BT 0081
Software Engineering

Contents

Edition: Spring 2009

BKID – B1090 30th Sept. 2009

Manipal

SUBJECT INTRODUCTION

Software Engineering (BT0081) is a four credit subject in BScIT Program. This SLM explores the concepts of software engineering, as its importance is very well felt in software community and attempts to develop technologies that will make it easier, faster and less expensive to build high quality software products. More emphasis is given on software development strategies and assessment of process life cycle models. This SLM on "Software Engineering" is divided into fourteen units. The brief account of them is given below:

Unit 1: Software Development Approaches

Computer software has become a driving force. It is the engine that drives business decision making. This unit explains the evolving role of software. Various software characteristics and its applications are also discussed in this unit.

Unit 2: Software Design Process

Software systems are now omnipresent. Software is used to help run manufacturing industry, schools, universities, health care, finance and government. This unit describes the meaning and definition of software engineering and various software development models such as – serial or linear sequential development model, iteration model, increment model, finally parallel or concurrent development model.

Unit 3: Software Reliability

Reliability is the most important dynamic characteristic of almost all software systems. Unreliable software results in high costs for end-users. This unit deals with the software reliability metrics and programming for reliability. This unit also gives a brief account of software reuse.

Unit 4: Software Design Principles

The output of the requirement analysis process is a set of system models that present abstract description of the system to be developed. This unit deals with the system models, software and architectural design.

Unit 5: Object Oriented Design

Object-oriented design transforms the analysis model created using object-oriented analysis into a design model that serves as a blueprint for software construction. This unit covers the object oriented design, service usage, object interface design and structural decomposition.

Unit 6: Assessment of Process Life-cycle Models

In software development, the critical role of time as a factor in development is considered, including not only the various scheduling constraints on time to develop, but also the business-driven parameter of time to market. This unit covers the overview of the assessment of process, the dimension of process, and the need for a business model in software engineering.

Unit 7: Configuration Management

Large software systems may be considered as configuration of components. Many different versions, made up of different component configurations of the system are created. This unit covers change management, version and release management, software maintenance, software reengineering, and software re-factoring.

Unit 8: Software Testing Techniques

The importance of software testing and its implications with respect to software quality cannot be overemphasized. This unit deals with the software testing fundamentals, testing principles, white box testing, control structure testing, black box testing, boundary value analysis, and testing GUIs.

Unit 9: Software Testing Assurance

Verification and validation encompass a wide array of software quality assurance activities that include formal technical reviews, quality and configuration audits, performance monitoring etc. This unit deals with the verification and validation (V&V), test plan, test strategies, testing methods and tools.

Unit 10: Software Testing Strategies

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason, a template for software testing should be defined for the software process. This unit covers the organizing for

software testing, software testing strategy, unit testing, top-down integration, and bottom-up integration testing.

Unit 11: People and Software Engineering

Multidisciplinary thinking helps us to understand problems better and therefore solve problems more effectively. This unit covers the traditional software engineering, the importance of people in problem solving process, and the human driven software engineering.

Unit 12: Software Technology and Problem Solving

Information technology has ubiquitously influenced business and affected management approaches to problem solving. A key manifestation of this technology is the software technology that has pervaded all aspects of life, from household appliances to entertainment devices and communication media. This unit deals with the software technology as enabling business tool and a limited business tool.

Unit 13: Diversification of Problem Solving Strategies in SE

The factors that have contributed to the diversification of software process models have often been related to the expansion in goals and capabilities in the software industry. This unit covers the understanding of diversification in software engineering, the hidden value of differences and skills required at the project management levels.

Unit 14: Case Study

Business schools have been using case studies for years to develop a student's analytical abilities, but they are rarely seen in software engineering courses. Here one such a case study is presented which focus the students on specific software development problems.

Objective of studying the subject

After studying this subject, you should be able to:

- apply all the software design principles
- develop industry standard software based on object oriented design
- test the developed software using various testing strategies

EduNxt™ For various multimedia and other resources on the subject, log on to EduNxt portal of SMU DDE at www.smude.edu.in.

Unit 1 Software Development Approaches

Structure:

1.1 Introduction

Computer software has become a driving force. It is the engine that drives business decision making. It serves as the basis for modern scientific investigation and engineering problem solving. It is a key factor that differentiates modern products and services. It is embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial processes, entertainment, office products etc. Software is virtually inescapable in the modern world. And as we move into the twenty-first century, it will become the driver for new advances in everything from elementary education to genetic engineering.

Computer software is the product that software engineers design and build. It encompasses programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms and data that combine numbers and text but also include representations of pictorial, video and audio information.

Software is a set of application programs that are built by software engineers and are used by virtually everyone in the industrialized world either directly or indirectly.

Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture and our everyday activities.

Software is generally built like you build any other successful product, by applying a process that leads to a high quality result that meets the needs of

the people who will use the product. We apply a software engineering approach to develop this product.

From the point of view of a software engineer, the work product is the programs, documents and data that are computer software. But from the user's point of view, the work product is the resultant information that somehow makes the user's world better.

Objectives:

After studying this unit, you should be able to:

- explain the evolving role of software
- describe software characteristics
- discuss various software applications

1.2 Evolving Role of Software

Software impact on our society and culture continues to be profound. As its importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs. Some of these technologies are targeted at specific application domain and others focus on a technology domain and still others are more broad based and focus on operating systems.

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 and as a vehicle used to deliver the product, software acts as the basis for the control of the computer, the networks and the creation and control of other programs.

Software delivers the most important product of our time – information. Software transforms personal data so that the data can be made more useful in a local context. It manages business information to enhance competitiveness. It provides a gateway to worldwide information networks and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architecture, vast increase in memory and storage capacity, and a wide variety of input and output options have all made it possible for a significant contribution of software on our day to day life.

Why does it take so long to get software developed?

Why are development costs so high?

Why can't we find all the errors before we give the software to customer?

Why do we continue to have difficulty in measuring progress as software is being developed?

These are some of the common questions that we have been asking programmers in all the past history of the software development era and we continue to ask them even now. This concern infact has led us to the adoption of software engineering practice.

1.3 Software Characteristics

Software is a logical rather than a physical system element. Therefore software has characteristics that are considerably different than those of hardware:

a) Software is developed or engineered; it is not manufactured in the classical sense.
b) Software doesn't "wear out".
c) Although the industry is moving toward component-based assembly, most software continues to be custom built.

1.4 Software Applications

Software may be applied in any situation for which a prespecified set of procedural steps has been defined. 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. Software that controls an automated machine 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 multi-user 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.

Software applications can be neatly compartmentalized into different categories.

System software: System software is a collection of programs written to service other programs. Some system software processes complex information structures. Other systems applications process largely indeterminate data. It is characterized by heavy interaction with hardware, heavy usage by multiple users, concurrent operation that requires scheduling, resource sharing, and sophisticated process management, complex data structures and multiple external interfaces.

Real time software: Software that monitors / analyzes / controls real-world events as they occur is called real time.

Business Software: Business information processing is the largest single software application area. Discrete systems like payroll, accounts receivable/payable have evolved into management information systems (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 operations or management decision making.

Engineering and scientific software: Engineering and scientific software has been characterized by “number crunching” algorithms. Applications range from astronomy to volcano logy, from automotive stress analysis to space shuttle orbital dynamics and from molecular biology to automated manufacturing.

Embedded software: Embedded software resides only in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can provide very limited and esoteric functions or provide significant function and control capability.

Personal computer software: Day to day useful applications like word processing, spreadsheets, multimedia, database management, personal

and business financial applications are some of the common examples for personal computer software.

Web-based software: The web pages retrieved by a browser are software that incorporates executable instructions and data. In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

Artificial Intelligence software: Artificial Intelligence software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge based systems, pattern recognition, game playing are representative examples of applications within this category.

Software crisis: The set of problems that are encountered in the development of computer software is not limited to software that does not function properly rather the affliction encompasses problems associated with how we develop software, how we support a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software.

1.5 Summary

Software has become the key element in the evolution of computer based systems and products. Over the past 50 years, 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 persist even today. Software has become the limiting factor in the continuing 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 engineering is to provide a framework for building software with higher quality.

Self Assessment Questions:

1. Software is a _____ rather than physical system element.
2. _____ refers to the predictability of the order and timing of information.
3. _____ software is a collection of programs written to service other programs.

4. _____ software has been characterized by “number crunching” algorithms.
5. _____ software resides only in read-only memory and is used to control products and systems for the consumer and industrial markets.

1.6 Terminal Questions

1. How have the early days affected software development practices today?
2. What do you understand by information determinacy?
3. Discuss the impact of “information era”
4. What are the different categories of software?

1.7 Answers

Self Assessment Questions:

1. Logical
2. Information determinacy
3. System
4. Engineering and scientific
5. Embedded

Terminal Questions:

1. Software impact on our society and culture continues to be profound. As its importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs. (Refer section 1.2)
2. 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. (Refer section 1. 4)
3. Software may be applied in any situation for which a pre-specified set of procedural steps has been defined. Information content and determinacy are important factors in determining the nature of a software application. (Refer section 1.4 & 1.5)
4. System software, Application software, Embedded software etc. (Refer section 1.4)

Unit 2 Software Design Processes

Structure:

2.1 Introduction

Software systems are now omnipresent. Software is used to help run manufacturing industry, schools, universities, health care, finance and government. The computational power and sophistication of computers have increased ever since, while their costs have been reduced dramatically. The specification, development, management and evolution of these software systems make up the discipline of software engineering.

The more powerful a computer is the more sophisticated programs it can run. Even simple software systems have a high inherent complexity so that engineering principles have to be used in their development. The discipline of software engineering discusses systematic and cost-effective software development approaches, which have come out from past innovations and lessons learnt from mistakes. Software Engineering principles have evolved over the past fifty years of contributions from numerous researches and software professionals.

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 and the generic phases. 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 the software development process the focus is on the activities directly related to production of the software, for example, design coding, and testing. A development process model specifies some activities that, according to the model, should be performed, and the order in which they should be performed.

As the development process specifies the major development and quality assurance activities that need to be performed in the project, the development process really forms the core of the software process. The management process is decided based on the development process. Due to the importance of development process, various models have been proposed.

Objectives:

After studying this unit, you should be able to:

- explain the meaning of software engineering
- describe program maintenance
- discuss software product and software process models

2.2 What is Meant by Software Engineering?

Software Engineering is an engineering discipline whose focus is the cost-effective development of high-quality software systems. It is a sub discipline of Computer Science that attempts to apply engineering principles to the creation, operation, modification and maintenance of the software components of various systems.

Software engineering is an engineering discipline which is concerned with all aspects of software production. Software engineering is concerned with the practicalities of developing and delivering useful software. The cost of software engineering includes roughly 60% of development costs and 40% of testing costs. Structured approaches to software development include system models, notations, rules, design advice and process guidelines. Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software are the key challenges facing Software Engineering.

What is engineering?

Engineering is the application of well-understood scientific methods to the construction, operation, modification and maintenance of useful devices and systems.

What is software?

Software comprises the aspects of a system not reduced to tangible devices e.g., computer programs and documentation. It is distinguished from hardware, which consists of tangible devices, and often exists as collections of states of hardware devices. The boundary between hardware and software can be blurry, as with firmware and micro code.

Systems

A system is an assemblage of components that interact in some manner among themselves and, possibly, with the world outside the system boundary.

We understand systems by decomposing them into:

Subsystems

System components

It is very difficult to separate the software components of a system from the other components of a system.

2.3 Definitions of Software Engineering

Software Engineering is the systematic approach to the development, operation, maintenance and retirement of software. This is the definition as per IEEE.

According to Bauer, Software Engineering is nothing but the establishment and use of sound engineering principles in order to obtain economical software that is reliable and works efficiently on real machines.

There is yet another definition for software engineering. It is the application of science and mathematics by which the capabilities of computer equipment are made useful to humans via computer programs, procedures, and associated documentation. This is by Boehm.

An engineering approach to software engineering is characterized by a practical, orderly, and measured development of software. The principal aim

of this approach is to produce satisfactory systems on time and within budget. There is a good reason for tackling the problem of planning, developing, evaluating and maintaining software using the engineering approach. Quite simply this approach is needed to avoid chaos in developing software. The engineering approach is practical because it is based on proven methods and practices in software development. The approach is orderly and development can be mapped to fit customer requirements. Finally, this approach is measured, during each phase, software metrics are applied to products to gauge quality, cost and reliability of what has been produced.

Software Maintenance

Maintenance refers to the support phase of software development. Maintenance focuses on CHANGE associated with error correction, adaptations required as the software environment evolves, and changes due to enhancements brought about by changing customer requirements or improved capability on the part of developers. Four types of maintenance are typically encountered.

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.

Adaption

Overtime, 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. Perfective Maintenance extends the software beyond its original functional requirements. Developers can also initiate enhancements by utilizing their experience on similar project and replicating the same on earlier developed systems.

Self Assessment Questions

1. _____ is an engineering discipline whose focus is the cost-effective development of high-quality software systems.
2. The cost of software engineering includes roughly _____ of development costs and _____ of testing costs.
3. _____ refers to the support phase of software development.

2.4 The Serial or Linear Sequential Development Model

This Model also called as 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 support. Figure 2.1 shows the linear sequential model for software engineering. Modeled after a conventional engineering cycle, the linear sequential model has the following activities:

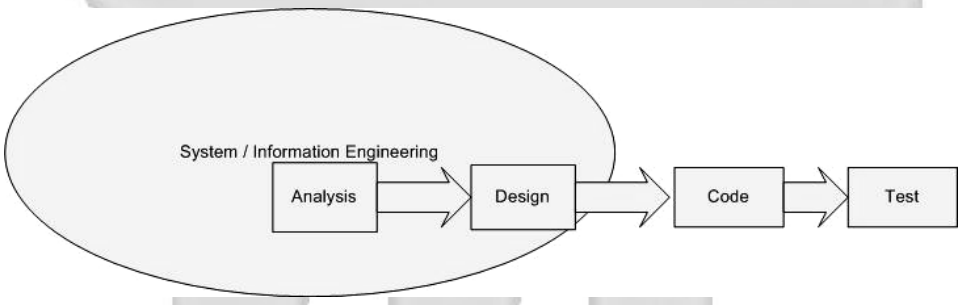


Fig. 2.1: The linear sequential model

System/Information Engineering and modeling:

Because software is a part of a large system, 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 interact with other element such as hardware, people and databases. System engineering and analysis encompasses requirements gathering at the system level with a small amount of top level design and analysis. Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

Software requirement analysis

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

Design

Software design is actually a multistep process that focuses on four distinct attributes of a program, data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding 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, ensuring 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 results.

