnel can access various servers and that any data that is transmitted across the LAN, or over the Internet, is encrypted. Usually companies which need a terminal server with these advanced functions want to remotely control, monitor, diagnose and troubleshoot equipment over a telecommunications network.

How Terminal Server Architecture Works

The goal of this section is to provide a thorough understanding of the underlying structure of Terminal Server. A thorough understanding of the operating system's operation at this level will help to better troubleshoot Terminal Server and optimize its performance.

Below is the Terminal Server Structure diagram as shown in Figure 15.9, with explanation on the function of each part of the diagram.

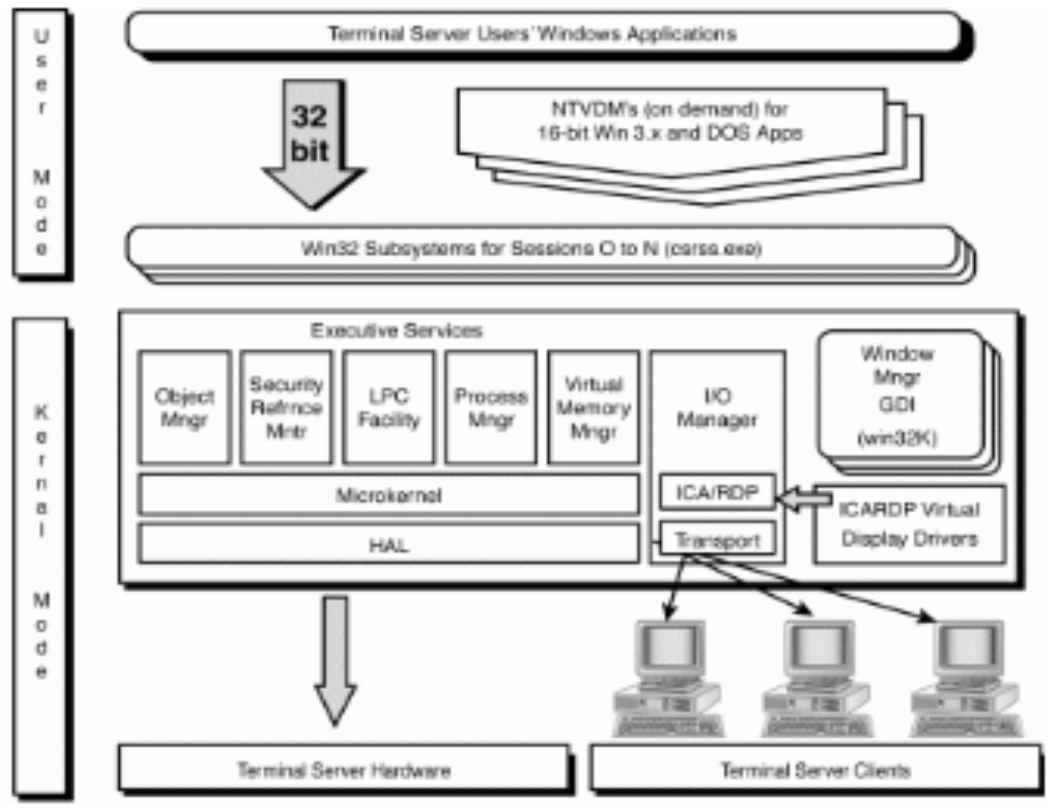


Figure 11.11 Terminal Server Structure

- ***Kernel mode*** in kernel mode, an application has direct access to the hardware. Examples of kernel mode applications are the HAL, Microkernel, Virtual Memory Manager, and device drivers. Because the applications are running in kernel mode, the system is not protected from them, and as a result, they must be written and tested very carefully. One poorly written application can lock up or crash the entire system.
- ***User mode*** in user mode, an application can access hardware and operating system resources only through the Win 32 subsystem. This layer of isolation provides a great amount of protection and portability for applications. Examples of applications running in the user mode are DOS applications, Windows 3.x (16-bit), Windows 32, POSIX, and OS/2. If a user mode application crashes, the crash affects only the process the application is running in. By using Task Manager, you can simply delete the frozen process. All other applications continue to run without interruption.

The underlying difference between kernel and user mode has to do with the processor privilege level. Kernel mode is often referred to as privileged mode. On most modern processors, including Intel and Alpha, you can run an application at different privilege levels. On the Intel processors, there are four privilege levels or rings in which applications can run: 0–3. In reality, kernel mode refers to programs that run at privilege level 0. These programs have unrestricted access to all hardware and memory in the system. Programs running at privilege level 0 can set the privilege level for programs that they call. In contrast, user mode refers to applications and subsystems that are set by the kernel to run in privilege level 3. In this mode, applications are "boxed-in." The processor allows them to work only within their own memory areas. The processor also restricts them from directly accessing I/O, such as disk drives, communications ports, and video and network cards. To access these devices, applications must make a special gated call to a kernel mode program that handles the hardware.

3.11 Database Servers

A Database server is a computer program that provides database services to other computer programs or computers, as defined by the client–server model. The term may also refer to a computer dedicated to running such a program. Database management systems frequently provide database server functionality, and some DBMSs (e.g., MySQL) rely exclusively on the client–server model for database access. Such a server is accessed either through a "front end" running on the user’s computer which displays requested data or the "back end" which runs on the server and handles tasks such as data analysis and storage. In a master-slave model, database master servers are central and primary locations of data while database slave servers are synchronized backups of the master acting as proxies. Some examples of proprietary database servers are Oracle, DB2, Informix, and Microsoft SQL Server. Examples of GNU General Public License database servers are Ingres and MySQL. Every server uses its own query logic and structure. The SQL query language is more or less the same in all the database servers.

How Database Server Works

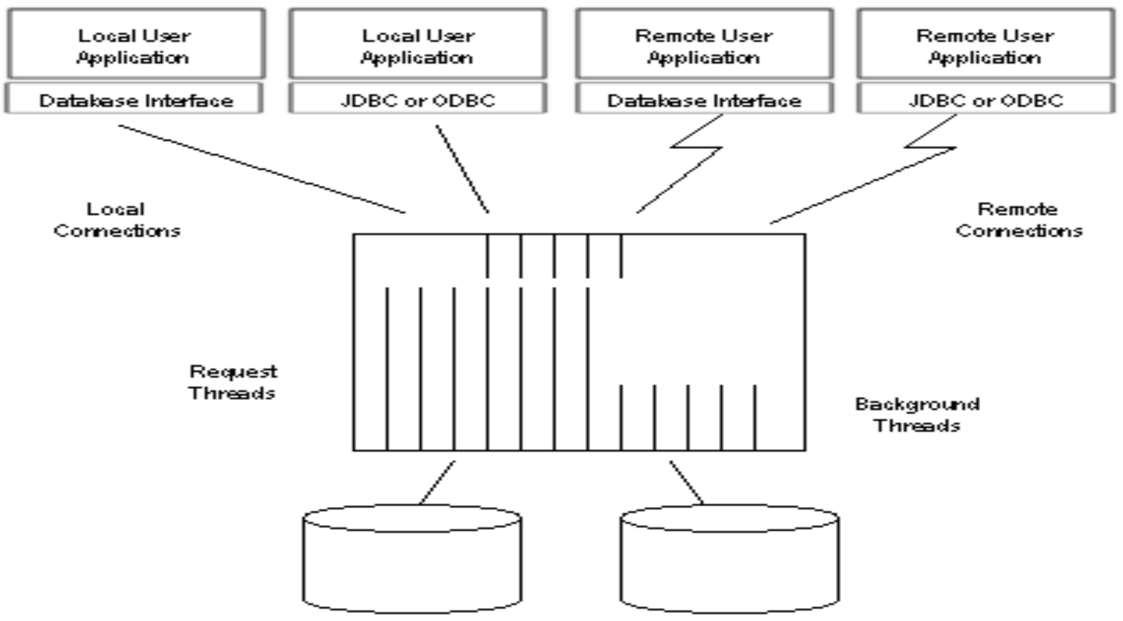


Figure 11.12 Structure of database Servers

With reference to the figure above, the **Communication threads** are used to handle parts of the communication between the applications and the database server. On some platforms other mechanisms are used to handle the communication between the applications and the database server. Whatever the mechanism, all communication with the database server is multi-threaded, allowing large numbers of simultaneous user requests.

Both local and remote applications are handled directly by the Database Server. This means that in **Client/Server** environments, where SQL executes in a distributed environment with the client and server on different machines, all remote clients connect directly to the Database Server. Thereby avoiding any additional overhead of network service processes being started, either on the client or on the server machine.

The **Request threads** perform the SQL operations requested by the applications. When the Database Server is requested to perform a SQL operation it allocates one of its Request threads to perform the task. When the SQL operation is complete the result is returned back to the application, and the Request thread that has performed the operation returns to a waiting state until it receives another server request. Since the SQL operations are evaluated entirely within the Database Server, inter-process communication is reduced to a minimum.

When a SQL query or a stored routine is executed by a Request thread, the compiled version of the query or the routine is stored within the Database Server. In this way the same, compiled version of the query or routine can be used again by other applications. This leads to improved performance, since a SQL query or a stored routine only need to be compiled once by the Database Server.

The **Background threads** perform database services including all database updates, online backup and database shadowing. These services are performed asynchronously in the background to the application processes, which means that the application process does not have to wait for the physical completion of a transaction or a shadow update, but can continue as soon as the transaction has been prepared and secured to disk.

I/O-operations are performed in parallel directly by the request and background threads using asynchronous I/O. Thereby any need for separate I/O-threads are avoided.

Examples of database servers are oracle, DB2, Microsoft SQL etc.

Self-Assessment Exercise(s)

What is the client?

What is the server?

Self-Assessment Answer(s)

A. What is the client?

The client is a process (program) that sends a message to a server process, requesting that the server perform a task (services).

B. What is the server?

A server process fulfills the client request by performing the task required. Server programs receive requests from client programs execute database retrieval and updates and dispatch responses to client requests

4.0 Summary/Conclusion

The *client/server* characteristic describes the relationship of cooperating programs in an application.

The server component provides a function or service to one or many clients, which initiate requests for such services

Functions such as email exchange, web access and database access are built on the client/server model. Users accessing banking services from their computer use a web browser client to send a request to a web server at a bank.

The interaction between client and server is often described using sequence diagrams

5.0 Tutor-Marked Assignments

1. Briefly explain Client /Server Computing?
2. What is a terminal Server and how does it work?

6.0 References/Further readings

http://101.lv/learn/access/aba20fi.htm

http://www.stratiss.com/clientsrvrapp.shtml

http://en.wikipedia.org/wiki/Client%E2%80%93server_model

Unit 2

Client/Server Strategies

Contents

1.0 Introduction

As you might have inferred from the previous chapter, it is very easy to implement client/server ineffectively. This can result in worse performance rather than better performance. The developer's task is to intelligently apply appropriate techniques that deploy client/server systems effectively.

The following sections discuss strategies to help you develop smart client/server applications.

2.0 Learning Outcome:

In this unit the following will be learnt:

a. About Client/Server Computing
b. About various types of servers

3.0 Learning Content

3.1 Selecting the Best Record set Type

Sometimes it is best to create a dynaset, and at other times it is more efficient to create a snapshot. It is very important that you understand under what circumstances each choice is the most appropriate.

In essence, a *dynaset* is a collection of bookmarks that enables each record on the server to be identified uniquely. Each bookmark corresponds to one record on the server and is generally equivalent to the primary key of the record. Because the bookmark is a direct pointer back to the original data, a dynaset is an updatable set of records. When you create a dynaset, you create a set of bookmarks of all rows that meet the query criteria. If you open a recordset using code, only the first bookmark is returned to the user's PC's memory. The remaining columns from the record are brought into memory only if they are directly referenced using code. This means that large fields, such as OLE and Memo, are not retrieved from the server unless they are explicitly accessed using code. Access uses the primary key to fetch the remainder of the columns. As the code moves from record to record in the dynaset, additional bookmarks and columns are retrieved from the server.

All the bookmarks are not retrieved unless a Move Last method is issued or each record in the record set is visited using code. Although this keyset method of data retrieval is relatively efficient, dynasets carry significant overhead associated with their ability to be editable. This is why snapshots are often more efficient.

A ***snapshot*** is a set of records returned from a query.

When a snapshot type is opened, of recordset, all columns from the first row are retrieved into memory. As you move to each row, all columns within the row are retrieved. If a MoveLast method is issued, all rows and all columns meeting the query criteria are immediately retrieved into the client machine's memory. Because a snapshot is not editable and maintains no link back to the server, it can be more efficient. This is generally true only for relatively small recordsets. The caveat lies in the fact that all rows and all columns in the result set are returned to the user's memory whether they are accessed or not. With a result set containing over 500 records, the fact that all columns are returned to the user's memory outweighs the benefits provided by a snapshot. In these cases, you may want to create a Read Only dynaset.

3.2 Forward Store Procedure

If your data does not need to be updated and it is sufficient to move forward through a recordset, you may want to use a forward-scrolling snapshot. Forward-scrolling snapshots are extremely fast and efficient. You create a forward-scrolling snapshot using the dbForwardOnly option of the OpenRecordset method. This renders the recordset forward-scrolling only. This means that you cannot issue a MovePrevious or MoveFirst method. You also cannot use a MoveLast. This is because only one record is retrieved at a time. There is no concept of a set of records, so Access cannot move to the last record. This method of data retrieval provides significantly better performance than regular snapshots with large recordsets.

3.3 Key Set Fetching

The fact that dynasets return a set of primary keys causes problems with forms. With a very large set of records and a large primary key, sending just the primary keys over the network wire can generate a huge volume of network traffic. When you open a form, Access retrieves just enough data to display on the form. It then continues to fetch the remainder of the primary keys satisfying the query criteria. Whenever keyboard input is sensed, the fetching process stops until idle time is available. It then continues to fetch the remainder of the primary keys. To prevent the huge volume of network traffic associated with this process, you must carefully limit the size of the dynasets that are returned. Methods of accomplishing this are covered in the section titled Optimizing Forms.

3.4 Utilizing Pass -Through Queries and Stored Procedures

It is important to remember that executing pass-through queries and stored procedures is much more efficient than returning a recordset to be processed by Access. The difference lies in where the processing occurs. With pass-through queries and stored procedures, all the processing is completed on the server. When operations are performed using VBA code, all the records that will be affected by the process must be returned to the user's memory, modified, and then returned to the server. This generates a significant amount of network traffic and slows down processing immensely.

3.5 Pre-connecting

In dealing with ODBC databases, connections to the server are transparently handled by Jet. When you issue a command, a connection is established with the server. When you finish an operation, Jet keeps the connection open in anticipation of the next operation. The amount of time that the connection is cached is determined by the ConnectionTimeout setting in the Windows Registry. You may want to utilize the fact that a connection is cached to connect to the back-end when your application first loads, before the first form or report even opens. The connection and authentication information will be cached and used when needed.

As seen in the LinkToSQL routine in chapter 20, you can send password information stored in variables as parameters when creating a link to a server. These values could easily have come from a login form. The following code preconnects to the server. It would generally be placed in the startup form for your application:

```
Sub PreConnect(strDBName As String, _
         strDataSetName As String, _
         strUserID As String, _

         strPassWord As String)
   Dim db As DATABASE
   Dim strConnectString As String
   strConnectString = "ODBC;DATABASE=" & strDBName & _
                ";DSN=" & strDataSetName & _
                ";UID=" & strUserID & _
                ";PWD=" & strPassWord
   Set db = OpenDatabase("", False, False, strConnectString)
   db.Close   'Closes the database but maintains the connection
End Sub
```

The trick here is that the connection and authentication information will be maintained even when the database is closed.

3.6 Reducing the Number of Connections

Some database servers are capable of running multiple queries on one connection. Other servers, such as Microsoft SQL Server, are capable of processing only one query per connection. You should try to limit the number of connections required by your application. Here are some ways that you can reduce the number of connections that your application requires.

Dynasets containing more than 100 records require two connections, one to fetch the key values from the server, and the other to fetch the data associated with the first 100 records. Therefore, try to limit query results to fewer than 100 records wherever possible.

If connections are at a premium, you should close connections that you are no longer using. This can be accomplished by moving to the last record in the result set or by running a Top 100 Percent query. Both of these techniques have dramatic negative effects on performance because all the records in the result set are fetched. Therefore, these techniques should be used only if reducing connections is more important that optimizing performance.

Finally, you might want to set a connection timeout. This means that if no action has been taken for a specified period of time, the connection will be closed. The default value for the connection timeout is 10 minutes. This value can be modified in the My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Access\7.0\Jet\3.0\Engines\ODBC key of the Windows Registry by changing the ConnectionTimeout setting. The timeout occurs even if a form is open. Fortunately, Access automatically reestablishes the connection when it is needed.

3.7 Optimizing Data Handling

One of the best things that you can do to optimize data handling—such as edits, inserts, and deletes—is to add a version field (timestamp) to each remote table. This version field is used when users update the data on the remote table to avoid overwrite conflicts. If this field does not exist, the server compares every field to see whether they have changed since the user first began editing the record. This is quite inefficient and is much slower than evaluating a timestamp.

The use of transactions is another way to improve performance significantly, because transactions enable multiple updates to be written as a single batch. As an added benefit, they protect your data by ensuring that everything has executed successfully before changes are committed to disk.

Optimizing Queries and Forms

On the whole, the movement to client/server improves performance. If you are not careful when designing your queries, forms, and reports, the movement to client/server can actually degrade performance. There are several things that you can do to ensure that the movement to client/server is beneficial. These techniques are broken down into query techniques, form techniques, and report techniques.

Optimizing Queries

Servers cannot perform many of the functions offered by the Access query builder. The functions that cannot be processed on the server are performed on the workstation. This often results in a large amount of data being sent over the network wire. This extra traffic can be eliminated if your queries are designed so that they can be processed solely by the server.

The following are examples of problem queries that cannot be performed on the server:

Top N% queries

- Queries containing user-defined or Access functions
- Queries that involve tables from two different data sources—for example, a query that joins tables from two different servers or from an Access table and a server table

Optimizing Forms

The following techniques can help you design forms that capitalize on the benefits of the client/server architecture. The idea is to design your forms so that they request the minimal amount of data from the server and that they obtain additional data only if requested by the user. This means that you request as few records and fields as possible from the server. This can be accomplished by basing forms on queries rather than directly on the tables. It can be further refined by designing your forms specifically with data retrieval in mind. For example, a form can initially be opened with no RecordSource. The form can require that users limit the criteria before any records are displayed.

You should store static tables, such as a state table, locally. This reduces network traffic and requests to the server. Furthermore, combo boxes and list boxes should not be based on server data. Whenever possible, the row source for combo boxes and list boxes should be based on local static tables. If this is not possible, you can use a text box in conjunction with a combo box. The Row Source of the combo box is initially left blank. The user must enter the first few characters into the text box. The Row Source of the combo box is then based on a Select statement using the characters entered into the text box.

Furthermore, OLE object and Memo fields are large and therefore significantly increase network traffic. It is best not to display the contents of these fields unless they are specifically requested by the user. This can be accomplished by setting the Visible property of OLE and memo fields to False, or by placing these fields on another page of the form. You can add a command button that enables the user to display the additional data when required.

The form shown in Figure 12.1 illustrates the implementation of several of these methods. The detail section of the form is initially not visible. The form has no RecordSource, and the data that underlies the combo box that appears on the form is stored in a local table. The After Update event of the combo box looks like this:

```
Private Sub cboBookType_AfterUpdate()
  Me.RecordSource = "Select * From dbo_titles Where Type Like '" & _
        cboBookType.Value & "*';"
  Me.Detail.Visible = True
End Sub
```

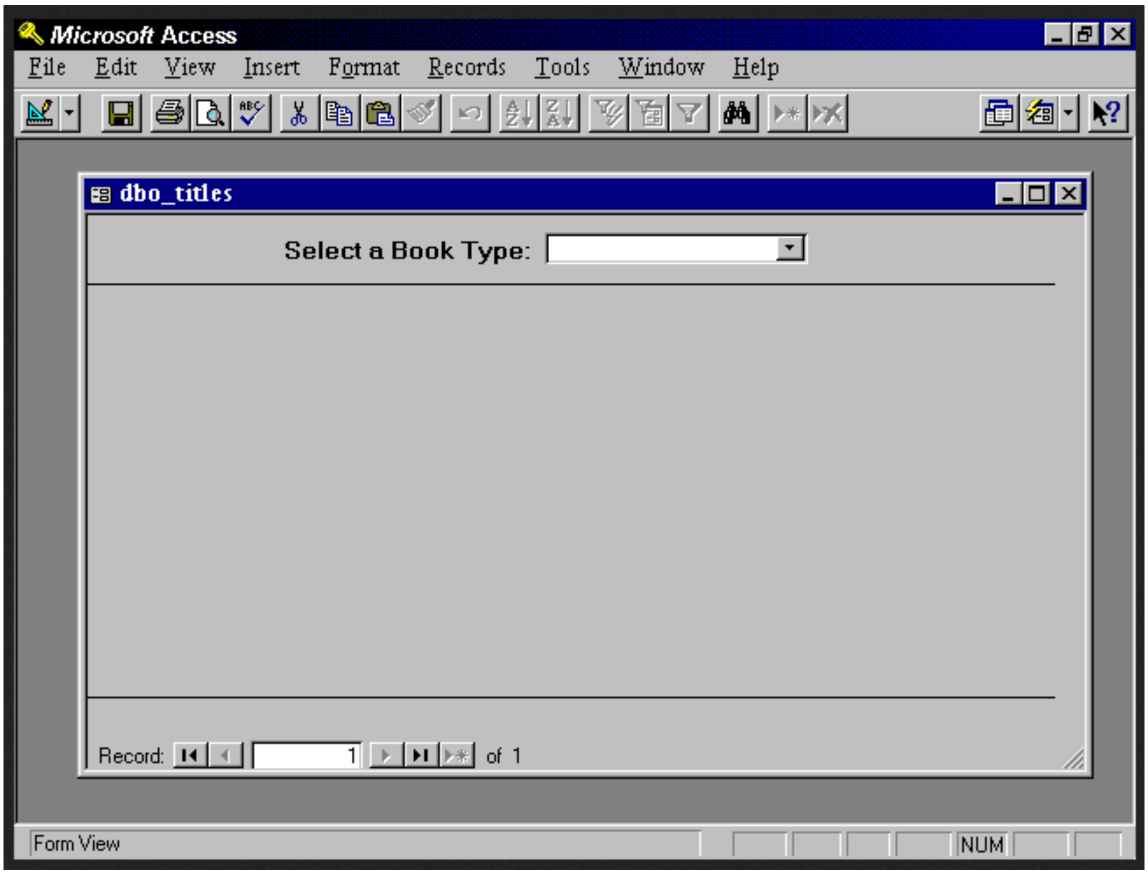
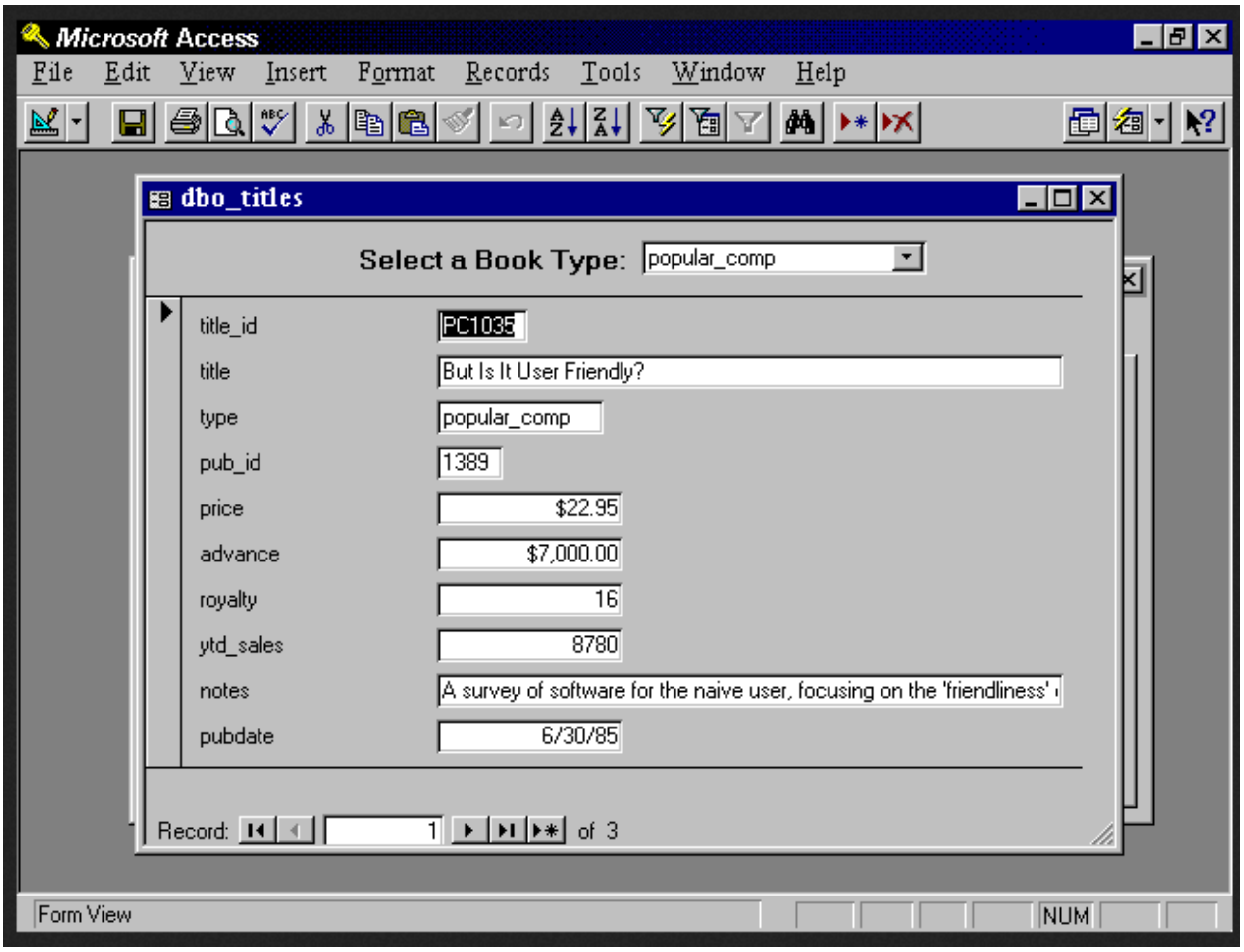


Figure 12.1. Form using After Update of combo box to populate RecordSource and make detail visible.

The Visible property of the detail section of the form is initially set to False. When the user selects an entry from the combo box, the RecordSource of the form is set equal to a Select statement, which selects specific titles from the dbo_titles table database. The Detail section of the form is then made visible (see Figure 12.2).