Support

Software will undergo change after it is delivered to the customer. Change will occur because errors have been encountered, because the software must be adopted to accommodate changes in its external environments or because the customer requires functional or performance enhancements. Software maintenance re-applies each of the preceding phases to an existing program rather than a new one.

A successful software product is one that satisfies all the objectives of the development project. These objectives include satisfying the requirements and performing the development within time and cost constraints.

Generally, for any reasonable size projects, all the phases listed in the model must be performed explicitly and formally.

The second reason is the one that is now under debate. For many projects the linear ordering of these phases is clearly the optimum way to organize these activities. However some argue that for many projects this ordering of activity is unfeasible or suboptimal. Still waterfall model is conceptually the simplest process model for software development that has been used most often.

Limitation of the linear sequential model

1. The linear sequential model or waterfall model assumes the requirement of a system which can be frozen (baseline) before the design begins. This is possible for systems designed to automate an existing manual system. But for a new system, determining the requirements is difficult as the user does not even know the requirements. Hence, having unchanging requirements is unrealistic for such projects.
2. Freezing the requirements usually requires choosing the hardware (because it forms a part of the requirement specifications) A large project might take a few years to complete. If the hardware is selected early, then due to the speed at which hardware technology is changing , it is likely the final software will use a hardware technology on the verge of becoming obsolete. This is clearly not desirable for such expensive software systems.
3. The waterfall model stipulates that the requirements be completely specified before the rest of the development can proceed. In some situations it might be desirable to first develop a part of the system completely and then later enhance the system in phases. This is often done for software products that are developed not necessarily for a client, but for general marketing, in which case the requirements are likely to be determined largely by the developers themselves.
4. It is a document driven process that requires formal documents at the end of each phase. This approach tends to make the process documentation-heavy and is not suitable for many applications, particularly interactive application, where developing elaborate documentation of the user interfaces is not feasible. Also, if the development is done using fourth generation language or modern

development tools, developing elaborate specifications before implementation is sometimes unnecessary.

Despite these limitations, the serial model is the most widely used process model. It is well suited for routine types of projects where the requirements are well understood. That is if the developing organization is quite familiar with the problem domain and requirements for the software are quite clear, the waterfall model or serial model works well.

RAD Model

Rapid Application Development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle. The RAD model is a high speed adaptation of the linear sequential model in which the rapid development is achieved by using component-based construction. If requirements are clear and well understood and the project scope is constrained, the RAD process enables a development team to create a fully functional system within a very short period of time.

The RAD approach encompasses the following phases:

Business modeling

Here we try to find answers to questions like what information drives the business process. What information is generated? Who generates it? Where does the information go? Who processes it? Etc.,

Data modeling: Here the information flow which would have been defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business.

Process modeling

The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application generation:

RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components(when possible)

or create reusable components(when necessary). In all cases, automated tools are used to facilitate construction of the software.

Testing and turnover

Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.

Drawbacks of the RAD model:

- For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
- RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either, RAD projects will fail.
- Not all types of applications are appropriate for RAD. If a system cannot be properly modularized, building the components necessary for RAD will be problematic. If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
- RAD is not appropriate when technical risks are high. This occurs when a new application makes a heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

2.5 Iterative Development Model

The iterative enhance model counters the third limitation of the waterfall model and tries to combine a benefit of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the waterfall model. The increment models provide feedback to the client i.e., useful for determining the final requirements of the system.

In the first step of this model, a simple initial implementation is done for a subset of the overall problem. This subset is one that contains some of the key aspects of the problem that are easy to understand and implement and which form a useful and usable system. A project control list is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far the project is at any given step from the final system.

Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called the design phase, implementation phase and analysis phase. The process is integrated until the project control list is empty, at which time the final implementation of the system will be available. The iterative enhancement process model is shown in figure 2.2.

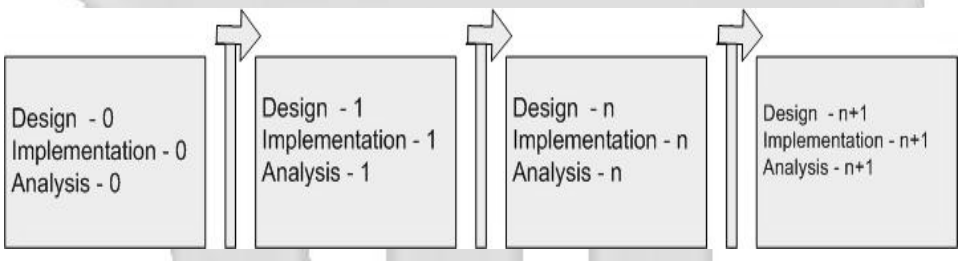


Fig. 2.2: The iterative enhancement model

The project control list guides the iteration steps and keeps track of all tasks that must be done. Based on the analysis, one of the tasks in the list can include redesign of defective components or redesign of the entire system. Redesign of the system will occur only in the initial steps. In the later steps, the design would have stabilized and there is less chance of redesign. Each entry in the list is a task that should be performed in one step of the iterative enhancement process and should be completely understood. Selecting tasks in this manner will minimize the chance of error and reduce the redesign work. The design and implementation phases of each step can be performed in a top-down manner or by using some other technique.

One effective use of this type of model is for product development, in which the developers themselves provide the specifications and therefore have a lot of control on what specifications go in the system and what stay out.

In a customized software development, where the client has to essentially provide and approve the specifications, it is not always clear how this process can be applied. Another practical problem with this type of development project comes in generating the business contract-how will the cost of additional features be determined and negotiated, particularly because the client organization is likely to be tied to the original vendor who developed the first version. Overall, in these types of projects, this process model can be useful if the “core” of the applications to be developed is well understood and the “increments” can be easily defined and negotiated. In client-oriented projects, this process has the major advantage that the client’s organization does not have to pay for the entire software together, it can get the main part of the software developed and perform cost-benefit analysis for it before enhancing the software with more capabilities.

2.6 The Incremental Development Model

The incremental model combines elements of the linear sequential model with the iterative of prototyping. Figure 2.3 shows the incremental model that applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable “increment” of the software. For e.g., 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 could incorporate the prototyping paradigm.

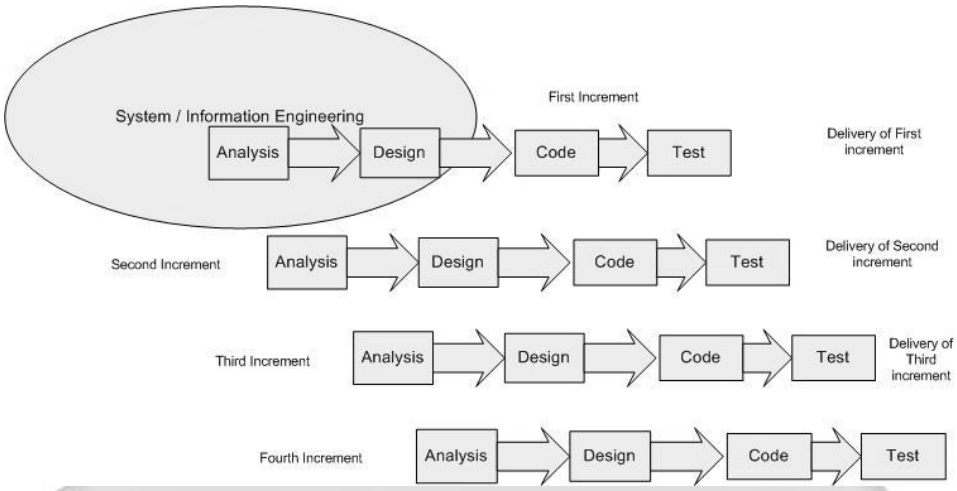


Fig. 2.3: The incremental model

When an incremental model is used, the first increment is a core product. That is, basic requirements are addressed, but many supplementary features remain undelivered. The customer uses the core product. 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. The incremental process model is iterative in nature. The incremental model focuses on the delivery of an operational product with each increment.

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 e.g. a major system might require the availability of new hardware i.e., under development and whose delivery date is uncertain. It 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.

Spiral model

This model couples the iterative nature of the prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

Usually the spiral model consists of around six task regions or phases.

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 risks.

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. (e.g. documentation and training).

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.

As the evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center. The first circuit around the spiral might result in the development of a product specification; subsequent circuit passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each passes through the planning region resulting in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Self Assessment Questions

4. The _____ Model is also called as the Classic life cycle or the Waterfall model.
5. _____ is actually a multistep process that focuses on four distinct attributes of a program, data structure, software architecture, interface representations, and procedural (algorithmic) detail.
6. The _____ combines elements of the linear sequential model with the iterative of prototyping.

2.7 The Parallel or Concurrent Development Model

The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states. For e.g., the engineering activity defined for the spiral model is accomplished by invoking the following tasks: Prototyping and / or analysis modeling, requirements specification, and design.

Figure 2.4 shows a schematic representation of one activity with the concurrent process model. The activity-analysis may be in any one of the states noted at any given time. Similarly, other activities (e.g. Design or customer communication) can be represented in an analogous manner. All activities exist concurrently but reside in different states. For e.g., early in a project the customer communication activity has completed its first iteration and exists in the **awaiting Changes** State. The analysis activity (which existed in the **none** state while initial customer communication was completed) now makes a transition into the **under development** state. If the customer indicates that changes in requirements must be made, the analysis activity moves from the **under development** state into the **awaiting changes** state.

The concurrent process model defines a series of events that will trigger transition from state to state for each of the software engineering activities. For e.g., during early stages of design, an inconsistency in the analysis model is uncovered. This generates the event analysis model correction, which will trigger the analysis activity from the **done** state into the **awaiting Changes** State.

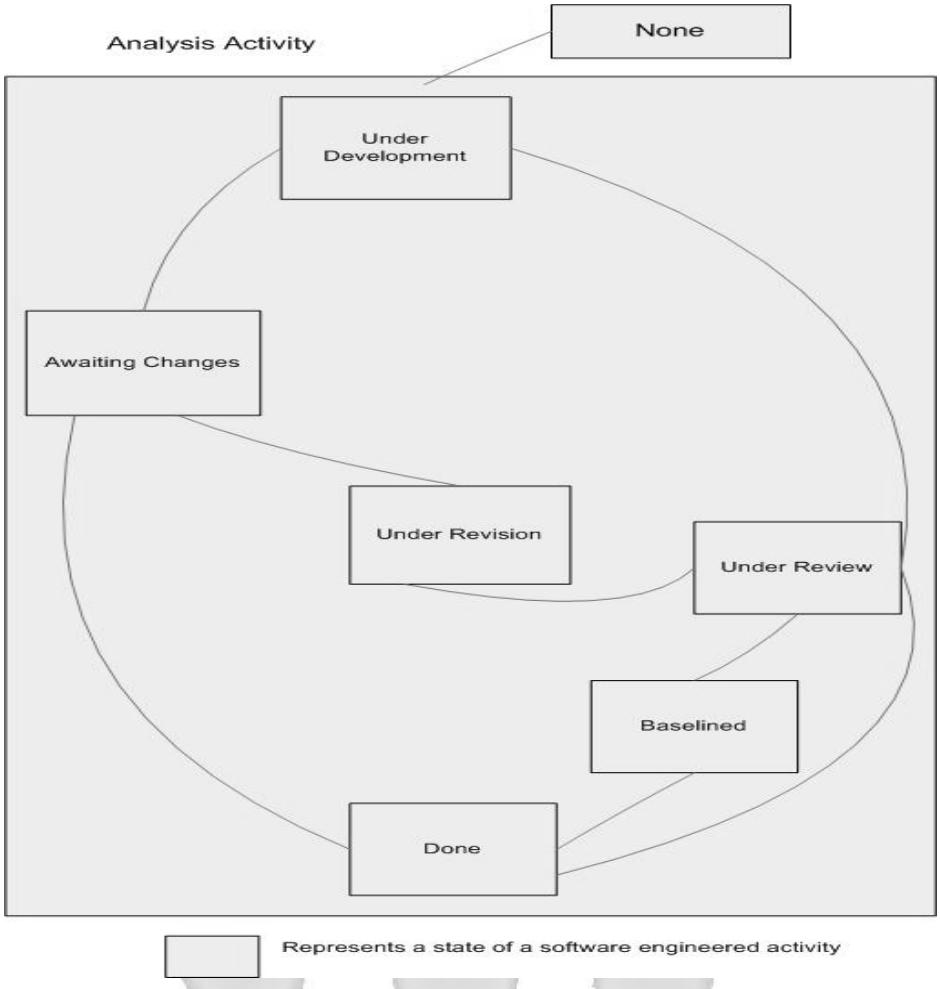


Fig. 2.4: One element of concurrent process model

The concurrent process model is often used as the paradigm for the development of client/server applications. A client/server system is composed of a set of functional components. When applied to client/server, the concurrent process model defines activities in two dimensions a system dimension and component dimension. System level issues are addressed using three activities, design assembly, and use. The component dimension addressed with two-activity design and realization. Concurrency is achieved in two ways; (1) System and component activities occur simultaneously and can be modeled using the state – oriented approach (2) a typical client

server application is implemented with many components, each of which can be designed and realized concurrently.

The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software-engineering activities to a sequence of events, it defines a net work of activities. Each activity on the network exists simultaneously with other activities. Events generated within a given activity or at some other place in the activity network trigger transitions among the sates of an activity.

Component based development model:

This model incorporates the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model composes applications from prepackaged software components called classes.

Classes created in past software engineering projects are stored in a class library or repository. Once candidate classes are identified, the class library is searched to determine if these classes already exist. If they do, they are extracted from the library and reused. If a candidate class does not reside in the library, it is engineered using object-oriented methods. The first iteration of the application to be built is then composed using classes extracted from the library and any new classes built to meet the unique needs of the application. Process flow then returns to the spiral and will ultimately re-enter the component assembly iteration during subsequent passes through the engineering activity.

The component based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits although it is very much dependent on the robustness of the component library.

2.8 Hacking

The growing dependence of society on software also places tremendous social responsibilities on the shoulders of software engineers and their managers. When the software is being used to monitor the health of patients, control nuclear power plants, apply the breaks in an automobile, transfer billions of dollars in an instant, launch missiles, or navigate an

airplane, it is not simply good engineering to build reliable software; it is also the engineer's ethical responsibilities to do so.

Program defects are not merely inconvenient "bugs" or interesting technical puzzles to be captured, but potentially serious business-or life-threatening errors. Building reliable software is technical objective of the software engineer, but it also has ethical and social implications that must guide the actions of a serious professional. In this light, " Hacking", i.e., inserting "play full" bugs into programs, creating viruses, writing quick and dirty code just to meet a schedule or a market window, shipping defective software, and even shipping software that works but does not meet the agreed upon specifications is unethical.

Self Assessment Questions

7. The _____ can be represented schematically as a series of major technical activities, tasks, and their associated states.
8. _____ model incorporates the characteristics of the spiral model.
9. Inserting 'play full' bugs into programs and similar activities is called _____.

2.9 Summary

- Software engineering is a discipline that integrates process, methods and tools for the development of computer software.
- The software process models consist of activities, which are involved, in developing software products. Basic activities are software specification, development, validation and evolution.
- The linear sequential model of the software process considers each process activity as a separate and discrete phase.
- The evolutionary development model such as iterative, increment model of the software process treats specification, development and validation as concurrent activities.

2.10 Terminal Questions

1. Explain the Serial or Linear Development Model in detail.
2. Give various definitions of Software Engineering.
3. What is Iterative Development Model? Explain in detail.
4. What is Hacking?

2.11 Answers

Self Assessment Questions

1. Software Engineering
2. 60%, 40%
3. Maintenance
4. Serial or Linear Development
5. Software Design
6. Incremental Model
7. Concurrent Process Model
8. Component Based Development
9. Hacking

Terminal Questions

1. This Model is also called as 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 support. (Refer section 2.4)
2. Software Engineering is the systematic approach to the development, operation, maintenance and retirement of software. This is the definition as per IEEE. (Refer section 2.3)
3. The iterative enhance model counters the third limitation of the waterfall model and tries to combine a benefit of both prototyping and the waterfall model. (Refer section 2.5)
4. inserting “play full” bugs into programs, creating viruses, writing quick and dirty code just to meet a schedule or a market window, shipping defective software, and even shipping software that works but does not meet the agreed upon specifications is called Hacking. (Refer section 2.8)

Unit 3 Software Reliability

Structure:

3.1 Introduction

Reliability is the most important dynamic characteristic of almost all software systems. Unreliable software results in high costs for end-users. Developers of unreliable systems may acquire a bad reputation for quality and lose future business opportunities.

The **Reliability of a software** system is a measure of how well users think it provides the services that they require. Reliability is usually defined as the probability of failure-free operation for a specified time in a specified environment for a specific purpose. Say it is claimed that software installed on an aircraft will be 99.99% reliable during an average flight of five hours. This means that a software failure of some kind will probably occur in one flight out of 10000.

A formal definition of reliability may not equate to user's experience of the software. The difficulty in relating such a figure to user's experience arises because it does not take the nature of the failure into account. A user does not consider all services to be of equal importance. A system might be thought of as unreliable if it ever failed to provide some critical service. For example, say a system was used to control braking on an aircraft but failed to work under a single set of very rare conditions. If an aircraft crashed because of these failure conditions, pilots of similar aircraft would regard the software as unreliable.

There is a general requirement for more reliable systems in all application domains. Customers expect their software to operate without failure to be available when it is required. Improved programming techniques, better programming languages and better quality management have led to very significant improvements in reliability for most software. However, for some systems, such as those which control unattended machinery, these 'normal' techniques may not be enough to achieve the level of reliability required, In these cases special **programming techniques may be necessary to achieve the required reliability.** Some of these techniques are discussed in this chapter.

The concept on **Software reuse** has been included in the following section. Because improved reliability is one of the benefits of reuse. Software components are not just used in one system but are tried and tested in a variety of different environments. Design and implementation forms are discovered and eliminated so the reusable components contain few errors. Although absolute reliability specification is impossible, reusable components may have an associative quality certification. This allows re-users to incorporate them with confidence in their systems.

Objectives:

After studying this unit, you should be able to:

- explain the meaning of software reliability
- describe reliability metrics
- discuss statistical analysis and programming for reliability

3.2 Software Reliability

Software reliability is a function of the number of failures experienced by a particular user of that software. A software failure occurs when the software is executing. It is a situation in which the software does not deliver the service expected by the user. Software failures are not the same as software faults although these terms are often used interchangeably.

Formal specifications and proof do not guarantee that the software will be reliable in practical use. The reasons for this are:

(1) **The specifications may not reflect the real requirements of system users** many failures experienced by users were a consequence of specification errors and omissions, which could not be detected by

formal system specification. It may even be the case that the opaqueness of formal notations makes it more difficult for users to establish whether or not a system meets their real requirements.

(2) **The proof may contain errors** Program proofs are large and complex so, like large and complex programs, they usually contain errors.

(3) **The Proof may assume a usage pattern, which is incorrect**. If the system is not used as anticipated, the proof may be invalid.

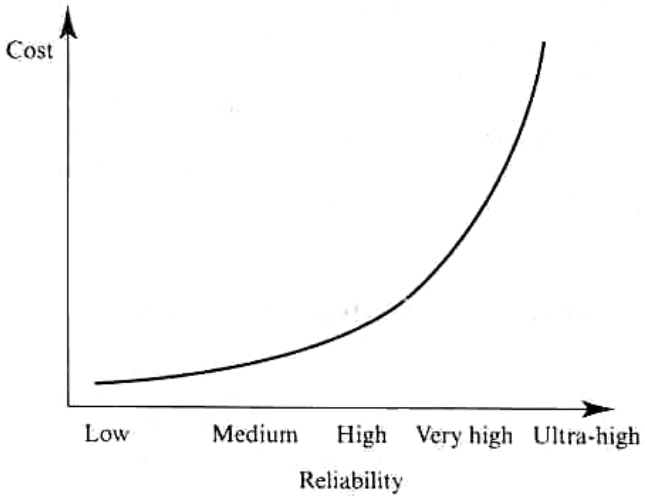


Fig. 3.1: Cost Vs Reliability

Because of additional design, implementation and validation overheads, increasing reliability can dramatically increase development costs. Figure 3.1 shown above is the relationship between costs and incremental improvements in reliability.

If it is possible to measure if a system is 100% reliable as this would require an amount of time equal to the lifetime of the system. However, as reliability requirements increase, system costs usually rise exponentially. This is mostly due to the need of redundant hardware and a vast increase in testing costs to check that the required reliability has been achieved. As discussed some specifications, which call for, ultra-reliable systems are unrealistic. The number of tests required to validate these specifications cannot be carried out in a reasonable time.

There is, of course, an efficiency penalty, which must be paid for increasing reliability. Reliable software must include extra, often redundant, code to perform the necessary checking for exceptional conditions. This reduces program execution speed and increases the amount of store required by the program. Reliability should always take precedence over efficiency for the following reasons:

1) **Computers are now cheap and fast:** There is little need to maximize equipment usage. Paradoxically, however, faster equipment leads to increasing expectations on the part of the user so efficiency considerations cannot be completely ignored.
2) **Unreliable software is liable to be discarded by users:** If a company attains a reputation for unreliability because of single unreliable product, it is likely to affect future sales of all of that company's products.
3) **System failure costs may be enormous:** For some applications, such a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system.
4) **Unreliable systems are difficult to improve:** It is usually possible to tune an inefficient system because most execution time is spent in small program sections. An unreliable system is more difficult to improve as unreliability tends to be distributed throughout the system.
5) **Inefficiency is predictable:** Programs take a long time to execute and users can adjust their work to take this into account. Unreliability, by contrast, usually surprises the user. Software that is unreliable can have hidden errors which can violate system and user data without warning and whose consequences are not immediately obvious. For example, a fault in a CAD program used to design aircraft might not be discovered until several plane crashes occur.
6) **Unreliable systems may cause information loss:** Information is very expensive to collect and maintains; it may sometimes be worth more than the computer system on which it is processed. A great deal of effort and money is spent duplicating valuable data to guard against data corruption caused by unreliable software.

The software process used to develop that product influences the reliability of the software product. A repeatable process, which is oriented towards

defect avoidance, is likely to develop a reliable system. However, there is not a simple relationship between product and process reliability.

Users often complain that systems are unreliable. This may be due to poor software engineering. However, a common cause of perceived unreliability is incomplete specifications. The system performs as specified but the specifications do not set out how the software should behave in exceptional situations. As professionals, software engineers must do their best to produce reliable systems, which take meaningful and useful actions in such situations.

3.3 Software Reliability Metrics

Metrics which have been used for software reliability specification are shown in Figure 3.2 shown below .The choice of which metric should be used depends on the type of system to which it applies and the requirements of the application domain. For some systems, it may be appropriate to use different reliability metrics for different sub-systems.

Metric	Explanation	Example systems
POFOD Probability of failure on demand	This is a measure of the likelihood that the system will fail when a service request is made. For example, a POFOD of 0.001 means that 1 out of 1000 service requests may result in failure.	Safety-critical and non-stop systems, such as hardware control systems.
ROCOF Rate of failure occurrence	This is a measure of the frequency of occurrence with which unexpected behaviour is likely to occur. For example, a ROCOF of 2/100 means that 2 failures are likely to occur in each 100 operational time units. This metric is sometimes called the failure intensity.	Operating systems, transaction processing systems.
MTTF Mean time to failure	This is a measure of the time between observed system failures. For example, an MTTF of 500 means that 1 failure can be expected every 500 time units. If the system is not being changed, it is the reciprocal of the ROCOF.	Systems with long transactions such as CAD systems. The MTTF must be greater than the transaction time.
AVAIL Availability	This is a measure of how likely the system is to be available for use. For example, an availability of 0.998 means that in every 1000 time units, the system is likely to be available for 998 of these.	Continuously running systems such as telephone switching systems.

Fig. 3.2: Reliability metrics

In some cases, system users are most concerned about how often the system will fail, perhaps because there is a significant cost in restarting the system. In those cases, a metric based on a rate of failure occurrence (ROCOF) or the mean time to failure should be used.

In other cases, it is essential that a system should always meet a request for service because there is some cost in failing to deliver the service. The number of failures in some time period is less important. In those cases, a metric based on the probability of failure on demand (POFOD) should be used. Finally, users or system operators may be mostly concerned that the system is available when a request for service is made. They will incur some loss if the system is unavailable. Availability (AVAIL) takes into account the repair or restart time.

There are three kinds of measurement, which can be made when assessing the reliability of a system:

1) The number of system failures given a number of systems inputs. This is used to measure the POFOD.
2) The time (or number of transaction) between system failures. This is used to measure ROCOF and MTTF.
3) The elapsed repair or restart time when a system failure occurs. Given that the system must be continuously available, this is used to measure AVAIL.

Time is a factor in all of this reliability metrics. It is essential that the appropriate time units should be chosen if measurements are to be meaningful. Time units, which may be used, are calendar time, processor time or may be some discrete unit such as number of transactions.

Software reliability specification

System requirements documents, reliability requirements are expressed in an informal, qualitative, un-testable way. Ideally, the required level of reliability should be expressed quantitatively in the software requirement specification. Depending on the type of system, one or more of the metrics discussed in the previous section may be used for reliability specifications. Statistical testing techniques (discussed later) should be used to measure the system reliability. The software test plan should include an operational profile of the software to assess its reliability.

The steps involved in establishing a reliability specification are as follows:

1) For each identified sub-system, identify the different types of system failure, which may occur and analyze the consequences of these failures.
2) From the system failure analysis, partition failures into appropriate classes. A reasonable starting point is to use the failure types shown in Figure 3.3 below. For each failure class identified, define the reliability requirement using the appropriate reliability metric. It is not necessary to use the same metric for different classes of failure. For example, where a failure requires some intervention to recover from it, the probability of that failure occurring on demand might be the most appropriate metric. When automatic recover is possible and the effect of the failure is some user inconvenience, ROCOF might be more appropriate.

Failure Class	Description
Transient	Occurs only with certain inputs
Permanent	Occurs with all inputs
Recoverable	System can recover without operator intervention
Unrecoverable	Operator intervention needed to recover from failure
Non-corrupting	Failure does not corrupt system state or data
Corrupting	Failure corrupts system state or data.

Fig. 3.3: Failure classification

The cost of developing and validating a reliability specification for software system is very high.

Statistical testing

Statistical testing is a software testing process in which the objective is to measure the reliability of the software rather than to discover software faults. It users different test data from defect testing, which is intended to find faults in the software.

The steps involved in statistical testing are:

1) Determine the operational profile of the software. The operational profile is the probable pattern of usage of the software. This can be determined by analyzing historical data to discover the different classes of input to the program and the probability of their occurrence.

2) Select or generate a set of test data corresponding to the operational profile.
3) Apply these test cases to the program, recording the amount of execution time between each observed system failure. It may not be appropriate to use raw execution time. As discussed in the previous section, the time units chosen should be appropriate for the reliability metric used.
4) After a statistically significant number of failures have been observed, the software reliability can then be computed. This involves using the number of failures detected and the time between these failures to computer the required reliability metric.

This approach to reliability estimation is not easy to apply in practice. The difficulties, which arise, are due to:

- Operational profile uncertainty;
- High cost of operational profile generation;
- Statistical uncertainty when high reliability is specified.

Self Assessment Questions

1. The _____ of a software system is a measure of how well users think it provides the services that they require.
2. MTTF stands for _____.
3. _____ testing is a software testing process in which the objective is to measure the reliability of the software rather than to discover software faults.

3.4 Programming for Reliability

There is a general requirement for more reliable systems in all application domains. Customers expect their software to operate without failures and to be available when it is required. Improved programming techniques, better programming languages and better quality management have led to very significant improvements in reliability for most software. However, for some systems, such as those, which control unattended machinery, these 'normal' techniques may not be enough to achieve the level of reliability required. In these cases, special programming techniques may be necessary to achieve the required reliability. Some of these techniques are discussed in this chapter.

Reliability in a software system can be achieved using three strategies:

- **Fault avoidance:** This is the most important strategy, which is applicable to all types of system. The design and implementation process should be organized with the objective of producing fault-free systems.
- **Fault tolerance:** This strategy assumes that residual faults remain in the system. Facilities are provided in the software to allow operation to continue when these faults cause system failures.
- **Fault detection:** Faults are detected before the software is put into operation. The software validation process uses static and dynamic methods to discover any faults, which remain in a system after implementation.