Select a Book Type: popular_comp

title_id	PC1035
title	But Is It User Friendly?
type	popular_comp
pub_id	1389
price	$22.95
advance	$7,000.00
royalty	16
ytd_sales	8780
notes	A survey of software for the naive user, focusing on the 'friendliness'
pubdate	6/30/85

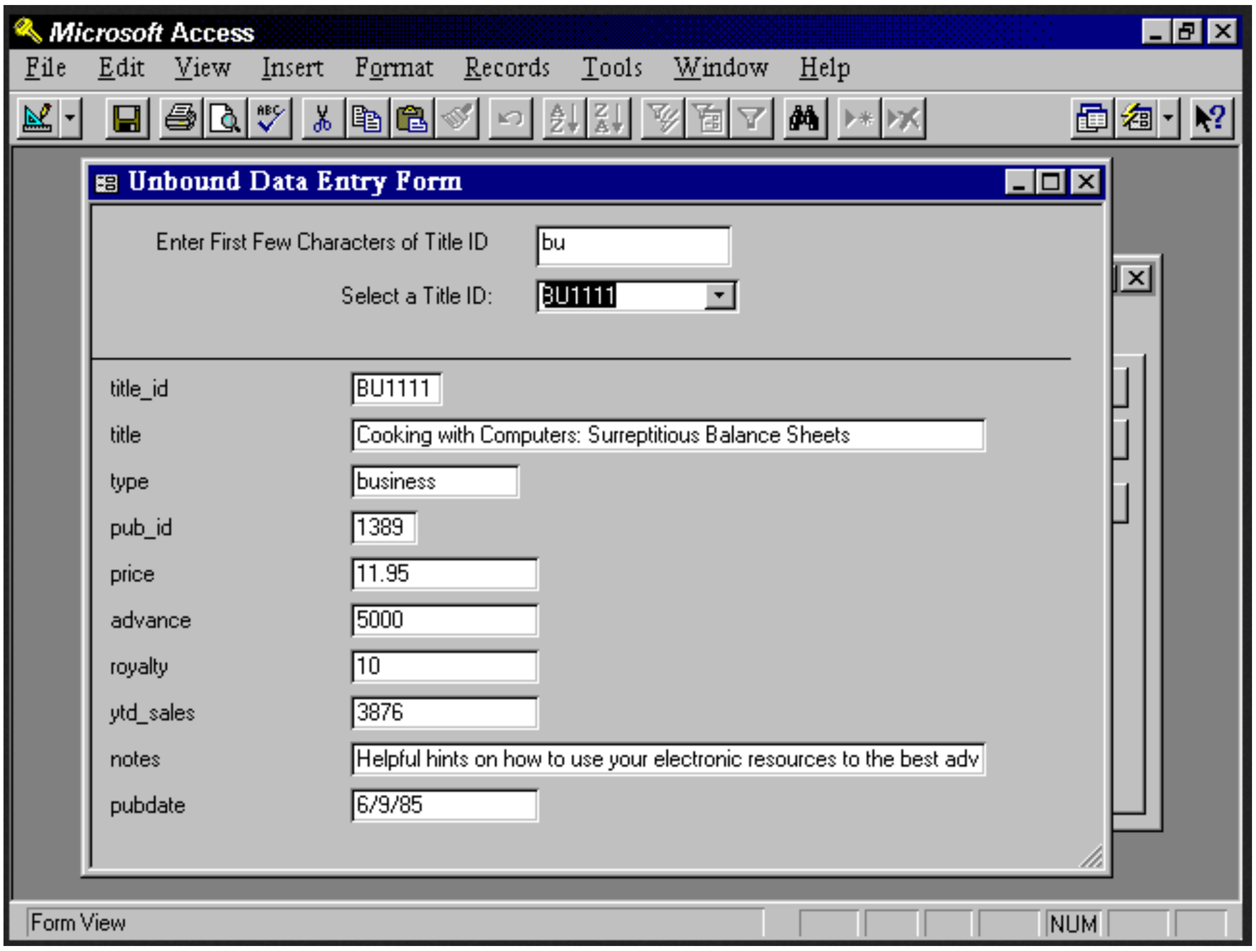
Record: 1 of 3

Form View NUM

Figure 12.2. Form using After Update of combo box to populate RecordSource with detail visible.

Finally, you may want to use unbound forms. This involves creating a form and then removing its RecordSource. Users are provided with a combo box that enables them to select one record. A recordset is built from the client/server data with the one row that the user selected. With this method of form design, everything needs to be coded. Your form needs to handle all adds, edits, and deletes. An example of such a form is shown in Figure 12.3. None of the controls on the form have their Control Source filled in. The name of each control corresponds with a field in the database server table. The Open event of the form looks like this:

```
Private Sub Form_Open(Cancel As Integer)
  Set mdb = CurrentDb
  Me.txtTitle.SetFocus
End Sub
```



Enter First Few Characters of Title ID	bu
Select a Title ID:	BU1111
title_id	BU1111
title	Cooking with Computers: Surreptitious Balance Sheets
type	business
pub_id	1389
price	11.95
advance	5000
royalty	10
ytd_sales	3876
notes	Helpful hints on how to use your electronic resources to the best adv
pubdate	6/9/85

Figure 12.3. Unbound form displaying one row of data.

It sets a module level database variable to the current database and sets focus to the txtTitle text box. The After Update event of the text box looks like this:

```
Private Sub txtTitle_AfterUpdate()
  Me!cboTitle.RowSource = "SELECT DISTINCTROW [dbo_titles].[title_id] " _
      & "FROM [dbo_titles] " _
      & "WHERE [dbo_titles].[title_id] Like '" & Me!txtTitle.Text & "*';"
End Sub
```

It sets the RowSource property of the combo box to a Select statement that selects all records from the titles table where the title_id field begins with the first few characters that the user typed. In this way, the combo box is not populated with all the titles from the server. The After Update event of the combo box looks like this:

```
Private Sub cboTitle_AfterUpdate()
  Dim fSuccess As Boolean
  Set mrst = mdb.OpenRecordset("Select * From dbo_Titles " _
      & "Where Title_ID = '" & Me!cboTitle.Value & "';")
  fSuccess = PopulateForm(Me, mrst)
  If Not fSuccess Then
    MsgBox "Record Not Found"
  End If
End Sub
```

The OpenRecordset method is used to open a recordset based on the linked table called dbo_Titles. Notice that only the records with the matching Title_ID are retrieved. Because the Title_ID is the primary key, only one record is returned. The PopulateForm function is then called:

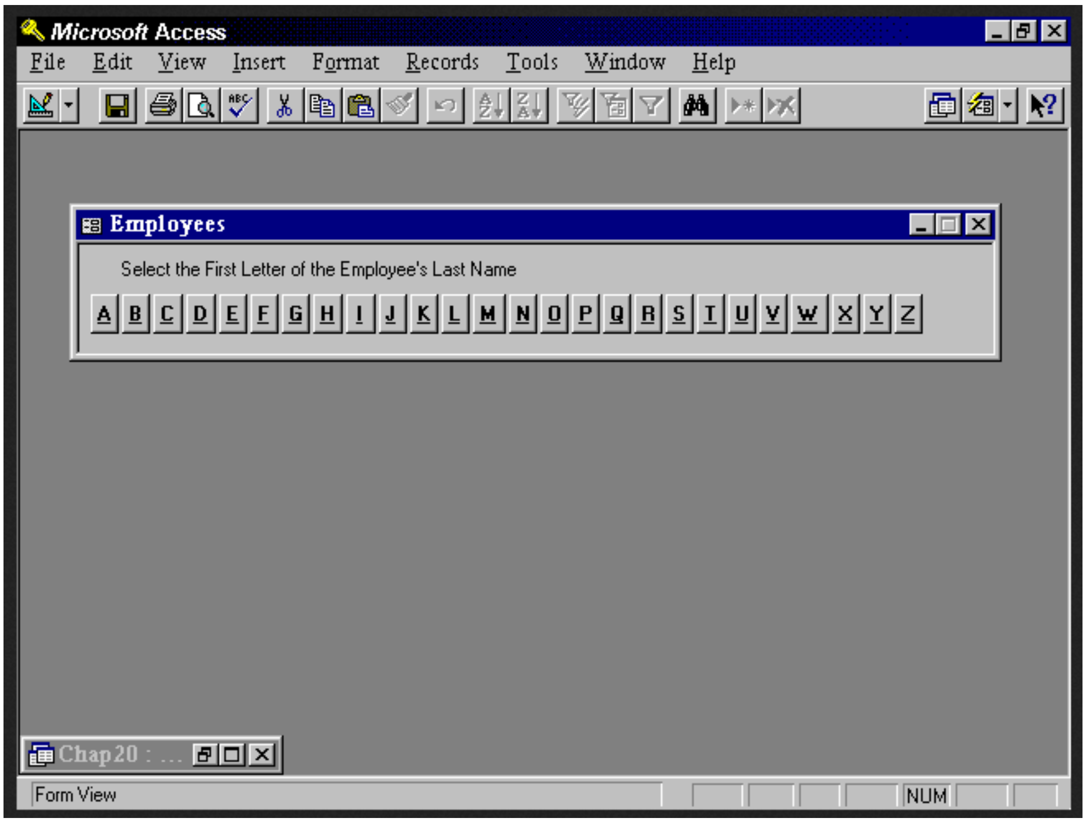
```
Function PopulateForm(frmAny As Form, rstAny As Recordset)
  If rstAny.EOF Then
    PopulateForm = False
  Else
    Dim fld As Field
    For Each fld In rstAny.Fields
      frmAny(fld.Name) = fld
    Next fld
    PopulateForm = True
  End If
End Function
```

The PopulateForm function checks to ensure that the recordset that was passed has records. It then loops through each field on the form, matching field names with controls on the form. It sets the value of each control on the form to the value of the field in the recordset with the same name as the control name.

Note that these changes to the data within the form do not update the data on the database server. Furthermore, the form does not provide for inserts or deletes. You need to write code to issue Updates, Inserts, and Deletes, and you have to provide command buttons to give your users access to that functionality.

3.8 Practical Examples

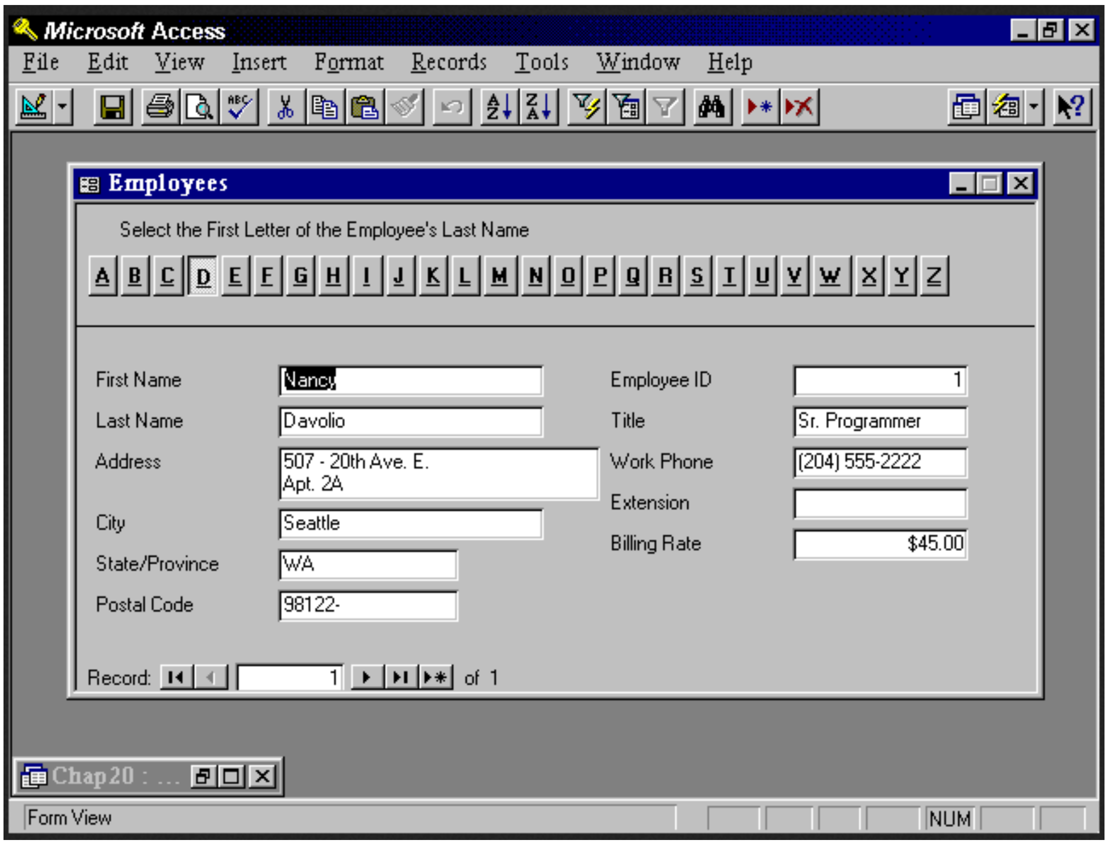
The employee table is probably going to be moved to a database server in the near future because it contains sensitive data. For this reason, let's design the Employee form with client/server in mind. The form limits the data displayed within the form. The form opens with only an option group containing the letters of the alphabet (see Figure 12.4). After the user selects a letter, the detail section of the form is displayed, and the RecordSource for the form is populated with a Select statement (see Figure 12.5).



Select the First Letter of the Employee's Last Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Figure 12.4. Employee form with Option Group to select employee last name.



Select the First Letter of the Employee's Last Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

First Name	Nancy	Employee ID	1
Last Name	Davolio	Title	Sr. Programmer
Address	507 - 20th Ave. E. Apt. 2A	Work Phone	(204) 555-2222
		Extension	
City	Seattle	Billing Rate	$45.00
State/Province	WA		
Postal Code	98122-		

Record: 1 of 1

Figure 12.5. Full view of employee form.

The Open event of the form looks like this:

```
Private Sub Form Open (Cancel As Integer)
  Me.Detail.Visible = False
End Sub
```

The Visible property of the detail section of the form is set to False. The AfterUpdate event of the Option Group looks like this:

```
Private Sub optEmpName_AfterUpdate()
  Me.RecordSource = "Select * from tblEmployees Where LastName Like '" _
      & Chr$(Me![optEmpName].Value) & "*';"
  Me.NavigationButtons = True
  Me.Detail.Visible = True
  DoCmd.DoMenuItem acFormBar, 7, 6, , acMenuVer70
End Sub
```

It populates the RecordSource of the form using a Select statement. It makes the navigation buttons and the detail section of the form visible. Finally, it resizes the window to the form.