3.4.1 Fault avoidance

A good software process should be oriented towards fault avoidance rather than fault detection and removal. It should have the objective of developing fault-free software. Fault-free software means software, which conforms to its specification. Of course, there may be errors in the specification or it may not reflect the real needs of the user so fault-free software does not necessarily mean that the software will always behave as the user wants.

Fault avoidance and the development of fault-free software rely on:

1. The availability of a precise system specification, which is an unambiguous description of what, must be implemented.
2. The adoption of an organizational quality philosophy in which quality is the driver of the software process. Programmers should expect to write bug-free program.
3. The adoption of an approach to software design and implementation which is based on information hiding and encapsulation and which encourages the production of readable programs.
4. The use of a strongly typed programming language so that possible errors are detected by the language compiler.
5. Restriction on the use of programming construct, such as pointers, which are inherently error-prone.

Achieving fault-free software is virtually impossible if low-level programming Languages with limited type checking are used for program development.

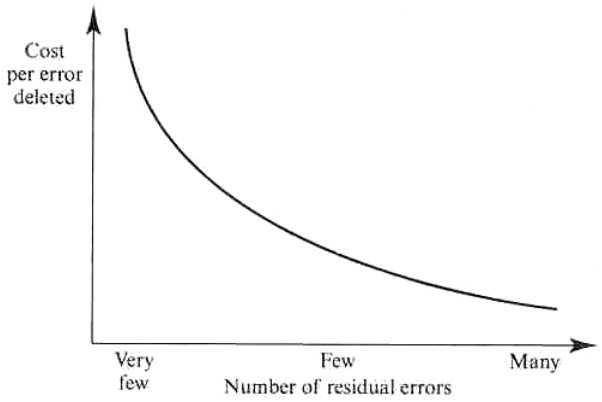


Fig. 3.4: The increasing cost of residual fault of removal

We must be realistic and accept that human errors will always occur. Faults may remain in the software after development. Therefore, the development process must include a validation phase, which checks the developed software for the presence of faults. This validation phase is usually very expensive. As faults are removed from a program, the cost of finding and removing remaining faults tends to rise exponentially. As the software becomes more reliable, more and more testing is required to find fewer and fewer faults.

Structured programming and error avoidance

Structured programming is term which is to mean programming without using go to statements, programming using only while loops and if statements as control constructs and designing using a top-down approach. The adoption of structured programming was an important milestone in the development of software engineering because it was the first step away from an undisciplined approach to software development.

Go to statement was an inherently error-prone programming construct. The disciplined use of control structures force programmers to think carefully about their program. Hence they are less likely to make mistakes during development. Structured programming means programs can be read sequentially and are therefore easier to understand and inspect. However, avoiding unsafe control statements is only the first step in programming for reliability.

Faults are less likely to be introduced into programs if the use of these constructs is minimized. These constructs include:

1) **Floating-point numbers:** Floating-point numbers are inherently imprecise. They present a particular problem when they are compared because representation imprecision may lead to invalid comparisons. Fixed-point numbers, where a number is represented to a given number of decimal places, are safer as exact comparisons are possible.
2) **Pointer:** *Pointers* are low-level constructs, which refer directly to areas of the machine memory. They are dangerous because errors in their use can be devastating and because they allow 'aliasing'. This means the same entity may be referenced using different names. Aliasing makes programs harder to may be referenced using different names. Aliasing makes programs harder to understand so that errors are more difficult to find. However, efficiency requirements mean that it is often impractical to avoid the use of pointers.
3) **Dynamic memory allocation:** Program memory is allocated at run-time rather than compile-time. The danger with this is that the memory may not be de-allocated so that the system eventually runs out of available memory. This can be a very subtle type of errors to detect as the system may run successfully for a long time before the problem occurs.
4) **Parallelism:** Parallelism is dangerous because of the difficulties of predicting the subtle effects of timing interactions between parallel processes. Timing problems cannot usually e detected by program inspection and the peculiar combination of circumstances, which cause a timing problem, may not result during system testing. Parallelism may be unavoidable but its use should be carefully controlled to minimize inter-process dependencies. Programming language facilities, such as Ada tasks, help avoid some of the problems of parallelism as the compiler can detect some kinds of programming errors.
5) **Recursion:** Recursion is the situation in which a subroutine calls itself or calls another subroutine, which then calls the calling subroutine. Its use can result in very concise programs but it can be difficult to follow the logic of recursive programs. Errors in using recursion may result in the allocation of all they system's memory as temporary stack variables are created.

6) **Interrupts:** Interrupts are a means of forcing control to transfer to a section of code irrespective of the code currently executing. The dangers of this are obvious as the interrupt may cause a critical operation to be terminated.

3.4.2 Fault tolerance

A fault-tolerant system can continue in operation after some system failures have occurred. Fault tolerance is needed in situations where system failure would cause some accident or where a loss of system operation would cause large economic losses. For example, the computers in an aircraft must continue in operation until the aircraft has landed; the computers in an traffic control system must be continuously available.

Fault-tolerance facilities are required if the system is to failure. There are four aspects to fault tolerance.

1. **Failure detection:** The system must detect a particular state combination has resulted or will result in a system failure.
2. **Damage assessment:** The parts of the system state, which have been affected by the failure, must be detected.
3. **Fault recovery:** The system must restore its state to a known 'safe' state. This may be achieved by correcting the damaged state or by restoring the system the system to a known 'safe' state. Forward error recovery is more complex as it involves diagnosing system faults and knowing what the system state should have been had the faults not caused a system failure.
4. **Fault repair:** This involves modifying the system so that the fault does not recur. In many cases, software failures are transient and due to a peculiar combination of system inputs. No repair is necessary as normal processing can resume immediately after fault recovery. This is an important distinction between hardware and software faults.

There has been a need for many years to build fault-tolerant hardware. The most commonly used hardware fault-tolerant technique is based around the notion of triple-modular redundancy (TMR) shown in figure 3.5. The hardware unit is replicated three (or sometimes more) times. The output from each unit is compared. If one of the units fails and does not produce the same output as the other units, its output is ignored. The system functions with two working units.

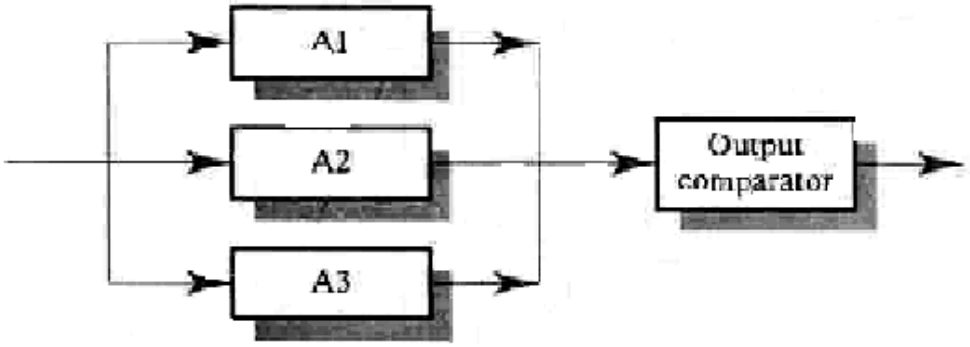


Fig. 3.5: Triple modular redundancy to cope with hardware failure

The weakness of both these approaches to fault tolerance is that they are based on the assumption that the specification is correct. They do not tolerate specification errors.

There have been two comparable approaches to the provision of software fault tolerance. Both have been derived from the hardware model where a component is replicated.

(1) **N-version programming:** Using a common specification, the software system is implemented in a number of different versions by different teams. These versions are executed in parallel. Their outputs are compared using a voting system and inconsistent outputs are rejected. At least three versions of the system should be available.

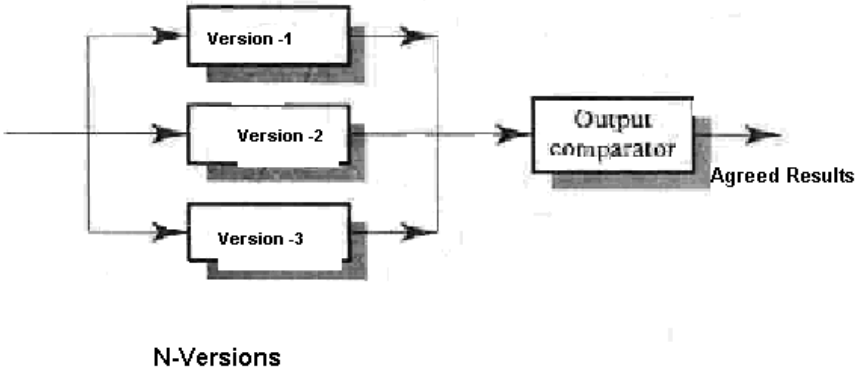


Fig. 3.6: N-version programming

(2) **Recovery Blocks:** this is a finer grain approach to fault tolerance. Each program component includes a test to check if the component has executed successfully. It also includes alternative code, which allows the

system to back-up and repeat the computation if the test detects a failure. Unlike N-version programming, the implementation is different rather than independent implementation of the same specification. They are executed in sequence rather than in parallel.

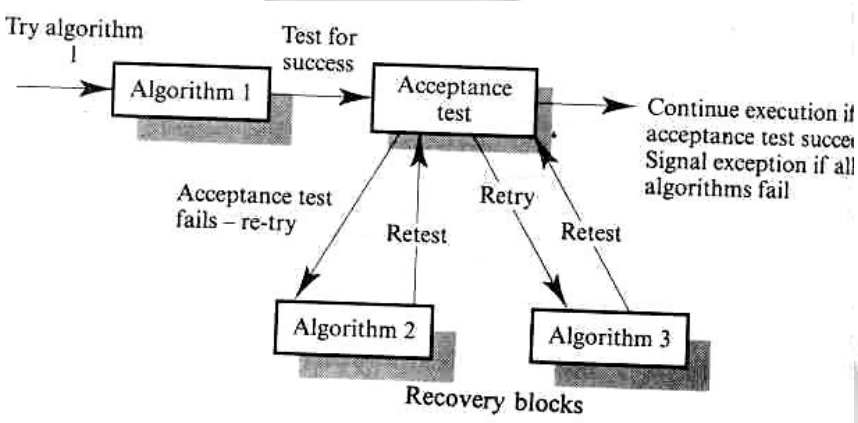


Fig. 3.7: Recovery blocks

Exception Handling

When an error of some kind or an unexpected event occurs during the execution of a program, this is called an exception. Exceptions may be caused by hardware or software errors. When an exception has not been anticipated, control is transferred to system exceptions handling mechanism. If an exception has been anticipated, code must be included in the program to detect and handle that exception.

Most programming languages do not include facilities to detect and handle exceptions. The normal decision constructs (if statements) of the language must be used to detect the exception and control constructs used to transfer control to exception occurs in a sequence of nested procedure calls, there is not easy way to transmit it from one procedure to another.

Consider example as shown in figure 3.8 below a number of nested procedure calls where procedure A calls procedure B which calls procedure C. If an exception occurs during the execution of C this may be so serious that execution of B cannot continue. Procedure B has to return immediately to Procedure A, which must also be informed that B has terminated abnormally and that an exception has occurred.

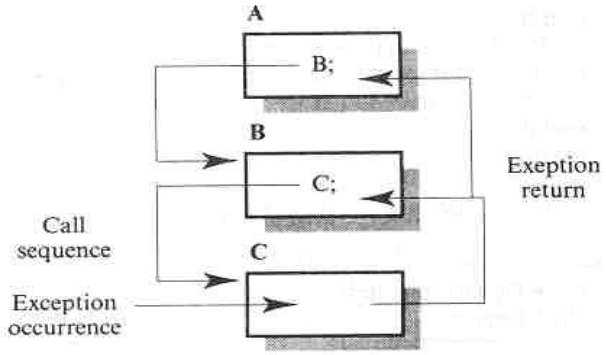


Fig. 3.8: Exception return in embedded procedure calls

An exception handler is something like a case statement. It states exception names and appropriate actions for each exception.

Table 3.1 below illustrates the use of exceptions and exception handling. These program fragments show the design of a temperature controller on a food freezer. The required temperature may be set between –18 and –40 degrees Celsius. Food may start to defrost and bacteria become active at temperatures over – 18 degrees. The control system maintains this temperature by switching a refrigerant pump on and off depending on the value of a temperature sensor. If the required temperature cannot be maintained, the controlled sets off an alarm.

```
void Control_freezer ( const float Danger_temp)
{
    float Ambient_temp ;
    // try means exceptions will be handled in this block
    // Assume that Sensor, Temperature_dial and Pump are
    //objects which have been declared elsewhere
    try {
        while (true) {
            Ambient_temp = Sensor.Get_temperature () ;
            if (Ambient_temp > Temperature_dial.Setting () )
                if (Pump.Status () == off)
                {
                    Pump.Switch (on) ;
                    Wait (Cooling_time) ;
                }
                else
                    if (Pump.Status () == on)
                        Pump.Switch (off) ;
            if ( Ambient_temp > Danger_temp )
                    throw Freezer_too_hot ( ) ;
        } // end of while loop
    } // end of exception handling try block
    // catch indicates the exception handling code.
    catch ( Freezer_too_hot )
        Alarm.Activate () ;
}
```

Table 3.1: Exceptions in a freezer temperature controller(C++)

The temperature of the freezer is discovered by interrogating an object called Sensor and the required temperature by inspecting an object called the exceptions Freezer_too_hot and Control_problem and the type FREEZER_TEMP are declared. There are no built-in exceptions in C++ but other information is declared in a separate header file.

The temperature controller tests the temperature and switches the pump as required. If the temperature is too hot, it transfers control to the exception handler, which activates an alarm.

In C++, Once an exception has been, it is not re-thrown.

Defensive programming

Defensive programming is an approach to program development whereby programmers assume that there may be undetected faults or inconsistencies in their programs. Redundant code is incorporated to check the System State after modifications and to ensure that the state change is consistent. If inconsistencies are detected, the state change is retracted or the state is restored to a known correct state.

Defensive programming is an approach to fault tolerance, which can be carried out without a fault-tolerant controller. The techniques used, however, are fundamental to the activities in the fault tolerance process, namely detecting a failure, damage assessment, and recovering from that failure.

Failure prevention

Programming languages such as Ada and C++ allow many errors which cause state corruption and system failure to be detected at compile-time. The compiler can detect those problems which uses the strict type rules of the language. Compiler checking is obviously limited to static values but the compiler can also automatically add code to a program to perform run-time checks.

Damage assessment

Damage assessment involves analyzing the system state to gauge the extent of the state corruption. In many cases, corruption can be avoided by checking for fault occurrence before finally committing a change of state. If a fault is detected, the state change is not accepted so that no damage is caused. However, damage assessment may be needed when a fault arises

because a sequence of state changes (all of which are individually correct) causes the system to enter an incorrect state.

The role of the damage assessment procedures is not to recover from the fault but to assess what parts of the state space have been affected by the fault. Damage can only be assessed if it is possible to apply some 'validity function', which checks if the state is consistent. If inconsistencies are found, these are highlighted or signaled in some way.

Other techniques which can be used for fault detection and damage assessment are dependent on the system state representation and on the application . Possible methods are:

- The use of checksums in data exchange and check digits in numeric data;
- The use of redundant links in data structures which contain pointers;
- The use of watchdog timers in concurrent systems.

A checksum is a value that is computed by applying some mathematical function to the data. The function used should give a unique value for the packet of data, which is exchanged. The sender who applies the checksum function to the data and appends that function value to the data computes this checksum. The receiver applies the same function to the data and compares the checksum values. If these differ, some data corruption has occurred.

When linked data structures are used, the representation can be made redundant by including backward pointers. That is, for every reference from A to B, there exists a comparable reference from B to A. It is also possible to keep count of the number of elements in the structure. Checking can determine whether or not all pointers have an inverse value and whether or not the stored size and the computed structure size are the same.

When processes must react within a specific time period, a watch-dog timer may be installed. A watch-dog timer is a timer which must be reset by the executing process after its action is complete. It is started at the same time as a process and times the process execution. If, for some reason the process fails to terminate, the watch-dog timer is not reset. The controller can therefore detect that a problem has arisen and take action to force process termination.

Fault recovery

Fault recovery is the process of modifying the state space of the system so that the effects of the fault are minimized. The system can continue in operation, perhaps in same degraded form. Forward recovery involves trying to correct the damaged System State. Backward recovery restores the System State to a known 'correct' state.

There are two general situations where forward error recovery can be applied:

(1) **When coded is corrupted** The use of coding techniques which add redundancy to the data allows errors to be corrected as well as detected.

(2) **When linked structures are corrupted** if forward and backward pointers are included in the data structure, the structure can be recreated if enough pointers remain uncorrupted. This technique is frequently used for file system and database repair.

Backward error recovery is a simpler technique, which restores the state to a known safe state after an error has been detected. Most database systems include backward error recovery. When a user initiates a database computation a transaction is initiated. Changes made during that transaction are not immediately incorporated in the database. The database is only updated after the transaction is finished and no problems are detected. If the transaction fails, the database is not updated.

Design by Contract

Meyer suggests an approach to design, called design by contract, to help ensure that a design meets its specifications. He begins by viewing software system as a set of communicating components whose interaction is based on a precisely defined specification of what each component is supposed to do. These specifications, called **contracts,** govern how the component is to interact with other components and systems. Such specification cannot guarantee correctness, but it forms a good basis for testing and validation.

Contract is written between two parties when one commissions the other for a particular service or product. Each party expects some benefit for some obligation; the supplier produces a service or product in a given period of time in exchange for money, and the client accepts the service or product for the money. The contract makes the obligation and benefits explicit.

Mayer applies the notion of a contract to software. A software component, called a **client,** adopts a strategy to perform a set of tasks, t1, t2,……tn. In turn, each nontrivial subtask, it is executed when the client calls another component, the **supplier**, to perform it. That is a contract between the two components to perform the sub-task. Each contract covers mutual obligation (called **preconditions**), benefits (called **post-conditions**), and consistency constraints (called **invariant).** Together, these contract properties are called assertions.

For example, suppose the client component has a table where each element is identified by a character string used as a key. Our supplier's component's task is to insert an element from the table into a dictionary of limited size. We can describe the contract between the two components in the following way.

1. The client component ensures that the dictionary is not full and that the key is nonempty.
2. The supplier component records the element in table.
3. The client component accesses the updated table where the element appears.
4. If the table is full or the key is empty, no action is taken.

Self Assessment Questions

1. What do you mean by fault avoidance.
2. Explain why fault tolerance facilities are required if the system is failure.
3. Generally in which situation fault recovery process is applied.

3.5 Software Reuse

The design process in most engineering disciplines is based on component reuse. Mechanical or electrical engineers do not specify a design in which every component has to be manufactured specially. They base their design on components that have been tried and tested in other systems. These components obviously include small components such as nuts and bolts. However, they may also be major sub-systems such as engines, condensers or turbines. By contrast, software system design usually assumes that all components are to be implemented specially for the system being developed. Apart from libraries such as window system libraries, there is no common base of reusable software components, which

is known by all software engineers. However, this situation is slowly changing. We need to reuse our software assets rather than redevelop the same software again and again. Demands for lower software production and maintenance costs along with increased quality can only be met by widespread and systematic software reuse. Component reuse, of course, does not just mean the reuse of code. It is possible to reuse specifications and designs. The potential gains from reusing abstract products of the development process, such as specifications, may be greater than those from reusing code components. Code contains low-level details, which may specialize it to such an extent that it cannot be reused. Designs or specifications are more abstract and hence more widely applicable.

The reuse of software can consider at a number of different levels:.

1) **Application system reuse:** The whole of an application system may be reused. The key problem here is ensuring that the software is portable; it should execute on several different platforms.
2) **Sub-system reuse:** Major sub-systems of an application may be reused. For example, a pattern-matching system developed as part of a text processing system may be reused in a database management system.
3) **Module or object reuse:** Components of a system representing a collection of functions may be reused. For example, an Ada package or a C++ object implementing a binary tree may be reused in different applications.
4) **Function reuse:** Software components, which implement a single function, such as a mathematical function, may be reused.

Four aspects of software reuse:

5) **Software development with reuses:** What are the advantages and problems of developing software with reusable components? How must software process evolve to incorporate reuse?
6) **Software development for reuse:** How can software components are generalized so that they are usable across a range of systems?
7) **Generator based reuse:** How do application generators support the reuse of domain concepts?
8) **Application system reuses:** How making them available on a range of machines reuse can entire application systems? What implementation strategies should be used to develop portable software?

3.5.1 Software development with reuse

Software development with reuse is an approach to development, which tries to maximize the reuse of existing components. An obvious advantage of this approach is that overall development costs should be reduced. Fewer software components need be specified, designed, implemented and validated. However, cost reduction is only one potential advantage of reuse. Systematic reuse in the development process offers further advantages:

1) **System reliability increased:** Reused components, which have been exercised in working systems, should be more reliable than new components. These components have been tested in operational systems and have therefore been exposed to realistic operating conditions.
2) **Overall process risk is reduced:** If a component exists, there is less uncertainty in the costs of reusing that component than in the costs of development. This is an important factor for project management as it reduces the uncertainties in project cost estimation. This is particularly true when relatively large components such as sub-systems are reused.
3) **Effective use can be made of specialists:** Instead of application specialists doing the same work on different projects, these specialists can develop reusable components which encapsulate their knowledge.
4) **Organizational standards can be embodied in reusable components:** Some standard, such as user interface standards, can be implemented as a set of standard components. For example, reusable components may be developed to implement menus in a user interface. All applications present the same menu formats to users. The use of standard user interfaces improves reliability, as users are less likely to make mistakes when presented with a familiar interface.
5) **Software development time can be reduced:** Bringing a system to market as early as possible is often more important than overall development costs. Reusing component speeds up system production because both development and validation time should be reduced.

Incorporating a specific reuse activity as shown in figure 3.9 below can incorporate reuse into the systems development process. The system designer completes a high-level design and specifications of the components of that design. These specifications are used to find

components to reuse. These may be incorporated at the architectural level or at more detailed design levels.

Fig. 3.9: Reuse in a standard development process

Although this model can result in significant reuse, it contrasts with the approach adopted in other engineering disciplines where reusability drives the design process. Rather than design then search for reusable components, engineers first search for reusable components. They base their design on these components.

There are three conditions for software development with reuse:

1. It must be possible to find appropriate reusable components. Organizations need a base properly catalogued and documented reusable component. The cost of findings an appropriate component in this catalogue must be relatively low.
2. The re-user of the components must have confidence that the components will behave as specified and will be reliable. Ideally, all components in an organization's catalogue should be certified to confirm that they have reached some quality standards.
3. The components must have associated documentation to help the re-user understand them and adapt them to a new application. The documentation should include information about where components have been reused and any reuse problems, which have been found.

Other difficulties in introducing development with reuse are:

1) It is difficult to quantify what the cost reductions might be as there are usually costs associated with reuse. Software components have to be discovered in a library, understood and sometimes adapted to work in a new environment. These reuse costs may sometimes be greater than the cost of re-implementing the component.
2) CASE toolsets do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system.
3) Some software engineers sometimes prefer to rewrite components, as they believe that they can improve on the reusable component. This is a

natural consequence of an educational process, which concentrates on original software development rather than reuse.

4) Our current techniques for classifying, cataloguing and retrieving software components are immature. Engineers must be reasonably confident of finding a component in the library before they will routinely include a component search as part of their normal development process.

3.5.2 Software development for reuse

Systematic reuse requires a properly catalogued and documented base of reusable components. Once a component has been developed and used in a system, it can be reused without change. More commonly, however, it will be necessary to adapt the component in some way to take account of the particular requirements for the system being developed. Below figure 3.10 shows the process of improving the reusability of a component.

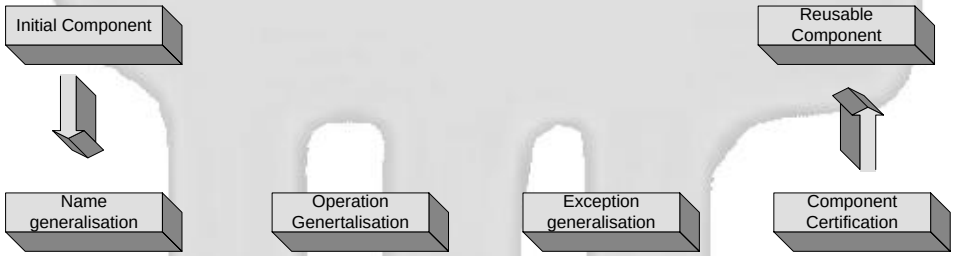


Fig. 3.10: The process of reusability enhancement

Adapting a component to make it reusable may involve making different types of changes:

(1) **Name generalization** the names used in the component may be modified so that they are neutral rather than a direct reflection of some specific application entity.

(2) **Operation generalization** this may involve adding operations to a component or removing operations, which are very specific to some application domain.

(3) **Exception generalization** this may involve checking each component to see which exceptions it might generate and including these exceptions in the component interface.

After generalization, the quality of the generalized component should be checked. This may require program inspections or testing. Ideally, the test data for a component should be made a available to re-users so that it also may be reused. The component may be certified as having reached the required quality standards.

3.5.3 Generator – based reuse

An alternative to the component-oriented view of reuse is the generator view. In this approach to reuse, reusable knowledge is captured in a program generator system, which can be programmed in a domain-oriented language. The application description specifies, in an abstract way, which reusable components are to be used, how they are to be combined and their parameterization. Using this information, an operational software system can be generated. Figure 3.11 illustrates this approach to reuse.

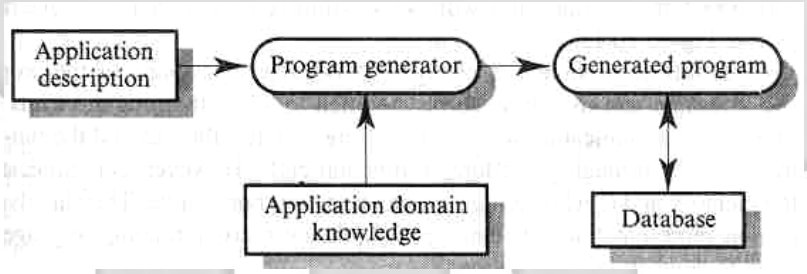


Fig. 3.11: Reuse of domain knowledge through application generation

Generator-based reuse is cost-effective but depends on identifying stereotypical domain abstractions.

3.5.4 Application system portability

A special case of software reuse is application system reuse where a whole application system is reused by implementing it across a range of different computers and operating systems. The problem here is not to discover components for reuse but to develop the system so that it is portable across different platforms.

Self Assessment Questions

4. A good software process should be oriented towards _____ rather than fault detection and removal.
5. The most commonly used hardware fault-tolerant technique is based around the notion of _____.

6. Occurring of an error of some kind or an unexpected event during the execution of a program is called an _____.

3.6 Summary

- The most important dynamic characteristic of most software systems is their reliability. The reason for this is that the costs of system failure often exceed the costs of developing the software system.
- Reliability specifications are often imperfect but software engineers are still responsible for producing reliable systems.
- Programs should not produce incorrect output, should not corrupt themselves or other programs and should take meaningful actions in unexpected situations.
- Reliability requirements should be defined quantitatively in the system requirement specification.
- Reliability in a program can be achieved by avoiding the introduction of faults and by including fault-tolerance facilities, which allow the system to remain operational after a fault, has caused a system failure.
- Software that is fault-tolerant can continue execution in spite of faults, which cause system failures.
- There are four aspects of program fault tolerance, namely failure detection, damage assessment, fault recovery and fault repair.
- Software reuse involves reusing existing components rather than developing them especially for an application. Systematic reuse can improve reliability, reduce management risk and reduce development costs.
- Abstract data types and objects are effective encapsulation of reusable components.
- Application system portability is a specialized form of reuse in which an entire application system is adapted for reuse on a different computer

3.7 Terminal Questions

1. What is Software Reliability? Explain.
2. Explain different Software Reliability Metrics.
3. How reliability in a software system can be achieved? Explain.
4. What are the different levels of software reuse?