3.9 The Web as an example of client/computing

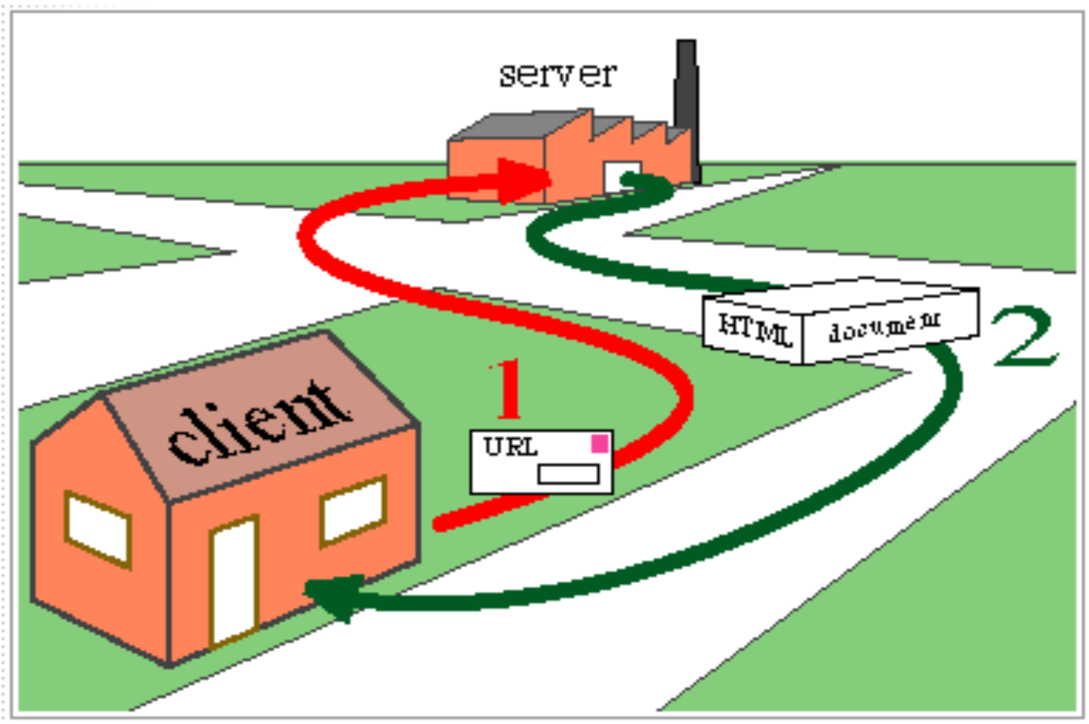


Figure 12.6 Client/sever model via the web

The Web is a world of information available at the click of a mouse. To use it, you need a computer, a connection to the Internet, and a browser.

When you run your browser, it finds and displays pages of information. The function of a Web browser is to interpret the programming language of the web pages (HTML, …) and transform it into the words and graphics that you see on your screen. If you need more information, all you have to do is click on a hyperlink. On each page, certain words, phrases, or even images are highlighted, and clicking on them causes the browser to go off and find another page, which probably contains more highlighted items, and so on.

All Web documents are stored on so-called server computers, represented in the image by a factory. Users can inspect these documents by requesting them from their local (personal) computers, represented by the house, and called a client.

All computers involved in the Web are connected by the Internet, represented by the roads. When you click on a hyperlink, your computer asks a server computer to return to you a document.

For example, starting from the FUT MINNA 'Welcome page' in Minna, your next click might fetch a document from an electronics lab at the other side of the world. All the information seems to be in the little box in front of you, though in reality it is spread over the globe.

The web is also friendly to the network: when you click on a piece of highlighted text your browser 'orders a document' from another computer, receives it by 'return mail' and displays it. You are then free to read the new page at leisure, without further consumption of network resources.

The Web may be used to initiate processes on either the client or the server. A request can start a database search on a server, returning a synthesized document. A document returned in an unfamiliar format can cause the browser to start a process on the client machine in order to interpret it.

The Web's ability to negotiate formats between client and server makes it possible to ship any type of document from a server to a client, provided the client has the appropriate software to handle that format. This makes video, sound and anything else accessible without the need for a single application to be able to interpret everything.

Self-Assessment Exercise(s)

What is web optimization?

Self-Assessment Answer(s)

Website optimization is a phrase that describes the procedures used to optimize – or to design from scratch – a website to rank well in search engines. Website optimization includes processes such as adding relevant keyword and phrases on the website, editing meta tags, image tags, and optimizing other components of your website to ensure that it is accessible to a search engine and improve the overall chances that the website will be indexed by search engines.

4.0 Summary/Conclusion

Strategies are quick means to achieve a result

It is essential to ensure a free connection amongst servers

5.0 Tutor-Marked Assignments

1. How do you reduce the number of connections on a server?
2. How is the web an example of client/server computing?

6.0 References/Further readings

http://101.lv/learn/access/aba20fi.htm

http://www.stratiss.com/clientsrvrapp.shtml

Module 7

Unit 1

Middleware

Contents

1.0 Introduction

Middleware, which is quickly becoming synonymous with enterprise applications integration (EAI), is software that is invisible to the user. It takes two or more different applications and makes them work seamlessly together. This is accomplished by placing middleware between layers of software to make the layers below and on the sides work with each other (Figure 1). On that broad definition, middleware could be almost any software in a layered software stack. Further, middleware is a continually evolving term. Since much of the software business is driven through the perceptions of the "hottest" current technologies, many companies are giving their software the name "middleware" because it is popular.

Middleware, or EAI, products enable information to be shared in a seamless real-time fashion across multiple functional departments, geographies and applications. Benefits include better customer service, accurate planning and forecasting, and reduced manual re-entry and associated data inaccuracies.

Middleware is essential to migrating mainframe applications to client/server applications, or to Java or internet-protocol based applications, and to providing for communication across heterogeneous platforms. This technology began to evolve during the 1990s to provide for interoperability in support of the move to client/server architectures. There are two primary applications for middleware using any of the above middleware initiatives: Computer Telephony and Software Interfaces such as via Java based middleware applications. In this discussion of middleware, we will explore both uses.

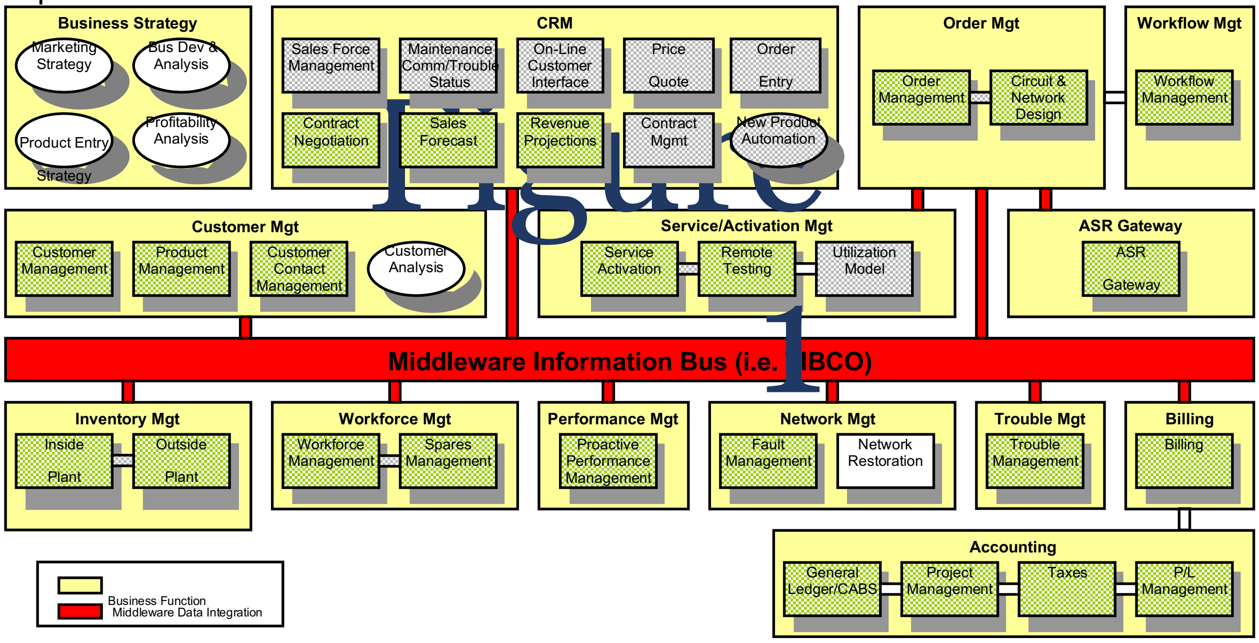


Figure 13.1: Middleware Software "Bus" Architecture

2.0 Learning Outcome:

In this unit the following will be learnt:

a. About Middleware
b. About its prospects

3.0 Learning Content

3.1 Brief History of EAI/Middleware

Enterprise applications, from as early as the 1960s through the late 1970s, were simple in design and functionality, developed largely in part to end repetitive tasks. "There was no thought whatsoever given to the integration of corporate data. The entire objective was to replicate manual procedures on the computer.

By the 1980s, several corporations were beginning to understand the value and necessity for application integration. Challenges arose, though, as many corporate IT staff members attempted to redesign already implemented applications to make them appear as if they were integrated. Examples include trying to perform operational transaction processing (associated with enterprise resource planning (ERP) system functionality) on systems designed for informational data processing (data warehousing functionality).

As ERP applications became much more prevalent in the 1990s, there was a need for corporations to be able to leverage already existing applications and data within the ERP system; this could only be done by introducing EAI (Figures 2 & 3). "Companies once used client/server technology to build departmental applications, but later realized the gains in linking multiple business processes. Other issues driving the EAI market include the further proliferation of packaged applications, applications that addressed the potential problems of the Year 2000, supply chain management/business-to-business (B2B) integration, streamlined business processes, web application integration, and overall technology advances within EAI development.

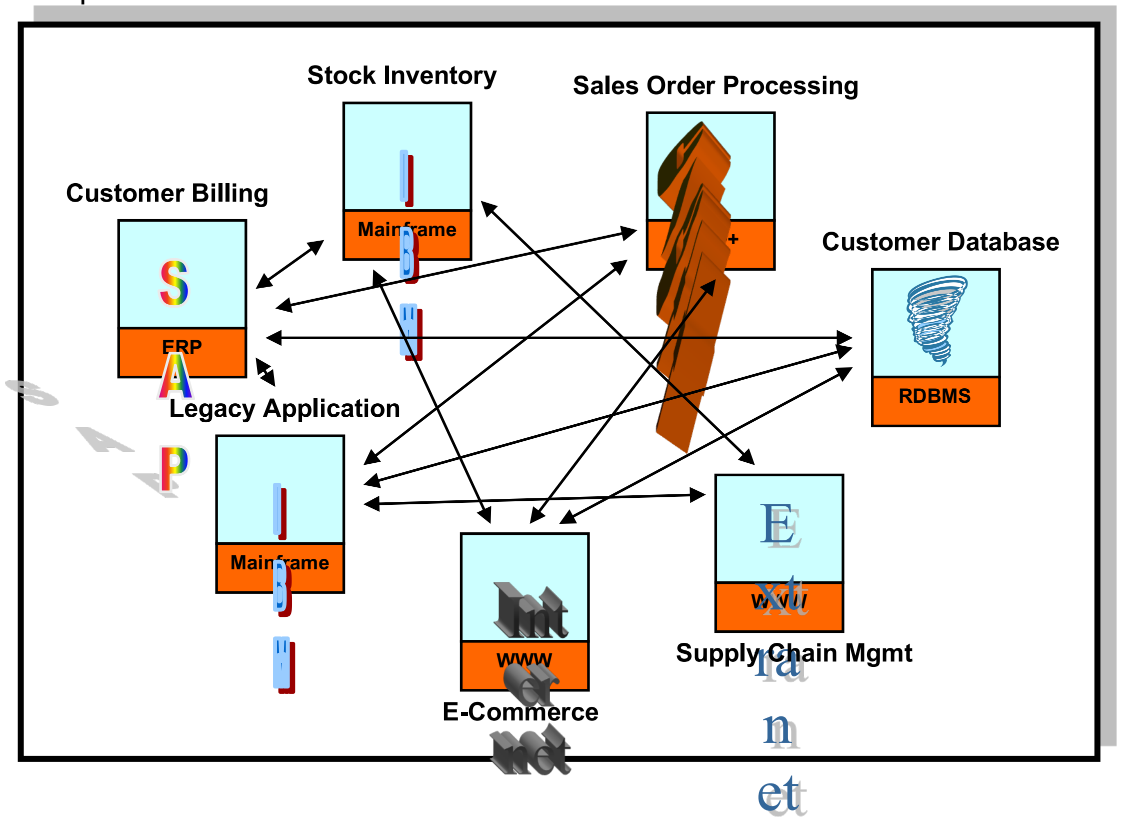


Figure 13.2: Legacy Enterprise Situation

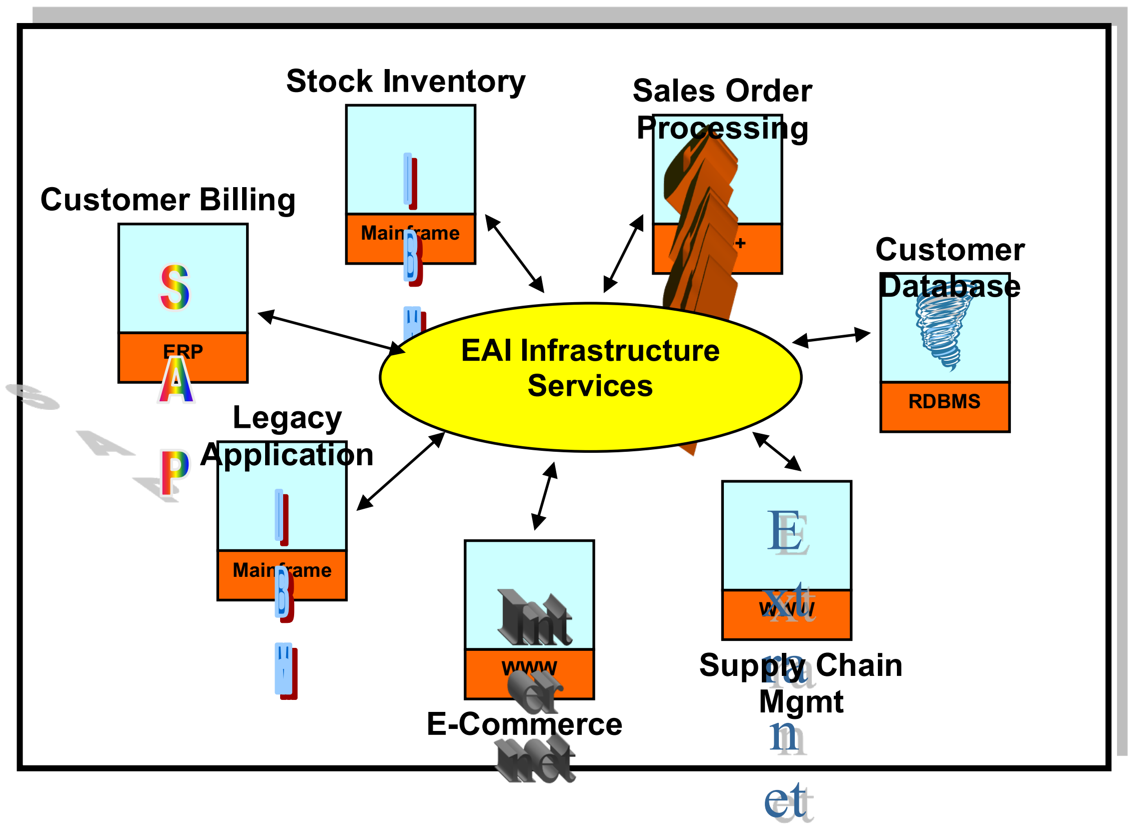


Figure 13.3: Middleware/EAI Enterprise Solution

What is involved in EAI/Middleware?