3.8 Answers

Self Assessment Questions

1. Reliability
2. Mean Time To Failure
3. Statistical
4. Fault avoidance
5. Triple Modular Redundancy (TMR)
6. Exception

Terminal Questions

1. Software reliability is a function of the number of failures experienced by a particular user of that software. (Refer section 3.2)
2. Refer section 3.3.
3. Reliability in a software system can be achieved using three strategies:
   - **Fault avoidance:** This is the most important strategy, which is applicable to all types of system. The design and implementation process should be organized with the objective of producing fault-free systems.
   - **Fault tolerance:** This strategy assumes that residual faults remain in the system. Facilities are provided in the software to allow operation to continue when these faults cause system failures.
   - **Fault detection:** Faults are detected before the software is put into operation. The software validation process uses static and dynamic methods to discover any faults, which remain in a system after implementation. (Refer section 3.4)
4. Application system reuse, sub-system reuse, module or object reuse, function reuse. (Refer section 3.5)

Unit 4 Software Design Principles

Structure:

4.1 Introduction

The output of the requirement analysis process is a set of **System models** that present abstract description of the system to be developed. Methods based approaches to analysis are systematic ways of producing these systems model. These system models are based on computational concepts such as objects or functions rather than application domain concepts. Therefore they are an important bridge between the analysis and design processes.

Good **Software design** is the key to effective engineering. 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.

The primary objective of **Architectural design** is to develop a modular program structure and represent the control relationships between modules. In addition, architectural design melds program structure and data structure, defining interfaces that enable data to flow throughout the program.

Object-oriented design (OOD) transforms the analysis model created using object-oriented analysis into a design model that serves as a blueprint for software construction. Unlike conventional software design methods,

OOD results in a design that achieves a number of different levels of modularity. Major system components are organized into system-level "modules" called subsystems. Data and the operations that manipulate the data are encapsulated into objects – a modular form that is the building block of an OO system. In addition, OOD must describe the specific data organization of attributes and the procedural detail of individual operations.

A **Function-oriented design** strategy relies on decomposing the system into a set of interacting functions with a centralized system state shared by these functions. Functions may also maintain local state information but only for the duration of their execution.

Objectives:

After studying this unit, you should be able to:

- explain the software design principles
- follow the structured approach for the software design
- describe the design specifications and verification

4.2 System Models

Different types of system models are based on different approaches to abstraction. A data-flow model concentrates on the flow of data and the functional transformations on that data. It gives out details of the data structures. By contrast, an entity-relation model is intended to document the system data and its relationships without concern of the functions in the system.

Examples of the different types of system model, which might be produced as part of the analysis process and the notations used to represent these models, are:

- **A data-processing model:** Data flow diagrams may be used to show how data is processed at different stages in the system.
- **A composition model**: entity-relation diagram may be used to show how some entities in the system are composed of other entities.
- **A classification model:** Objects class/inheritance diagrams may be used to show how entities have common characteristics.
- **A stimulus-response model:** State transition diagrams may be used to show how the system reacts to internal and external events.

- **A process model:** Process models may be used to show the principal activities and deliverables involved in carrying out some process.

Among these three, widely used types of system models are Data-flow models, Semantic data models, object models, and the Data dictionaries, which can be used to support all kinds of system model.

4.2.1 Data-flow models

Data-flow model is a way of showing how data is processed by a system. At the analysis level, they should be used to model the way in which data is processed in the existing system. The notations used in these models represents functional processing, data stores and data movements between functions.

Data-flow models are used to show how data flows through a sequence of processing steps. The data is transformed at each step before moving on to the next stage. These processing steps or transformations are program functions when data-flow diagrams are used to document a software design. Figure 4.1 shows the steps involved in processing an order for goods (such as computer equipment) in an organization.

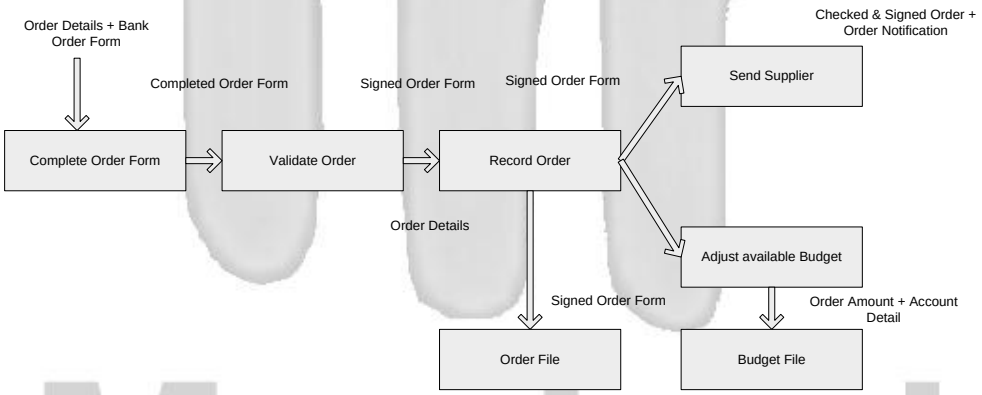


Fig. 4.1: Data flow diagrams of Order processing

The model shows how the order for the goods moves from process to process. It also shows the data stores that are involved in this process.

There are various notations used for data-flow diagrams. In figure rounded rectangles represent processing steps, arrow annotated with the data name represent flows and rectangles represent data stores (data sources). Data-

flow diagrams have the advantage that, unlike some other modeling notations, they are simple and intuitive. These diagrams are not a good way to describe sub-system with complex interfaces.

4.2.2 Semantic data models

The large software system makes use of a large database of information. In some cases, this database exists independently of the software system. In others, it is created for the system being developed. An important part of system modeling is to define the logical form of the data processed by the system. An approach to data modeling, which includes information about the semantics of the data, allows a better abstract model to be produced. Semantic data models always identify the entities in a database, their attributes and explicit relationship between them. Approach to semantic data modeling includes Entity-relationship modeling. Semantic data models are described using graphical notations. These graphical notations are understandable by users so they can participate in data modeling. The notations used are in figure 4.2 shown below.

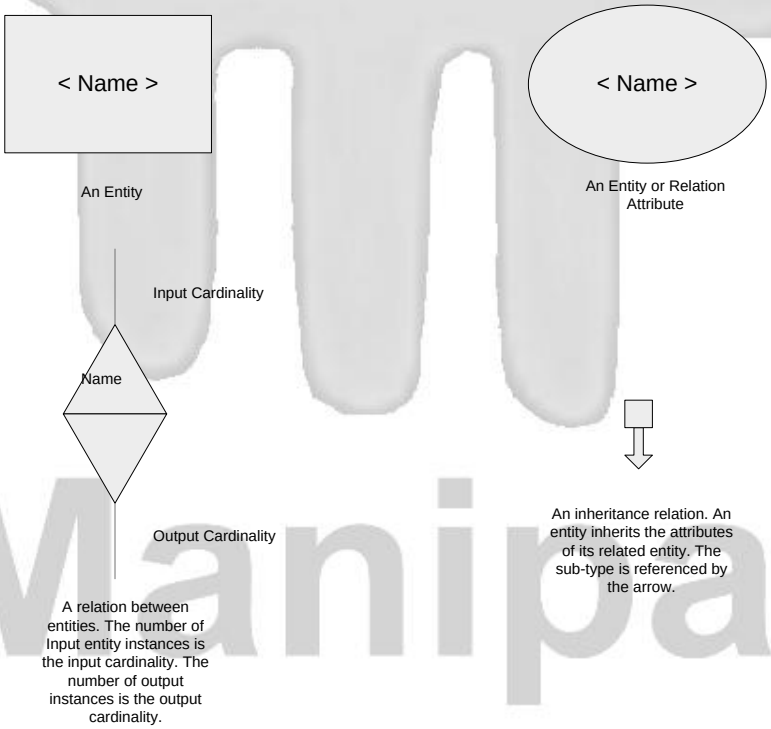


Fig. 4.2: Notations for semantic data models.

Relations between entities may be 1:1, which means one entity instance, participate in a relation with one other entity instance. And they may be 1:M, where an entity instance participates in relationship with more than one other entity instance, or M:N where several entity instances participate in a relation with several others.

Entity-relationship models have been widely used in database design. The database schemas derived from these models are naturally in third normal form, which is a desirable characteristic of relational schemas. Because of the explicit typing and the recognition of sub and super types, it is also straightforward to map these models onto object-oriented databases.

4.2.3 Object models

To support object-oriented programming, an object-oriented development approach may be adopted. This means expressing the system requirements using an object model, designing using an object-oriented approach and developing the system in object-oriented programming languages such as C++.

Object models developed during requirement analysis used to represent both system data and its processing. They combine some of the uses of data-flow and semantic data models. They are useful for showing how entities in the system may be classified and composed of other entities.

Object models of systems, which are developed during requirement analysis, should not include details of the individual objects in the system. They should model classes of objects representing entities. An object class is an abstraction over a set of objects, which identifies common attributes and the services or operations, which are provided by each object.

Various types of object models can be produced showing how object classes are related to each other, how objects are aggregated to form other objects, how objects use the services provided by other objects and so on. Figure 4.3 shows the notation, which is used to represent an object class. There are three parts to this. The object class name has its obvious meaning and the attribute section lists the attributes of that object class. When objects are created using the class as a template, all created objects acquire these attributes. They may then be assigned values that are conformant with the attribute type declared in the object class. The service

section shows the operations associated with the object. These operations may modify attribute values and may be activated from other classes.

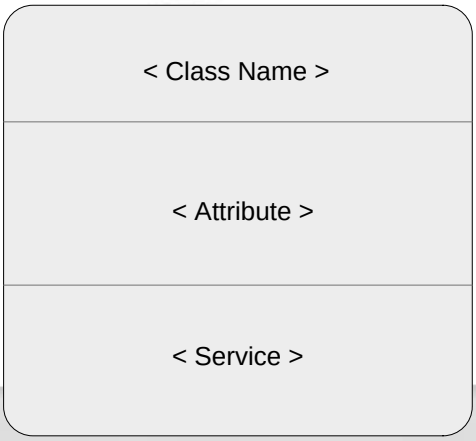


Fig. 4.3: Notation to represent an object class

4.2.3.1 Inheritance models

Object-oriented modeling involves identifying the classes of object, which are important in the domain being studied. These are then organized into taxonomy. Taxonomy is a classification scheme, which shows how an object class is related to other classes through common attributes and services. To display this taxonomy, we organize the classes into an inheritance or class hierarchy where the most general object classes are presented at the top of the hierarchy. More specialized objects inherit their attributes and services. These specialized objects may have their own attributes and services.

The figure below illustrates part of a simplified class hierarchy that might be developed when modeling a library system. This hierarchy gives information about the items held in the library. It is assumed that the library does not simply hold books but also other types of items such as music, recordings of films, magazines, newspapers and so on.

The figure 4.4 shows, the most general item is at the top of the tree and has a set of attributes and services, which are common to all library items. These are inherited by the classes (Published item, Recorded item) which add their own attributes and pass these on to lower-level items.

The design of class hierarchies is not a simple process. One advantage of developing such models is that the analyst needs to understand, in detail, the domain in which the system is to be installed.

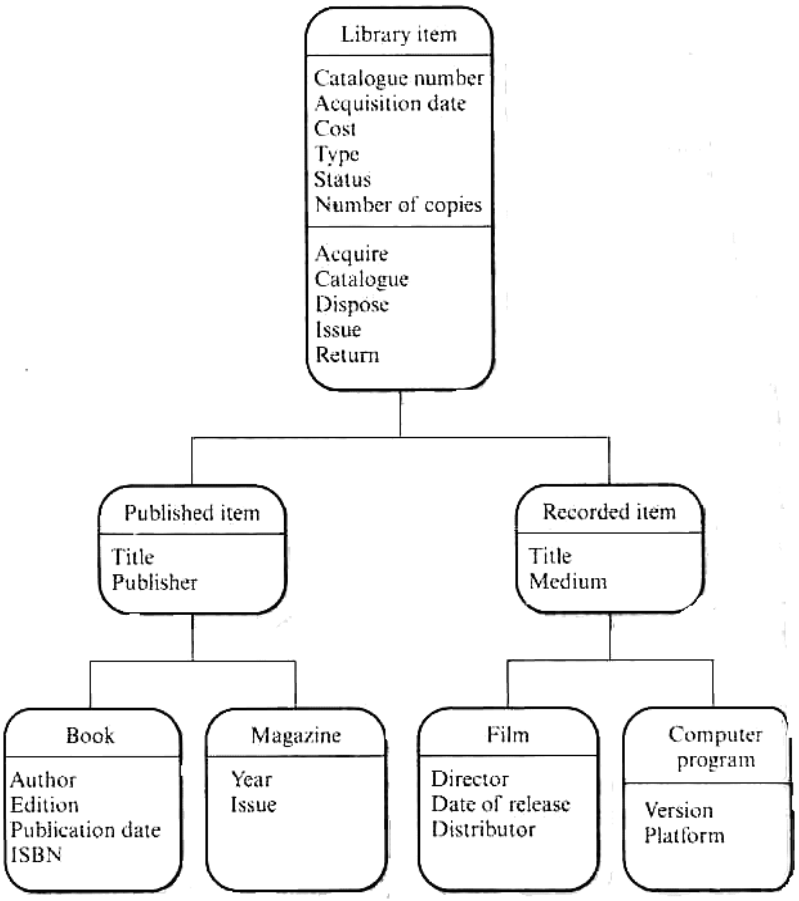


Fig. 4.4: Part of a class hierarchy for a library system

4.2.3.2 Object aggregation

Acquiring attributes and services through an inheritance relationship with other objects, some objects are aggregation of other objects. The classes representing these objects may be modeled using an aggregation model as shown in figure 4.5. In this example it has modeled potential library item which is the materials for particular class given in a university. This does not consist of a single item but includes lecture notes, assignments, sample solutions, copies of transparencies used in lectures, videotapes and so on.

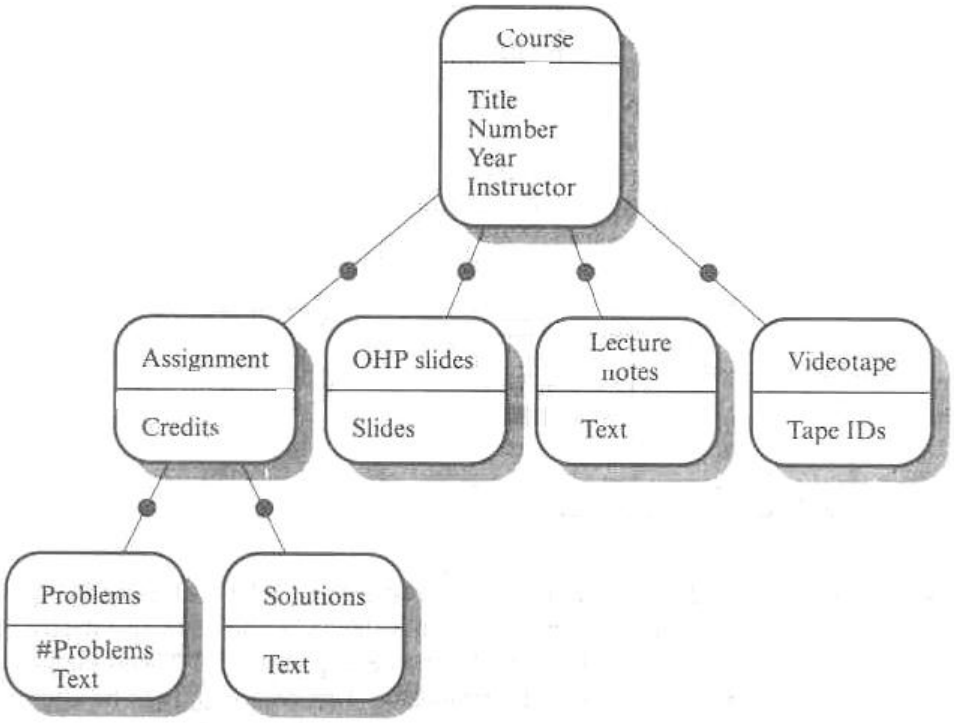


Fig. 4.5: An aggregate object representing a course.

Adding a block blob to a link means that the relationship between objects is a part of relationship rather than an inheritance relationship.

4.2.3.3 Service usage models

The hierarchical models which have covered show object classes and services associate with each object. They do not give any information about how object classes use the services provided by other classes. As well as these hierarchical models, a model showing how class is related to other classes through the operations used is also useful. Figure 4.6 shows some of the classes from the library model. It illustrates that the class 'Library user' makes use of the services 'Issue' and 'Return' associated with 'Library item'. The class 'Library staff' uses the 'Acquire', 'Catalogue' and "Dispose' services associated with 'Library item' and the 'Register' and 'De-register' services associated with 'Library user'.

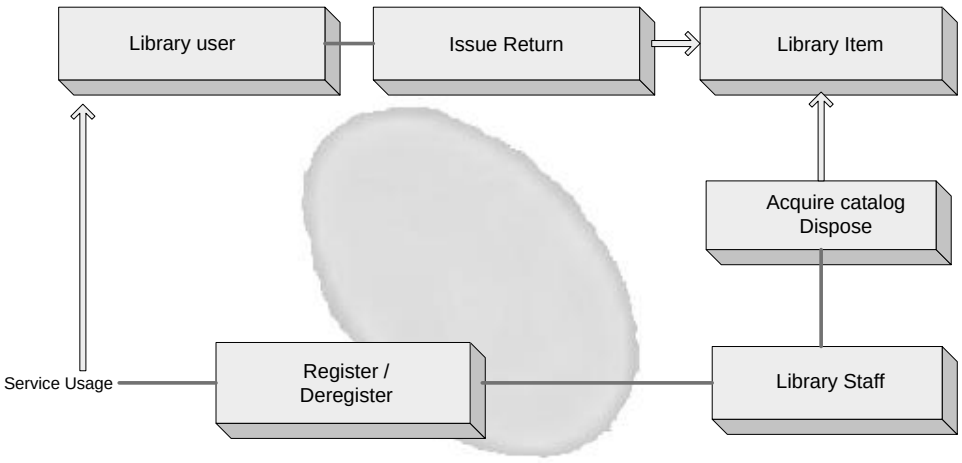


Fig. 4.6: Service usage

4.2.4 Data Dictionaries

Data dictionary is a list of names used by the systems, arranged alphabetically. As well as the name, the dictionary should include a description of the named entity and, if the name represents a composite object, their may be a description of the composition. Other information such as a date of the creation, the creator, and representation of the entity may also be included depending on the type of model which is being developed.

The advantages of using the data dictionary are:

1. It is a mechanism for name management. Many different people who have to invent names for entities and relationships may develop a large system model. These names should be consistently and should not clash. The data dictionary software can check for name uniqueness and tell requirements analyst of name duplications
2. It servers as a store of organization information which can link analysis, design, implementation and evaluation. As the system is developed, information is taken to inform the development. New information is added to it. All information about the entity is in one place.

All system names, whether they be names of entities, types, relations, attributes or services should be entered in the dictionary. Support software should be available to create, maintain and interrogate the dictionary. This software might be integrated with other tools so that dictionary creation is partially automated.

4.3 Software Design

Any design problem must be tackled in three stages:

(1) **Study and understand the problem** without understanding effective software design is impossible. The problem should be examined from a number of different angles or viewpoints as these provide different insights into the design requirements.

(2) **Identify gross features of at least one possible solution**. It is often useful to identify a number of solutions and to evaluate them all. The choice of solution depends on the designer's experience, the availability of reusable components, and the simplicity of the derived solutions. Designers usually prefer familiar solutions even if these are not optimal, as they understand their advantages and disadvantages.

(3) **Describe each abstraction used in the solution.** Before creating formal documentation, the designer may write an informal design description. This may be analyzed by developing it in detail. Errors and omissions in the high-level design will probably be discovered during this analysis. These are corrected before the design is documented.

4.3.1 The design process

A general model of a software design is a directed graph. The target of the design process is the creation of such a graph without inconsistencies. Nodes in this graph represent entities in the design entities such as process function or types. The link represents relation between these design entities such as calls, uses and so on. Software designers do not arrive at a finished design graph immediately but develop the design iteratively through a number of different versions. The design process involves adding formality and detail as the design is developed with constant backtracking to correct earlier, less formal, designs. The starting point is an informal design, which is refined by adding information to make it consistent and complete as shown in figure 4.7 below.

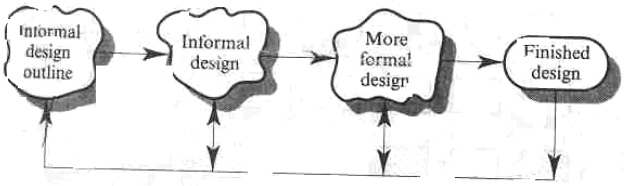


Fig. 4.7: The progression from an informal to a detailed design

A general model of the design process shown in figure 4.8 suggests that the stages of the design process are sequential. In fact, design process activities proceed in parallel. However, the activities shown are all part of the design process for large software systems. These design activities are:

(1) **Architectural designs** the sub-systems making up the system and their relationships are identified and documented.

(2) **Abstract specification** for each sub-system, an abstract specification of the services it provides and the constraints under which it must operate is produced.

(3) **Interface design** for each sub-system, its interface with other sub-systems is designed and documented. This interface specification must be unambiguous as it allows the sub-system to be used without knowledge of the sub-system operation.

(4) **Component design** Services are allocated to different components and the interfaces of these components are designed.

(5) **Data structure design** the data structures used in the system implementation is designed in detail and specified.

(6) **Algorithm design** the algorithms used to provide services are designed in detail and specified.

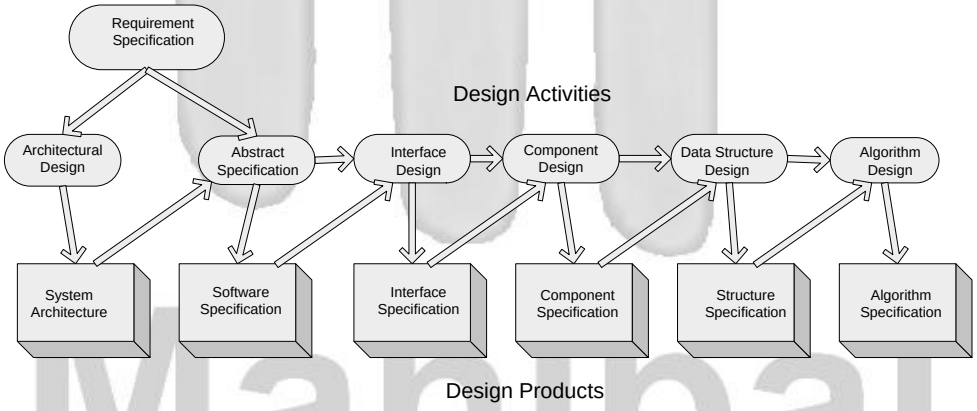


Fig. 4.8: A general model of the design process

This process is repeated for each sub-system until the components identified can be mapped directly into programming language components such as packages, procedures or functions.

4.3.2 Design Methods

A more methodical approach to software design is purposed by structured methods, which are sets of notations and guidelines for software design. Budgen (1993) describes some of the most commonly used methods such as structured design, structured systems analysis, Jackson System Development and various approaches to object-oriented design.

The use of structured methods involves producing large amounts of diagrammatic design documentation. CASE tools have been developed to support particular methods. Structured methods have been applied successfully in many large projects. They can deliver significant cost reductions because they use standard notations and ensure that standard design documentation is produced.

A mathematical method (such as the method for long division) is a strategy that will always lead to the same result irrespective of who applies the method. The term **structured methods** suggests that the designers should normally generate similar designs from the same specification. A structured method includes a set of activities, notations, report formats, rules and design guidelines. So structured methods often support some of the following models of a system:

(1) **A data-flow model** where the system is modeled using the data transformations, which take place as it, is processed.

(2) **An entity-relation model**, which is used to describe the logical data, structures being used.

(3) **A structural model** where the system components and their interactions are documented.

(4) If the method is **object-oriented** it will include an inheritance model of the system, a model of how objects are composed of other objects and, usually, an object-use model which shows how objects are used by other objects.

Particular method supplement these with other system models such as state transition diagrams, entity life histories that show how each entity is transformed as it is processed and so on. Most methods suggest a centralized repository for system information or a data dictionary should be used. No one method is better or worse than other methods: the success or

otherwise of methods often depends on their suitability for an application domain.

4.3.3 Design description

A software design is a model system that has many participating entities and relationships. This design is used in a number of different ways. It acts as a basis for detailed implementation; it serves as a communication medium between the designers of sub-systems; it provides information to system maintainers about the original intentions of the system designers, and so on.

Designs are documented in a set of design documents that describes the design for programmers and other designers. There are three main types of notation used in design documents:

(1) **Graphical notations:** These are used to display the relationships between the components making up the design and to relate the design to the real-world system is modeling. A graphical view of a design is an abstract view. It is most useful for giving an overall picture of the system.

(2) **Program description languages** these languages (PDLs) use control and structuring constructs based on programming language constructs but also allow explanatory text and (sometimes) additional types of statement to be used. These allow the intention of the designer to be expressed rather than the details of how the design is to be implemented.

(3) **Informal text** much of the information that is associated with a design cannot be expressed formally. Information about design rationale or non-functional considerations may be expressed using natural language text.

All of these different notations may be used in describing a system design.

4.3.4 Design strategies

The most commonly used software design strategy involved decomposing the design into functional components with system state information held in a shared data area. Since from late 1980s that this alternative, object oriented design has been widely adopted.

Two design strategies are summarized as follows:

(1) **Functional design:** The system is designed from a functional viewpoint, starting with a high-level view and progressively refining this into a more detailed design. The System State is centralized and shared between the functions operating on that state. Methods such as Jackson Structured Programming and the Warnier-Orr method are techniques of functional decomposition where the structure of the data is used to determine the functional structure used to process that data.

(2) **Object-oriented design:** The system is viewed as a collection of objects rather than as functions. Object-oriented design is based on the idea of information hiding and has been described by Meyer, Booch, and Jacobsen. and many others. JSD is a design method that falls somewhere between function-oriented and object-oriented design.

In an object-oriented design, the System State is decentralized and each object manages its own state information. Objects have a set of attributes defining their state and operations, which act on these attributes. Objects are usually members of an object class whose definition defines attributes and operations of class members. These may be inherited from one or more super-classes so that a class definition need only set out the differences between that class and its super-classes. Objects communicate by exchanging messages; an object calling a procedure associated with another object achieves most object communication.

There is no 'best' design strategy, which is suitable for all projects and all types of application. Functional and object-oriented approaches are complementary rather than opposing techniques. Software engineers select the most appropriate approach for each stage in the design process. In fact, large software systems are complex entities that different approaches might be used in the design of different parts of the system.

An object-oriented approach to software design seems to be natural at the highest and lowest levels of system design. Using different approaches to design may require the designer to convert his or her design from one model to another. Many designers are not trained in multiple approaches so prefer to use either **object-oriented or functional design.**

4.3.5 Design quality

A good design might be a design that allows efficient code to be produced; it might be a minimal design where the implementation is as compact as possible; or it might be the most maintainable design.

A maintainable design can be adapted to modify existing functions and add new functionally. The design must therefore be understandable and changes should be local in effect. The design components should be cohesive which means that all parts of the component should have a close logical relationship. They should be loosely coupled which means that they should not be tightly integrated. Coupling is a measure of the independence of components. The looser the coupling, the easier it is to adapt the design as the effects of change are localized.

Quality characteristics are equally applicable to object-oriented and function-oriented design. Because of the nature of object-oriented design, which encourages the development of independent components, it is usually easier to achieve maintainable designs as information is concealed within objects.

Self Assessment Questions

1. The output of the requirement analysis process is a set of _____ that present abstract description of the system to be developed.
2. _____ transforms the analysis model created using object-oriented analysis into a design model that serves as a blueprint for software construction.
3. CASE stands for _____.

4.4 Architectural Design

Large systems can be decomposed into sub-system that provides some related set of services. The initial design process of identifying this sub-system and establishing a framework for sub-system control and communication is called **Architectural design.**

Architectural design comes before detailed system specification, it should not include any design information. Architectural design is necessary to structure and organize the specification. This model is the starting point for the specification of the various parts of the system.

There is no generally accepted process model for architectural design. The process depends on application knowledge and on the skill and intuition of the system architect. For the process, the following activities are usually necessary:

(1) **System structuring:** The system is structured into a number of principal sub-systems where a sub-system is an independent software unit. Communications between sub-systems are identified.

(2) **Control modeling:** A general model of the control relationships between the parts of the system is established.