EAI is very involved and complex, and incorporates every level of an enterprise system – its architecture, hardware, software and processes. EAI involves integration at the following levels:

- ***Business Process Integration (BPI):*** When integrating business processes, a corporation must define, enable and manage the processes for the exchange of enterprise information among diverse business systems. This allows organizations to streamline operations, reduce costs and improve responsiveness to customer demands. Elements here include process management, process modeling, and workflow, which involve the combination of tasks, procedures, organizations, required input and output information, and tools needed for each step in a business process.
- ***Application Integration:*** At this level of integration, the goal is to bring data or a function from one application together with that of another application that together provide near real-time integration. Application Integration is used for, to name a few, B2B integration, implementing customer relationship management (CRM) systems that are integrated with a company's backend applications, web integration, and building Web sites that leverage multiple business systems. Custom integration development may also be necessary, particularly when integrating a legacy application with a newly implemented ERP application.
- ***Data Integration:*** In order for both Application Integration and Business Process Integration to succeed, the integration of data and database systems must be tackled. Prior to integration, data must be identified (where it is located), cataloged, and a metadata model must be built (a master guide for various data stores). Once these three steps are finished, data can then be shared/distributed across database systems.

- **Standards of Integration:** In order to achieve full Data Integration, standard formats for the data must be selected. Standards of Integration are those that promote the sharing and distribution of information and business data – standards that are at the core of Enterprise Application Integration/Middleware. These include COM+/DCOM, CORBA, EDI, JavaRMI, and XML.
- ***Platform Integration:*** To complete the system integration, the underlying architecture, software and hardware, and the separate needs of the heterogeneous network must be integrated. Platform Integration deals with the processes and tools that are required to allow these systems to communicate, both optimally and securely, so data can be passed through different applications without difficulty. For example, figuring out a way for an NT machine to pass information reliably to a UNIX machine is a large task for integrating an entire corporate system.[i]

3.2 Application Programming Interface (Revisited)

In order to fully understand middleware, one must first understand the concepts surrounding Application Programming Interfaces (APIs). The API, by definition, is a software program that is used to request and carry out lower-level services performed by the computer's operation system or by a telephone system's operating system (Figure 4). In a Windows environment, APIs also assist applications in managing windows, menus, icons, and other GUI elements. In short, an API is a "hook" into software. An API is a set of standard software interrupts, calls, and data formats that application programs use to initiate contact with network services, to write applications. This technology is a way to achieve the total cross-platform consistency mainframe communications programs, telephone equipment or program-to-program communications. For example, applications use APIs to call services that transport data across a network. Standardization of APIs at various layers of a communications protocol stack provides a uniform way that is a goal of open systems.

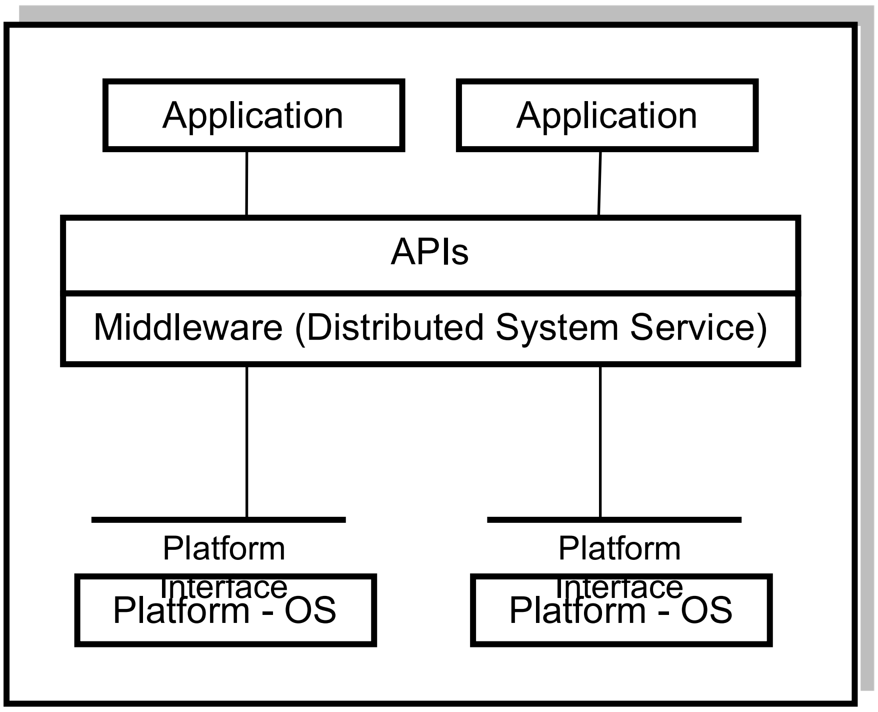


Figure 13.4: Basic API Architecture

3.3 Middleware/EAI Basics

As seen above in Figure 4, middleware works in concert with APIs. Further, it exists between the application and the operating system and network services on a system node in the network. Middleware services are sets of distributed software that provide a more functional set of APIs than does the operating system and network services. This increased functionality allows an application to:

- Locate transparently across the network, providing interaction with another application or service.
- Be independent from network services.
- Be reliable and available.
- Scale up in capacity without losing functionality.

Middleware accomplishes the above tasks via one of the following forms:

1. Transaction processing (TP) monitors, which provide tools and an environment for developing and deploying distributed applications.
2. Remote Procedure Call (RPCs), which enable the logic of an application to be distributed across the network. Program logic on remote systems can be executed as simply as calling a local routine.
3. Message-Oriented Middleware (MOM), which provides program-to-program data exchange, enabling the creation of distributed applications. MOM is analogous to email in the sense it is asynchronous and requires the recipients of messages to interpret their meaning and to take appropriate action.
4. Object Request Brokers (ORBs), which enable the objects that comprise an application to be distributed and shared across heterogeneous networks.
5. Transaction Flow Manager (TFM) or Intelligent Trade Management (ITM) – emerging technologies - which will act as a radar screen that tracks transactions from launch to landing.

3.4 Middleware and Computer Telephony

Middleware in computer telephony tends to be software that sits right above that part of the operating system that deals with telephony. This is the Telephone Server Application Programming Interface (TSAPI) in NetWare and the Telephone Application Programming Interface (TAPI) in Windows. Further, the middleware sits below the user interface and is, thus, invisible to the user.

TSAPI was described by AT&T, its' inventor, as "standards-based API for call control, call/device monitoring and query, call routing, device/system maintenance capabilities, and basic directory services. TAPI is also called the Microsoft/Intel Telephony API. As stated above, the API is a software program that is used to request and carry out lower-level services performed by the computer's operation system or by a telephone system's operating system. In the case of the TAPI, it is the telephone system's operating system. The TAPI set of functions allows windows applications (i.e. Windows 2000, NT) to program telephone-line-based devices such as single and multi-line phones (both digital and analog) and modems and fax machines in a device-independent manner. TAPI essentially does for telephony devices what the Windows printer system did to printers – makes them easy to install and allows many application programs to work with many telephony devices, irrespective of the device manufacturer.

TAPI is an evolutionary API providing convergence of both traditional PSTN telephony and IP Telephony. IP Telephony is an emerging set of technologies which enables voice, data, and video collaboration over existing LANs, WANs, and the Internet. TAPI enables IP Telephony on the Microsoft Windows operating system platform by providing simple and generic methods for making connections between two or more machines, and accessing any media streams involved in the connection (Figure13. 5).

In addition, TAPI also supports standards based H.323 conferencing (these standards define real-time multimedia communications for packet-based networks – now called IP Telephony) and IP multicast conferencing. Further, TAPI utilizes the Windows operating system's Active Directory service to simplify deployment within an organization, and includes quality of service (QoS) support to improve conference quality and network manageability.

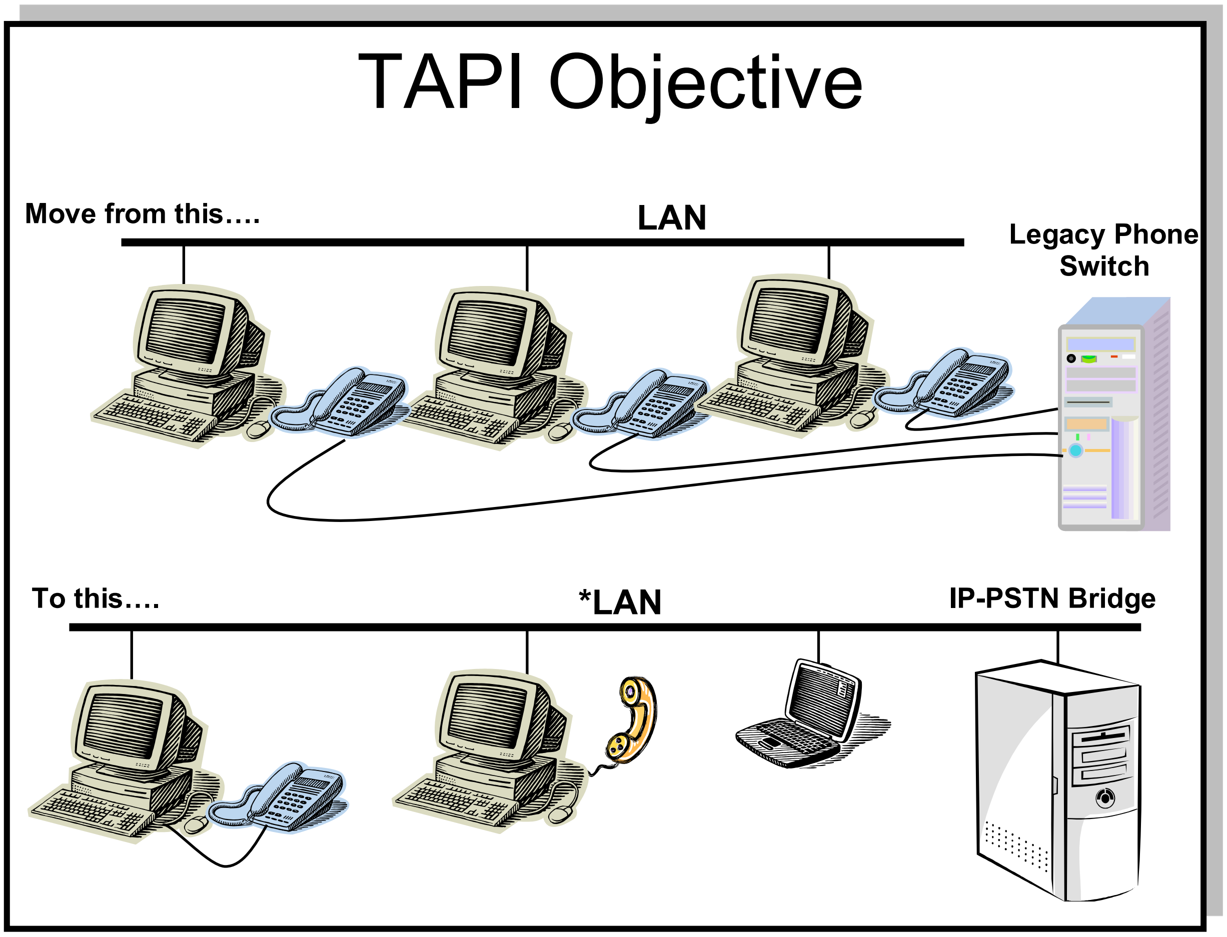


Figure 13.5: TAPI facilitates IP Telephony, which enables voice, data, and video over existing LANs, WANs, and the Internet.

3. 5 Java Middleware – Evolving Use of EAI Technology

Java middleware encompasses application servers like BEA WebLogic, messaging products like Active Software's ActiveWorks and Push Technologies' SpiritWAVE, and hybrid products that build on a DBMS legacy and add server-based Java object execution features. Further, even among application servers there is quite a spectrum, including those that are primarily servlet servers as well as those that are ORB-based or OODB-based. Drawing a line between all these products proves increasingly difficult. The unifying feature, however, is that they all attempt to solve the multitier application deployment problem by using Java and Internet technologies.

The business case to use Java in middleware is compelling. Among the advantages offered by Java middleware are the following:

- The ability of the internet to economically interconnect offices and organizations.
- The need for organizations to cooperate by sharing data and business processes.
- The desire to consolidate generic services and the management of these services.
- The desire to provide centralized application management, including startup, shutdown, maintenance, recovery, load balancing, and monitoring.
- The desire to use open services and protocols.
- The desire to redeploy business logic at will and unconstrained by infrastructure; this necessitates using open APIs and protocols, which are widely supported across most infrastructure products.
- The need to support cooperating mixed-architecture applications.
- The desire to move network and service infrastructure decisions out of the application space, so that system managers can make infrastructure decisions without being hampered by applications that depend on proprietary protocols or features.
- The desire to reduce the diversity and level of programmer staff skills needed and minimize the need for advanced tool-building expertise within projects.
- The desire to leverage object-oriented expertise by extending it into the server realm. Hence, newer object-oriented server products and object-to-relational bridges.

Since the goal of middleware is to centralize software infrastructure and its deployment, Java middleware is the next logical step in the evolution of middleware building upon the client/server roots. Organizations are now commonly attempting integration across departments, between organizations, and literally across the world. The key to building such integration is to leverage the existing technology of the internet. The internet has enticed businesses with its ability to serve as a global network that lets departments and partners interconnect efficiently and quickly.

Java provides a lingo that allows for easy interconnection of data and applications across organizational boundaries. In a distributed global environment that allows an organization no control over what technology choices partners make, smart companies choose open and platform-neutral standards. Companies cannot anticipate who will become their customers, partners, or subsidiaries in the future, so it is not always possible to plan for a common infrastructure with partners. In this uncertain situation, the best decision is increasingly thought to be the use of the most universal and adaptable technologies possible.

Java allows for the reduction of the number of programming languages and platforms that a staff must understand. This is because Java is now deployed in contexts as diverse as internet browsers, stored procedures within databases, business objects within middleware products, and client-side applications.

3.6 Middleware Usage Considerations

Although middleware has numerous obvious benefits in solving application connectivity and interoperability problems, middleware services are not without tribulations. The main issues are outlined below:

- There is a gap between principles and practice. Many popular middleware services use proprietary implementations (making applications dependent on a single vendor's product). The commercial off-the-shelf (COTS) software vendors (i.e. TIBCO and Vitria) are addressing this problem through the more stringent use of standardization. In spite of this progress, the issue remains a substantial problem.
- The sheer number of middleware services is a barrier to using them. To keep their computing environment manageably simple, developers have to select a small number of services that meet their needs for functionality and platform coverage. Since the goal of middleware is to provide for maximum interoperability, this barrier is particularly frustrating.
- While middleware services raise the level of abstraction of programming distributed applications, they still leave the application developer with hard design choices. For example, the developer must still decide what functionality to put on the client and server sides of a distributed application.

The key to overcoming the above listed problems is to completely understand both the application problem and the value of middleware services that can enable the distributed application. To determine the types of middleware services required, the developer must identify the functions required. These functions all fall into one of three categories:

1. Distributed system services – These include critical communications, program-to-program, and data management services. RPCs, MOMs and ORBs are included in this type of service.
2. Application enabling services – These middleware services give applications access to distributed services and the underlying network. This type of service includes transaction monitors and database services such as Structured Query Language (SQL).
3. Middleware management services - These enable applications and system functions to be continuously monitored to ensure optimum performance of the distributed environment. Middleware Developmental Stage.

3.7 Middleware Developmental Stage

A significant number of middleware services and vendors exist. Middleware applications are continuing to grow with the installation of more heterogeneous networks and as the client/server architecture continues to entrench and evolve into Java-centric and Internet-protocol centric architectures, more middleware products are emerging. For example, at least 46 Java middleware products were in existence in 2000 and that number has nearly doubled to date. Such products will continue to grow in demand as examples such as the Delta Airlines Cargo Handling System emerge. This system uses middleware technology to link over 40,000 terminals in 32 countries with UNIX services and IBM mainframes.

3.8 Middleware Costs, Limitations, and Economic Outlook

No technology is right for every situation and each technology has associated costs (monetary and otherwise). Some examples of the kinds of costs and limitations that a technology may possess are the following: A technology may impose an otherwise unnecessary interface standard, it might require investment in other technologies (see bullets below), it might require investment of time or money, or it may directly conflict with security or real-time requirements. Specific dependencies include:

What is needed to adopt this technology (this could mean training requirements, skill levels needed, programming languages, or specific architectures).

- How long it takes to incorporate or implement this technology.
- Barriers to the use of this technology.
- Reasons why this technology would not be used.

After taking all of the above into consideration, the costs of using middleware technology (i.e. license fees) in system development are entirely dependent on the applications, required operating systems, and the types of platforms.

As an example, at a start-up telecommunications carrier-of-carriers service provider, AFN Communications, there were approximately $20 million dollars expended just to implement operational and business support systems that were fully integrated across a client/server middleware information “bus”, which indicates that all applications are integrated via a common path. TIBCO software was utilized for this implementation and the use of this software required staff training, it took approximately one year to complete the project, a multi-platform and multi-OS situation presented continual barriers to using the information bus technology, and concerns over continued maintenance of the complicated integration were constantly discussed. In spite of all of the challenges, the heavily used middleware system is in place and has allowed their organization to reduce their necessary headcount by about 35%.

Another current example of the use of middleware technology is in the financial business sector. Financial institutions are preparing for T+1, the holy grail of the world's stock market settlement system. Under this exacting standard, trades would settle within one day instead of the two, three or even more days that it takes in some markets. These financial institutions are preparing to spend a lot of time with their middleware vendors as a 2004 deadline for full implementation of T+1 looms. Middleware, or EAI, is an "absolutely fundamental requirement" for straight-through processing (STP). "Participating in end-to-end transactions without the middleware layer that provides messaging and securities and data exchange between applications running the business is not possible. Further, without the advent and extension of the TFM or ITM transaction tracking capabilities from launch to landing within the middleware technology, it will be impossible to meet the T+1 goal. Concentrating only on the aforementioned T+1 market need, the financial industry is expected to spend $8 billion. Although hesitant to venture a guess regarding the dollars that will be dedicated to EAI, estimates of spending by financial institutions alone, which includes middleware, will hit $733 million this year and rise to $1.19 billion by 2003, a compounded annual growth rate of almost 25 percent. In spite of the extensive demand for middleware services both domestically and worldwide, individual companies specializing in these areas are experiencing varied financial results:“ Messaging and identity management vendor Critical Path Inc. said it will meet Wall

Street's expectations with revenues between $22 million and $24 million. The company did not specify its profit outlook, but said operating expenses excluding certain items will be $27 million to $29 million. It said those numbers are better than expected. In early afternoon trading, the company's shares were 11 percent higher, at 90 cents. Integration software vendor See Beyond Technologies Corp. said it will report revenues between $32 and $34 million, with a loss of 8 cents to 10 cents per share. Analysts had expected a profit, as well as revenue of $38 million. Its shares dropped 17 percent to $2.10 in early afternoon trading. Legacy extension/integration vendor Jacada Ltd. forecast revenues of $5.2 million to $5.5 million, and a net loss of 6 cents to 8 cents per share. Analysts were looking for a loss of 3 cents per share. The company's shares were trading down 17 percent at $1.90 Wednesday afternoon." ii[10].

3.9 Conclusion

Potential Challenges of Middleware/EAI Middleware product implementations are unique to the vendor. This results in a dependence on the vendor for maintenance support and future enhancements. Reliance on vendors, in this manner, could have a negative effect on a system's flexibility and maintainability. However, when evaluated against the cost of developing a unique middleware solution, the system developer and maintainer may view the potential negative effect as acceptable. Also, as Java and internet protocol middleware technologies evolve, many of these potentially detrimental issues will dissolve.

Self-Assessment Exercise(s)

State four examples of middleware?

What is middleware in android?

Self-Assessment Answer(s)

Examples of Middleware

- Database access technology - e.g. ODBC (Open DataBase Connectors)
- Java's database connectivity API : JDBC
- Remote computation products - e.g. ONC RPC, OSF RPC and RMI (Java Remote Method Invocation)
- Distributed Computing Environment (DCE) products, Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM)

What is middleware in android?

Android provides a middleware layer including libraries that provide services such as data storage, screen display, multimedia, and web browsing. Because the middleware libraries are compiled to machine language, services execute quickly. Middleware libraries also implement device-specific functions, so applications and the application framework need not concern themselves with variations between various Android devices. Android's middleware layer also contains the Dalvik virtual machine and its core Java application libraries.

4.0 Summary/Conclusion

- Middleware is computer software that provides services to software applications beyond those available from the operating system.
- Middleware can be described as "software glue.Thus middleware is not obviously part of an operating system, not a database management system, and neither is it part of one software application.
- Middleware makes it easier for software developers to perform communication and input/output, so they can focus on the specific purpose of their application.

5.0 Tutor-Marked Assignments

1. What do you understand by middleware?
2. How Does it work

6.0 References/Further readings

http://www.sei.cmu.edu/str/descriptions/middleware.html

Inmon, William. "A Brief History of Integration." EAI Journal.

Ren, Frances. "The Marketplace of Enterprise Application Integration (EAI). http://www.public.asu.edu/~mbfr2047/eai.html

Vander Hey, Dan. "One Customer, One View." Intelligent Enterprise. March 2000.

Yee, Andre. "Demystifying Business Process Integration." EaiQ.

http://eai.ittoolbox.com/browse.asp?c=EAIPeerPublishing&r=%2Fpub%2Feai%5Foverview%2Ehtm

Newton, Harry. Netwon's Telecom Dictionary. 15th ed. New York: Miller Freeman, Inc., 1999.

Unit 2

Web Technology

Contents

1.0 Introduction

A network of network is called internet. The internet is used to perform the following types of tasks:

- Electronic Mail
- Research
- Discussion
- Interactive Games

2.0 Learning Outcome:

In this unit the following will be learnt:

a. About a web
b. About tags in web

3.0 Learning Content

3.1 A short history of HTML

The secret of the internet is the separation of the act of transmission of data from the display of data. By separating the processes, transmission could be greatly fast because the server did not have to worry about how the data looked at the other end, since the display was left to the local receiving computer.

The information alongside the data (called tags) tells the receiving computer how to display different types of information. These coding methods, called markup languages, travel with data and are interpreted by the retrieving software. Codes to tell the computer how to interpret hypertext documents is called Hypertext Markup Language.HTML became the standard way to tag pages of information traveling over the internet.

Browsers and the Web

The web is simply a way of looking at the internet. This vast network consists of computers that manage data and the communication links via software and hardware called server.

Web servers receive requests for information, go out and find it in t heir databases, and return the proper pages of data to the requesting computer.

Browsers assisted users in finding information on the web and properly displaying it on computer screens.

Hypertext Transfer Protocol (HTTP) was developed as a way to find information and retrieve it over telephone lines.

3.2 Multimedia Tools

We are standing on the edge of the ability to broadcast on-demand video and video over the web, such as live event. In addition, working in three dimensions in real time motion (Called virtual reality) opens up new vistas.

Databases for Multimedia

Multimedia databases such as Illustra Information Technologies automatically open information sent via email and store images, captions and text in separate fields. Because of growing size and number of HTML pages, and audio/video files that make up a site, you need databases to manage the storage and use of multimedia items. There a r e four types of databases available for managing web sites: traditional relational databases such as Oracle; object oriented databases such as Object store; relational-object hybrid databases such as Illustra; and specialty databases such as Cinebase.

Oracle now supports multimedia storage via its new Universal Server Enterprise Edition, which includes Oracle Spatial Data option for storage and \retrieval of images, the ConText option for full text retrieval through SQL, and Oracle Video option, which serves video files to multiple clients.

Object oriented databases are useful for storing images, video and audio because they use an object model rather than a tabular model in the database design. An object can be anything. Hybrid mixtures of object-oriented and relational databases use the benefits of tabular data and object-based data. Specialty databases are built for single purpose, such as Media-On-Demand.

Digital Video

There are two video compression formats vying for power on the web: QuickTime and MPEG. Digital video does not have the picture quality of broadcast video because personal computers cannot support the bandwidth required to transmit all the data involved. Digital video is also called animation because the images that comprise a moving picture are sent one-by-one in a single pass without interlacing and at half the number of frames per second as broadcast video. Netscape supports the following plug-ins for viewing and creating digital video animations:

- Action by Open2U.
- CineWeb by Digigami.
- CoolFusion by Iterated Data Systems.
- InterVU by InterVu, Inc.
- MovieStar by Intelligence at large.
- QuickTime for Netscape by Apple Computer.
- ViewMovie by Ivan Cavero Belaunde,

Helper Applications

A helper application is basically a program that cooperates with Netscape and other browsers to perform functions such as displaying the contents of files, which the browsers cannot do.

Some of the helper applications are:

- Movie Viewing – AVI Video player for windows
- MPEG Animation viewing – Ghostscript

Electronic Publishing

It is one thing for HTML to send pictures and text over telephone wires to remote computers where a browser interprets arcane codes to display a replica of what was broadcast. It is quite another to be able to reproduce published documents with their complex layouts, colors and art on a remote computer. The remote computer may not have the fonts installed that were used to produce the document. What is needed is a way to make documents electronically portable.

A quick review of Markup Concepts

The goal of markup languages is to create a universal way to identify the structure and content of a document. SGML is the earliest specification for markup. SGML is based upon the use of a special file called a Document Type Description (DTD) document. The DTD sdfx describes the structure of a document.

The document's basic framework is defined, including types of elements used and how these elements are related to each other. The contents of a document are labeled with tags that identify the document's structure. Each tag has a beginning and ending. HTML was developed as a subset of SGML for handling hypertext link.

3.3 What is HTML 4.0?

Html 4.0 corrects design deficiencies produced by Html 3.2, especially in the support of forms and tables. Changes between Html 3.2 and Html 4.0

1. New elements and attributes - Q, INS, BUTTON, LABEL, FIELDSET
2. Modified elements and attributes - Table frame and rules attributes
3. Deprecated Elements - applet, center, font, basefont
4. Obsolete Elements - XMP, PlainText

Design Concepts Underlying Html 4.0 HTML 4.0 is an extensible web framework because it allows the specification to grow with the times.

The working group used the following concepts to ensure that their specifications met that goal:

- Interoperability
- Internationalization
- Accessibility
- Enhanced tables
- Style sheets
- Scripting – enabled
- Maintain html's ease of use

3.4 The Function of HTML in Contemporary Web Publishing

HTML is a subset of SGML, Standard Generalized Markup Language.

SGML is itself a rigorous format meant for designing other markup languages.

The basic concept behind HTML or any other markup languages is to embed commands that describe how different elements within a document should be interpreted in that document.

The Web's Defining Framework

HTML provides Web designers both a matrix and a mechanism.

As a matrix, its purpose is to the basic framework in which content is embedded.

As a mechanism, it provides ways to specify the manner in which that content is presented and, to a limited extent, allows various options for its functioning.

Paired with newer developments of scripting languages and style sheets, the true potential of HTML seems about to be fully realized, and the World Wide Web to really come into flower.

Tags and Elements in HTML

Tags are the main resources for HTML authors. They define the existence of all the elements a web page contains. Indeed, they define the existence of the page itself.

Each element is said to be contained by tags. When a web browser or other client agent parses the HTML code in a web page, it looks for these defining indicators to tell it how to process and display the elements. The code structure of elements is a start tag, followed by the contents, if any, and the end tag.

Most elements have required start and end tags, the function of which is to delimit the beginning and ending of the element. Other elements have required start tags, but optional end tags. Still other elements have only a start tag and no end tag. Each element has its own attributes. The start of an HTML page is defined by the <HTML> start tag and the end is defined by </HTML> tag. The HEAD and BODY elements are within that HTML element.

The material between <HEAD> </HEAD> tags is largely informational, although this is also where JavaScript and style-connection information are placed. The BODY element contains the majority of the other elements. The elements contained within a Web Page's body tags are many and various.

They include elements for controlling the insertion of images and sounds, tying in Java applets and other objects, setting the appearance and size of text, adding tables and forms.

3.5 Basic Structural Elements and Their Usage

There are four basic structural elements in HTML.

The <! DOCTYPE> tag represents the Document Type Declaration (DTD), which defines the version of HTML used on the page.

The next is the <HTML> tag itself, which simply states that the page is an HTML document.

The third is the HEAD element, which contains the document's title and other information. Then there is the BODY element, where the actual web page itself is contained.

A basic HTML page:

```
<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 // EN">
<HTML>
<HEAD>
<TITLE> PAGE TITLE GOES HERE </TITLE>
OPTIONAL META DATA GOES HERE.
</HEAD>
<BODY>
WEB PAGE GOES HERE…….
>/BODY>
</HTML>
```

The <DOCTYPE> element

Its purpose is to declare to browsers exactly what version of HTML was used to create the document. The general DOCTYPE declaration for HTML 4.0 is

\<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 // EN">

The reason for the EN is that the HTML specification has not yet been developed in any language but English. If has not yet been developed in any language but English.

If you are replacing the BODY element with FRAMESET, you should use the following DTD.

<! DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 FRAMESET // EN">

There is another technique for supplying the DOCTYPE declaration. This is called the System Identifier DTD. A System Identifier DTD is declared as follows:

<! DOCTYPE

HTML

SYSTEM

"http:// WWW.fut.Org/DTD/HTML 4 - strict.dtd">

The difference in this approach is that the DTD is referenced not by name, but by specifying the URL where it is found.

The HTML Element

It is the first and last thing in a web page, and its absence means that no web browser recognizes your work as an HTML page.

HTML element must have both start and end tags in order to function properly.

Thus, the document begins with <HTML> and ends with </HTML>.

The HTML start tag can have three attributes. The first is version, and it takes a URL value. The URL points to a location that has the Document Type Definition (DTD) for the version of HTML in use on that page.

The HTML start tag can also contain the lang and dir attributes, which respectively, establish the human language in which the web page is written and the direction of the printing.

HTML> tag might look like:

<HTML version = http://www.fut.org/DTD/HTML4- strict.dtd lang=en dir=rtl>

The HEAD Element

The HEAD element is a container for a HTML document header information.

It contains one required element – TITLE and several other optional attributes.

The HEAD element has both start and end tags, beginning with <HEAD> and ending with </HEAD>. A typical HEAD element looks like this:

<HEAD><TITLE> My Home Page </TITLE></HEAD>

The text between the TITLE tags is displayed in the title bar of a visitor's web browser.

Metadata

In addition to the title, a head element can contain other elements known as metadata. It is data that is not shown to the person viewing the page, but useful to user agents and search engines.

The currently used elements in this category are:

- Base
- Link
- Meta
- Script
- Style

The Base Element

The base element establishes a base URL from which relative URLs referenced in the html document can be calculated. Example,

<base href=http://www.test.org/>14

The result of this declaration is that any relative url in the html document is appended to this base url to achieve he full url.

Thus, the relative url index.html would be interpreted as http://www.test.org/index.html.

The Link Element

Its purpose is to establish relationships among different html documents. Used only in the head element.

Link takes three main attributes in addition to href: rel, rev and title.

Rel defines a forward relationship and rev defines a reverse one.

Example,

<link rev=”previous” href=”chapter1.html”

<link rel=”next” href=”chapter3.html”>

The title attribute simply gives a title to the document referenced by the url.

Link types are:

Alternate - Points to an optional replacement for the current document

Appendix - points to an appendix

Bookmark - Points to a named anchor

Chapter – Points to a chapter

Copyright - Points to a copyright statement

Glossary - Points to s glossary of terms

Help - Points to a help document

Index - Points to an index

Next - Points to the document that comes after the current one

Previous - Points to the document that comes before the current one

Start - Points to the beginning document

The Meta Element

When used with the name and content attributes, its main function is to establish metadata variable information for use by web search agents.

For example,

<meta name=”wt” content=”web technology”>

The script and style elements

Used for including script and style sheet into an html document.

The body element

The body element is where the displayable web page is found.

A typical body in an html document looks like:

<body background=”bg.gif” text=”000000” link=”0000FF”vlink=”FF8C00” alink=”000000”>Text, images, etc. are found here…</body>

3.6 Traditional Text and Formatting, Deprecated and Obsolete Elements

As HTML evolves, the function of some elements is replaced or suppressed by newer. Some are still fully functional and useful, but can be done more economically or efficiently.

For example, both the <IMG> and <APPLET> tags still work, but both perform similar tasks – embedding particular object in the HTML page.

A single <OBJECT> element would be designed that would encompass all possible embeddable objects.

Thus, both <IMG> and <APPLET> are now deprecated in favor of <OBJECT> tag. When an element is deprecated, that means that the W3C recommends that you no longer use it, but use, the newer solution.

However, deprecated elements are still a part of HTML specification, and still be supported by browsers.

Obsolete elements, are no longer defined in the HTML specification, and W3C does not require that client agents support them.

Text Layout

There are 2 basic types of text affecting elements in html.

The first kind performs text layout tasks and the second affects text’s appearance.

The P Element

The <p> tag is used to denote the beginning of a new paragraph.

Although the end tag </P> exists, its use is optional.

If the end tag is not used, the beginning of the next block level element is interpreted as the end of the paragraph. The align attribute is used to set the alignment of the paragraph with respect to the page size. Values are LEFT, RIGHT, CENTER and JUSTIFY. Example: <p align="center">

<P> tag Example

<p align=left>This paragraph is Left Aligned. </P>

<p align=center>This paragraph is Centered. </P>

<p align=right>This paragraph is Right Aligned. </P>

<p align=justify>This paragraph is justified. </P>

The BR Element

The <br> tag inserts a single line break.

The <br> tag is an empty tag which means that it has no end tag.

When placed after images, the clear attribute controls how text is handled when wrapping around those images. The clear attribute has four possible values: none, left, right and all.

The CENTER Element

The CENTER element causes all text between its start and end tags to be centered between the margins.

<CENTER> Text Goes Here </CENTER>

The HN Element

The highest level of heading is represented by <H1> tag, the lowest by the <H6> tag.

- <H1> This is an H1 heading. </H1>
- <H2> This is an H2 heading. </H2>
- <H3> This is an H3 heading. </H3>
- <H4> This is an H4 heading. </H4>
- <H5> This is an H5 heading. </H5>
- <H6> This is an H6 heading. </H6>

The HR Element
Horizontal rules are used to visually divide different segments of web pages from one another. A simple horizontal rule is coded as follows:
<HR>

Lists
One of the most popular methods for organizing information is by using lists.
HTML presents three basic kinds of lists: unordered lists, ordered lists, and definition lists.
In unordered lists, the list items are marked with bullets.
In ordered lists, they are marked with numbers, Roman numerals, or letters.
Definition lists are a little different; they have a pair of values, one for the term, and the other for its definition.

Unordered Lists
Unordered lists are specified with the <UL> tag.

Unordered lists are used when the order of the list items is unimportant.
The type attribute defines the type of bullets used to denote the individual list items.
The three options are: disc, circle and square.
Example

```
<UL type="disc">
<LH> UG Courses </LH>
<LI> BE (CSE) </LI>
<LI> BE (ECE) </LI>
<LI> BE (EEE) </LI>
</UL>
<UL type="square">
<LH> PG Courses </LH>
<LI> MCA </LI>
<LI> ME (CSE) </LI>
<LI> MBA </LI>
</UL>
```

Ordered Lists
Ordered lists are specified with the <OL> tag. They are used when the order of the list item is significant. OL elements have the type and start attributes.
The type attribute selects the kinds of numbering system utilized to order the list.
Example

```
<html>
<body>
<h4>An Ordered List:</h4>
<ol type=I>
<li>Coffee</li>
<li>Tea</li>
<li>Milk</li>
</ol>
</body>
</html>
```

Definition Lists

Definition lists are specified with the <DL> tag.

Definition lists consist of pairs of values, the first being the term to be defined, and the second being the definition of the tem. Example:

```
<DL>
<DT> Satellite Dish</DT>
```

<DD> Antenna like device which functions to receive and concentrate television signals. </DD>

</DL>

Nested Lists
Any kind of list – ordered, unordered or definition – can be nested within another list.

```
<UL type=disc>
<LI> UG Courses </LI>
<OL type=i>
<LI> BE (CSE) </LI>
<LI> BE (ECE) </LI>
<LI> BE (EEE) </LI>
</OL>
<LI> PG Courses </LI>
<OL type=i>
<LI> ME (CSE) </LI>
<LI> MCA </LI>
LI> MBA </LI>
</OL>
</UL>
```

Text Styles

The B and STRONG Elements
Using the <B> and <STRONG> tags has the effect of rendering text in bold print.
The I and EM elements
Using the <I> and <EM> tags has the effect of rendering text in italicized print.
STRIKE and U Elements
The STRIKE element causes text to be struck through.
The U element underlines the affected text.
The BIG, SMALL, SUP and SUB elements
These four elements actually change the size or position of the affected text.
The FONT and BASEFONT Elements
The FONT and BASEFONT Elements perform the same task and use the same methods for doing so.
The difference between them is the scope of their effect.

Both set the size, color and font face for the text, but the basefont element is global for all body text in the document, whereas the FONT element is strictly local and affects only the text between its start and end tags.

Example

```
<BASEFONT size=4 color="#000000" face="arial">
  <P>This is body text using the base font size. </P>
 <FONT size=+3> This is locally increased font. </FONT>
```

Self-Assessment Exercise(s)

What is HTTP?

Self-Assessment Answer(s)

HTTP, short for HyperText Transfer Protocol, is the protocol for transferring hypertext documents that makes the World Wide Web possible. A standard web address (such as http://www.google.com/) is called a URL; the prefix (http in the example) indicates its protocol.

4.0 Summary/Conclusion

Web 2.0 is a concept that takes the network as a platform for information sharing, interoperability, user-centered design,[1] and collaboration on the World Wide Web.

A Web site allows users to interact and collaborate with each other in a social media dialogue as creators (prosumers) of user-generated content in a virtual community, in contrast to websites where users (consumers) are limited to the passive viewing of content that was created for them.

Examples of Websites include social networking sites, blogs, wikis, video sharing sites, hosted services, web applications, mashups and folksonomies

5.0 Tutor-Marked Assignments

1. What do you understand by HTML
2. What are the advantages of a website?

6.0 References/Further readings

www.FaaDooEngineers.com

www.wikipedia.com

Unit 3

Introduction to Applets

Contents

1.0 Introduction

Applet is a java program that can be embedded into HTML pages. Java applets runs on the java enabled web browsers such as Mozilla and internet explorer.

Applet is designed to run remotely on the client browser, so there are some restrictions on it. Applet can't access system resources on the local computer.

Applets are used to make the web site more dynamic and entertaining.

2.0 Learning Outcome:

In this unit the following will be learnt:

a. About an applet
b. About how it works

3.0 Learning Content

3.1 Advantages of Applet

Applets are cross platform and can run on Windows, Mac OS and Linux Platform.

Applets run in a sandbox, so the user does not need to trust the code, so it can work without security approval. Applets are supported by most web browsers.

Applets are cached in most web browsers, so will be quick to load when returning to a web page.

Writing Applets instead of writing a program that has a main() method as in the case of an application, applets are any class that subclasses the Applet class and override some of its methods.

A simple Applet:

```
import java.awt.*;
import java. applet.*;
class Hello World Applet extends
Applet{
// Default constructor
     public void HelloWorld()

   {            // Call parent constructor
    super();
     }
public void paint ( Graphics g )
{
   g.setBackground ( Color.white );
  g.setColor   ( Color.blue );
  g.drawString ( "Hello world!", 0, size().height - 5);

  }
   }
```

3.2 Writing Applets

The <Applet> Tag

Here is an example of a simple APPLET tag:

<applet code="MyApplet.class" width=100 eight=140></applet>

This tells the viewer or browser to load the applet whose compiled code is in myApplet.class (in the same directory as the current HTML document), and to set the initial size of the applet to 100 pixels wide and 140 pixels high.

The <Object> Tag

The <object> tag replaces <applet> in html 4.0.

In this example, the CLASSID is used to specify the name of the java code to execute.

<Object CLASSID = “java:HelloWorld.start” height=100 width=100></Object>

The <Param> Tag

<Param> tags can be placed within the <Applet> or new <Object> tag to send additional information to the applet.

Example:

```
Public class Salutation extends Applet
{
Private String s;
Public void init()
{
s = getParameter(“salutation”);
}
Public void paint(Graphics g)
{
    g.drawString(salutation,10,10);
}
}
```

The <Object> Tag Html Code:

<Applet code=”Salutation.class” width=200 height=250><param name=”salutation” value=”Greetings and Saltutations”></param>

</Applet>

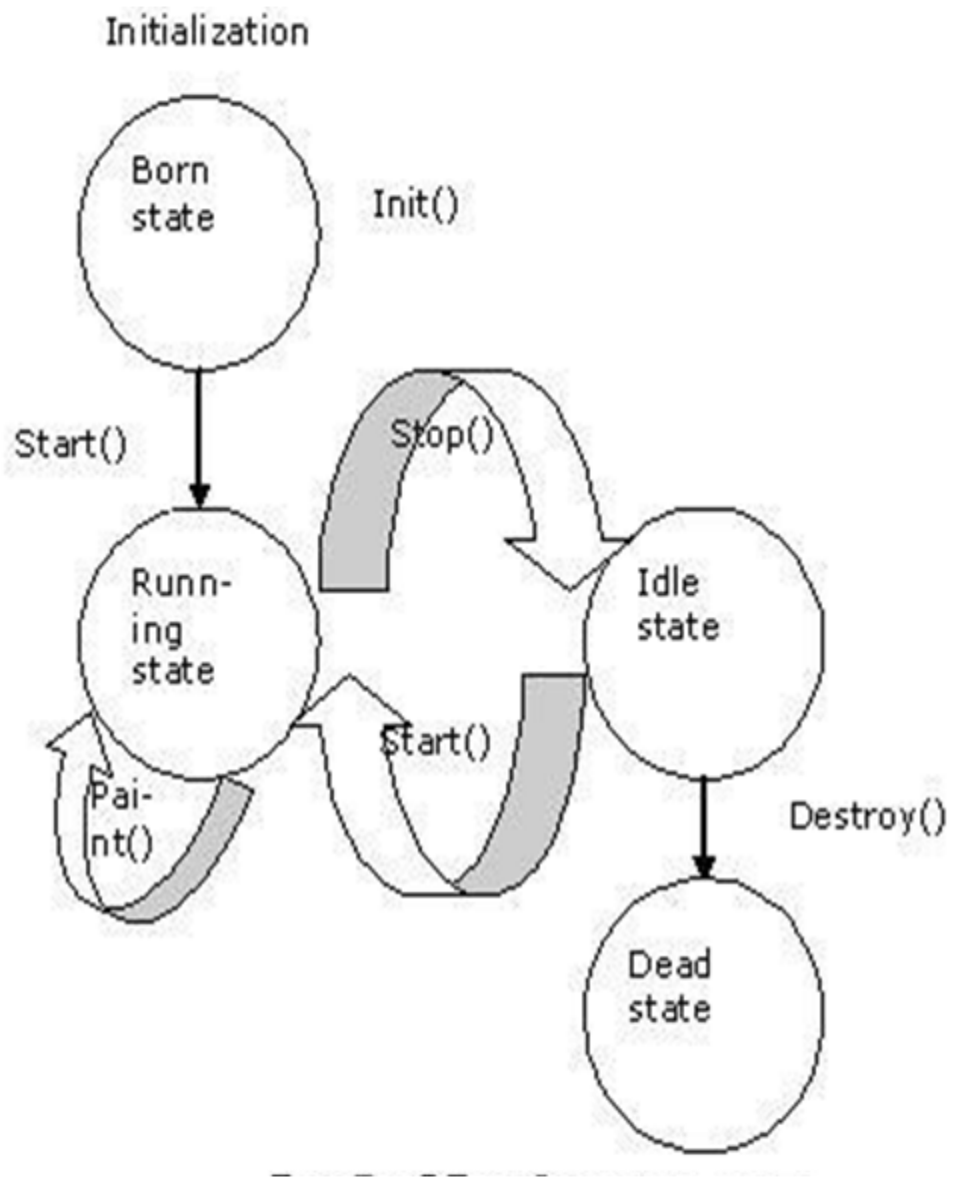
3.3 Applet methods (Applet Life Cycle Methods)

A list of the applet methods that can be overridden are as follows:

- init() - This method is called after the web page containing an applet is loaded into a browser.

- start() - This method is called every time the user comes back to the web page containing the applet.
- stop () - This method is called every time the user leaves the page containing the applet.
- paint () - This method is called any time the applet's repaint() method is called.
- Update() - This method is called every time the applet is repainted. By default, it erases the entire screen.

3.4 Applet Life Cycle



3.5 Resources Available to Applets

A number of resources are available to applets.

- The methods that provide access to resources are:

- AudioClip getAudioClip() - This method can be used to load an audio clip from a URL.
- Image getImage() - This method is used to load images from an URL.
- Void play(URL url) - This methods plays a loaded audio clip.

- Methods that provide access to information about an applet are:
    - URL getCodeBase() - This method returns the URL from which the applet's class files were loaded.
    - URL getDocumentBase() - This method returns the URL from which the applet's HTML document was loaded.
    - AppletContext getAppletContext() - This method returns an object of type AppletContext. An AppletContext object is useful for loading and viewing web pages.
    - showStatus(String msg) - This method displays the message in the browser's status bar.
    - resize()
    - This method resizes the applet's bounds.

3.6 Graphics and Double Buffering

Some of the most commonly used methods of Graphics class are:

- void draw string String string, int x, int y) - Draws a string to the screen at the x, y coordinates.
- void clearRect(int x, int y, int width, int height) - Clears a rectangle on the screen with its top-left corner located at x, y and the specified width and height.
- void fillRect (int x, int y, int width, int height) - Fills a rectangle on the screen with its top-left corner located at x, y and the specified width and height.
- void drawRect (int x, int y, int width, int height) - Draws a rectangle on the screen with its top-left corner located at x, y and the specified width and height.
- Void dispose() - Frees resources associated with a graphics context.

Example

```
import java.awt.*;
            import java.applet.*;
            public class graph_ex extends Applet
            {
```

```
public void paint (Graphics g)
{ setBackground(Color.cyan);
   g.drawString("Here are a selection of blank shapes.",20,40);
   g.drawLine(20,40,200,40);
   g.setColor(Color.blue);
   g.drawLine(20,50,70,90);
   g.setColor(Color.red);
   g.drawRect(100,50,32,55);
   g.setColor(Color.green);
   g.drawOval(150,46,60,60);
   g.setColor(Color.magenta);
   g.drawArc(230,50,65,50,30,270);
   g.setColor(Color.black);
    g.drawString("Here are the filled equivalents.",20,140);
    g.drawLine(20,140,200,140);
    g.setColor(Color.yellow);
    g.fillRect(100,150,32,55);
    g.setColor(Color.pink);
    g.fillOval(150,146,60,60);
  g.setColor(Color.darkGray);
   g.f illArc(230,150,65,50,30,270);
}
}
```

Drawing in applets is almost always done with double-buffering.

This means that drawing is first done to an off-screen image, and when all is done, the off-screen image is drawn on the screen. This reduces the nasty flickering applets otherwise have.

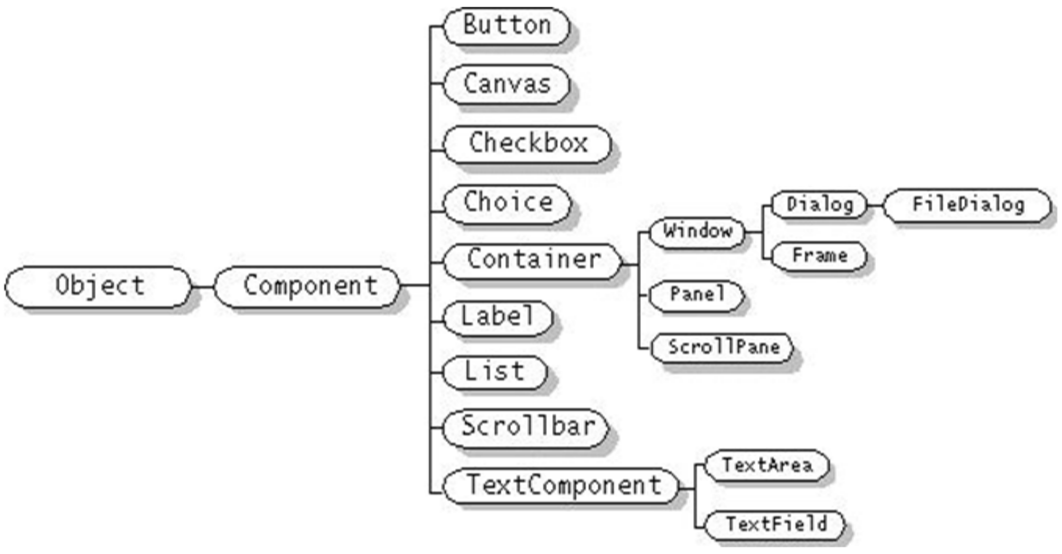
```
 import java.applet.*;
import java.awt.event.*;
import java.awt.*;
public class DoubleBuffering extends Applet implements
MouseMotionListener
{
    Graphics bufferGraphics;
    Image offscreen;
```

```
Dimension dim;
int curX, curY;
public void init()
{
        dim = getSize();
     // We'll redraw the applet eacht time the mouse has moved.
     addMouseMotionListener(this);
     setBackground(Color.black);
      offscreen = createImage(dim.width,dim.height);
     // by doing this everything that is drawn by bufferGraphics
     // will be written on the offscreen image.
     bufferGraphics = offscreen.getGraphics();
}
  public void paint(Graphics g)
{
     // Wipe off everything that has been drawn before
     // Otherwise previous drawings would also be displayed.
     bufferGraphics.clearRect(0,0,dim.width,dim.width);
     bufferGraphics.setColor(Color.red);
     bufferGraphics.drawString("Bad Double-buffered",10,10);
     // draw the rect at the current mouse position
     // to the offscreen image
     bufferGraphics.fillRect(curX,curY,20,20);
     // draw the offscreen image to the screen like a normal
     image.
     // Since offscreen is the screen width we start at 0,0.
     g.drawImage(offscreen,0,0,this);
}
public void update(Graphics g)
{
     paint(g);
}
// Save the current mouse position to paint a rectangle there.
// and request a repaint()
public void mouseMoved(MouseEvent evt)
```

```
                {
                    curX = evt.getX();
                    curY = evt.getY();
                    repaint();
                }
            public void mouseDragged(MouseEvent evt)
                {
                }
            }
```

3.7 Creating a User Interface Using the AWT

The following figures show the inheritance hierarchies for all the AWT component classes.



The basic structure of an applet that uses each of these predefined methods is:

```
import java.applet.Applet;
import java.awt.*;
public class AppletTemplate extends Applet {
public void init() {
// create GUI, initialize applet
}
public void start() {
// start threads, animations etc...
}
public void paint(Graphics g) {
// draw things in g
```

```
}
public void stop() {
// suspend threads, stop animations etc...
}
public void destroy() {
// free up system resources, stop threads
}
}
        import java.applet.*;
        import java.awt.event.*;
        import java.awt.*;
        public class calc extends Applet implements ActionListener{
          TextField tf1, tf2, tf3;
          Label l1,l2,l3;
          String Add, Subtract,Multiply,Divide;
            public void init(){
            tf1 = new TextField(10);
            tf2 = new TextField(10);
            tf3 =new TextField(10);
            l1 = new Label("Number 1");
            l2 = new Label("Number 2");
            l3 = new Label("Result");
             add(l1);  add(tf1);
             add(l2);  add(tf2);
            add(l3);  add(tf3);
            Button b = new Button("Add");
            Button c = new Button("Subtract");
            Button d = new Button("Multiply");
            Button e = new Button("Divide");
            b.addActionListener(this);
            c.addActionListener(this);
            d.addActionListener(this);
            e.addActionListener(this);
            add(b);      add(c);
            add(d);      add(e);
        }
public void actionPerformed(ActionEvent e)
{
    String s=e.getActionCommand();
    int n1 = Integer.parseInt(tf1.getText());
    int n2 = Integer.parseInt(tf2.getText());
    if(s=="Add")
         tf3.setText(String.valueOf(n1+n2));
    if(s=="Subtract")
       tf3.setText(String.valueOf(n1-n2));
    if(s=="Multiply")
       tf3.setText(String.valueOf(n1*n2));
    if(s=="Divide")
       tf3.setText(String.valueOf(n1/n2));
        }
}
```

Self-Assessment Exercise(s)

A. Show how an applet can be represented in an html tag

B. What are Paint() and Update() method used for()?

Self-Assessment Answer(s)

A. Show how an applet can be represented in an html tag

<Html><Applet code="Salutation.class" width=200 height=250><param name="salutation" value="Greetings and Saltutations"></param>

</Applet></html>

B. What are Paint() and Update method used for()?

Paint() - This method is called any time the applet's repaint() method is called.

Update() - This method is called every time the applet is repainted. By default, it erases the entire screen.

4.0 Summary/Conclusion

In computing, an applet is any small application that performs one specific task that runs within the scope of a larger program, often as a plug-in.

Java Applets can provide web applications with interactive features that cannot be provided by HTML
Examples on how to include an applet in an HTML page refer to the description of the <APPLET> tag.

5.0 Tutor-Marked Assignments

1. What are applets?
2. State two advantages of an applet?
3. What do the following stand for: init(), start(), stop()?

6.0 References/Further readings

www.FaaDooEngineers.com