(3) **Modular decomposition:** Each identified sub-system is decomposed into modules. The architect must decide on the types of module and their interconnections.

During any of these process stages, it may be necessary to develop the design in more detail to find out if architectural design decision allows the system to meet its requirements. The output of the architectural design process is an architectural design documents. This consists of a number of graphical representations of the system models along with associated descriptive text. It should describe how the system is structured into sub-systems and how each sub-system is structured into modules.

4.4.1 System structuring

The first phase of the architectural design activity is usually concerned with decomposing a system into a set of interacting sub-systems. At its most abstract level, an architectural design may be depicted as a block diagram in which each box represents a sub-system. Boxes within boxes indicate that the sub-system has itself been decomposed to sub-systems. Arrows mean that data and/or control is passed from sub-system in the direction of the arrows. This is illustrated in figure 4.9.

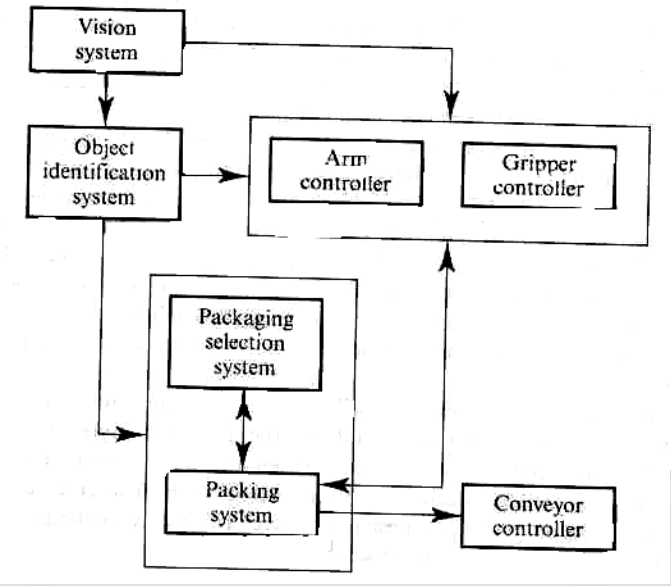


Fig. 4.9: Block diagram of packing robot control system

Figure shows an architectural design for a packing robot system. This robotic system can pack different kinds of object. It uses a vision sub-system to pick out objects on a conveyor, identifies the type of object, and selects the right kind of packaging from a range of possibilities. It then moves objects from the delivery conveyor to be packaged. Packaged objects are placed on another conveyor.

More specific models of the structure may be developed which show how sub-systems share data, how they are distributed and how they interface with each other. In this section three of these standard models, namely a repository model, a client-server model and an abstract machine model are discussed.

4.4.1.1 The repository model

Sub-systems making up a system must exchange information so that they can work together effectively. There are two ways in which this can be done:

(1) All shared data is held in a central database that can be accessed by all sub systems. A system model based on a shared database is sometimes called a **repository model**.

(2) Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing messages to them.

The majority of systems, which use large amounts of data, are organized around a shared database or repository. This model is therefore suited to applications where data is generated by one sub-system and used by another.

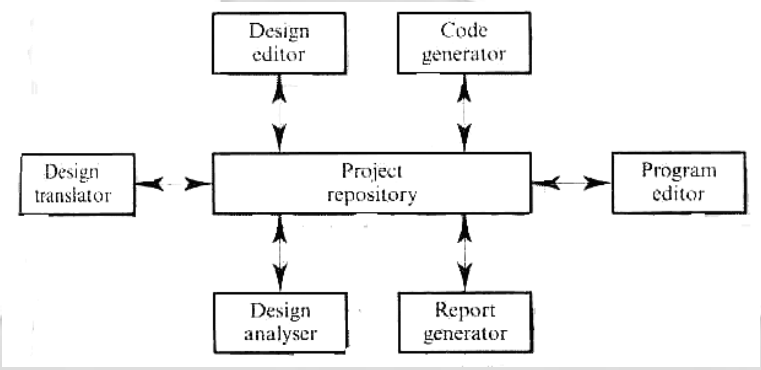


Fig. 4.10: The architecture of an integrated CASE tool set

Figure 4.10 shown above is an example of a CASE tool set architecture based on a shared repository.

The advantages and disadvantage of a shared repository are as follows:

(1) It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.

(2) Sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.

(3) Sub-systems, which produce data, need not be concerned with how that data is used by other sub-systems.

(4) Evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive and may be difficult or even impossible.

(5) Activities such as backup, security, access control and recovery from error are centralized. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.

(6) Different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.

(7) The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.

(8) It may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

4.4.1.2 The client–server model

The client-server architectural model is a distributed system model which show how data and processing is distributed across a range of processors is shown in figure 4.11. The major components of this model are:

(1) A set of stand-alone servers which offer services to other sub-systems. Examples of servers are print servers which offer printing services, file servers which offer file management services and a compile server which offers language translation services.

(2) A set of clients that call on the services offered by servers. These are normally sub-systems in their own right. There may be several instances of a client program executing concurrently.

(3) A network which allows the clients to access these services. In principle, this is not necessary as both the clients and the servers could run on a single machine. Clients must know the names of the available servers and the services that they provide. However, servers need not know either the identity of clients or how many clients there are. Clients access the services provided by a server through remote procedure calls.

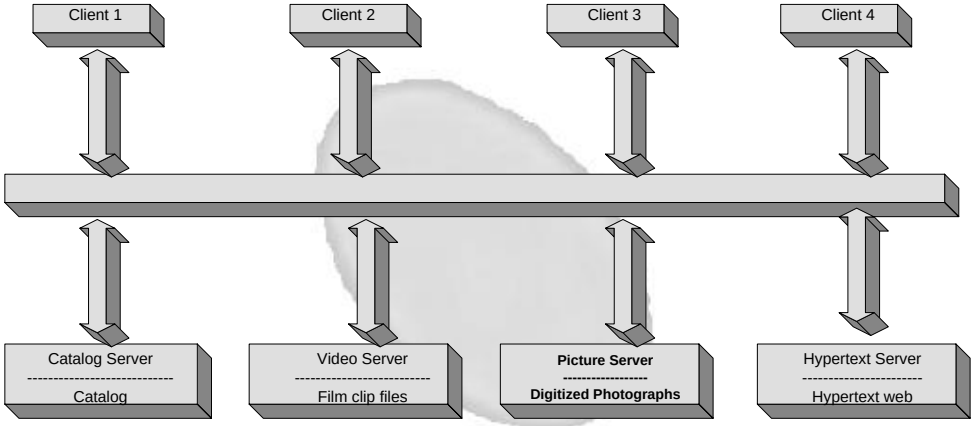


Fig. 4.11: The architecture of a film and picture library system

The client–server approach can be used to implement a repository-based system where the repository is provided as a system server. Sub-systems accessing the repository are clients. Normally, however, each sub-system manages its own data. Servers and clients exchange data for processing. This can result in performance problems when large amounts of data are exchanged. However, as faster networks are developed, this problem is becoming less significant.

The most important advantage of the client-server model is that distribution is straightforward. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it gradually with the rest of the system or to upgrade servers transparently without affecting other parts of the system.

4.4.1.3 The abstract machine model

The abstract machine model of architecture (sometimes called a layered model) models the interfacing of sub-systems. It organizes a system into a series of layers each of which provides a set of services. Each layer defines an abstract machine whose machine language (the services provided by the layer) is used to implement a next level of abstract machine. For example, a common way to implement a language is to define an ideal 'language machine' and compile the language into code for this machine. A further translation step then converts this abstract machine code to real machine code.

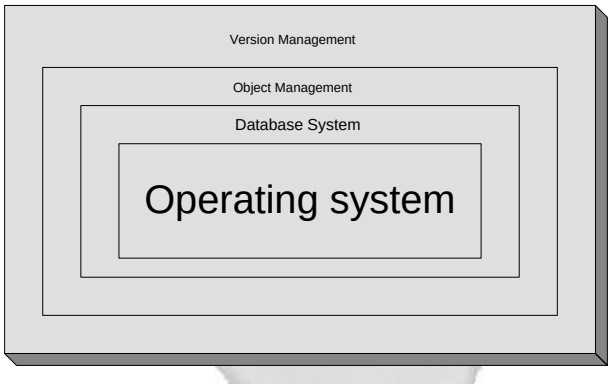


Fig. 4.12: Abstract machine model of a version management system

A well-known example of this approach is the OSI reference model of network Protocols. Another influential example of this approach was proposed by suggested a three-layer model for an Ada programming support environment (APSE).

The figure 4.12 shows that the version management system relies on managing versions of objects and provides general configuration management facilities. To support these configuration management facilities, it uses an object management system which provides information storage and management services for objects. This system uses a database system to provide basic data storage and services such as transaction management, rollback and recovery, and access control. The database management uses the underlying operating system facilities and file store in its management uses the underlying operating system facilities and file store in its implementation.

The layered approach supports the incremental development of system. As a layer is developed, some of the services provided by that layer may be made available to users. This architecture is also changeable and portable.

A disadvantage of the layered approach is that structuring system in this way can be difficult. Inner layers may provide basic facilities, such as file management, which are required by all abstract machines. Services required by the user may therefore require access to an abstract machine that is several levels beneath the outermost layer. This subverts the model, as an outer layer is not longer simply dependent on its immediate predecessor.

Performance can also be a problem because of the multiple levels of command interpretation, which are required. If there are many layers, some overhead is always associated with layer management. To avoid these problems, applications may have to communicate directly with inner layers rather than use facilities provided in the abstract machine.

4.4.2 Control models

The models for structuring a system are concerned with how a system is decomposed into sub-systems. To work as a system, sub-systems must be controlled so that their services are delivered to the right place at the right time. Structural models do not (and should not) include control information. Rather, the architect should organize the sub-systems according to some control model, which supplements the structure model is used. Control models at the architectural level are concerned with the control flow between sub-systems.

Two general approaches to control can be identified:

(1) **Centralized control:** One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another sub-system but will expect to have this control responsibility returned to it.

(2) **Event-based control:** Rather than control information being embedded in a sub-system, each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system.

Control models supplement structural models. All the above structural models may be implemented using either centralized or event-based control.

Centralized control

In a centralized control model, one sub-system is designated as the system controller and has responsibility for managing the execution of other sub-systems.

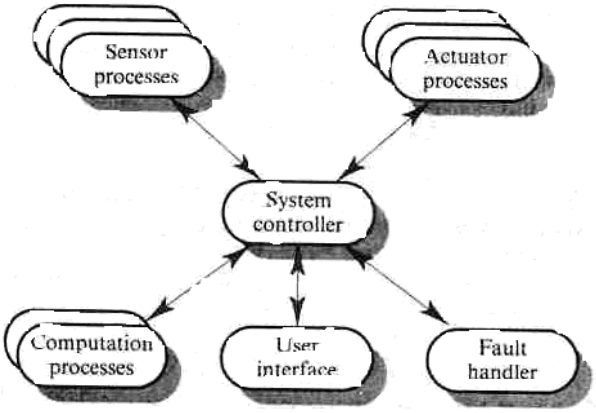


Fig. 4.13: A centralized model of real time system

Figure 4.13 shows an illustration of a centralized management model of control for a concurrent system. This model is often used in 'soft' real-time systems, which do not have very tight time constraints. The central controller manages the execution of a set of processes associated with sensors and actuators.

Event-driven systems

In centralized control models, control decisions are usually determined by the values of some system state variables. By contrast, event-driven control models are driven by externally generated events.

The distinction between an event and a simple input is that the timing of the event is outside the control of the process which handless that event. A sub-system may need to access state information to handle these events but this state information does not usually determine the flow of control.

There are two event-driven control models:

(1) **Broadcast models:** In these models, an event is, in principle, broadcast to all sub-systems. Any sub-system, which is designed to handle that event, responds to it.

(2) **Interrupt-driven models:** These are exclusively used in real-time systems where an interrupt handler detects external interrupts. They are then passed to some other component for processing.

Broadcast models are effective in integrating sub-systems distributed across different computers on a network. Interrupt-driven models are used in real-time systems with stringent timing requirements.

The advantage of this approach to control is that it allows very fast responses to events to be implemented. Its disadvantages are that it is complex to program and difficult to validate.

4.4.3 Modular decomposition

After a structural architecture has been designed, another level of decomposition may be part of the architectural design process. This is the decomposition of sub-systems into modules.

Here considered two models which may be used when decomposing a sub-system into modules:

(1) **An object-oriented model** the system is decomposed into a set of communicating objects.

(2) **Data-flow models** the system is decomposed into functional modules, which accept, input data and transform it, in some way, to output data. This is also called a pipeline approach.

In the **object-oriented model**, modules are objects with private state and defined operations on that state. In the data-flow model, modules are functional transformations. In both cases, modules may be implemented as sequential components or as processes.

The advantages of the object-oriented approach are objects are loosely coupled, the implementation of objects can be modified without affecting other objects. Objects are often representations of real-world entities so the structure of the system is readily understandable. Because these real-world entities are used in different systems, objects can be reused. Object-oriented programming languages have been developed which provide direct implementations of architectural components.

However, the object-oriented approach does have disadvantages. To use services, objects must explicitly reference the name and the interface of other objects. If an interface change is required to satisfy proposed system changes, the effect of that change on all users of the changed object must be evaluated. More complex entities are sometimes difficult to represent using an object model.

In a data-flow model, functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

The advantages of this architecture are:

(1) It supports the reuse of transformations.

(2) It is intuitive in that many people think of their work in terms of input and output processing.

(3) Evolving system by adding new transformations is usually straightforward.

(4) It is simple to implement either as a concurrent or a sequential system.

4.4.4 Domain-specific architectures

The previous architectural models are general models. They can be applied to many different classes of application. As well as these general models, architectural models, which are specific to a particular application domain, may also be used. Instances of these systems differ in detail. The common architectural structure can be reused when developing new systems. These architectural models are called **domain-specific architectures**.

There are two types of domain-specific architectural model:

(1) **Generic models** which are abstractions from a number of real systems. They encapsulate the principal characteristics of these systems. The class of systems modeled using a generic model is usually quite restricted. For example, in real-time systems, there might be generic architectural models of different system types such as data collection systems, monitoring systems, and so on.

(2) **Reference models**, which are more, abstract and describe a larger class of systems. They provide a means of informing system architects about that class of system.

Generic models are usually derived “bottom-up” from existing systems where as reference models are derived “top-down”.

Self Assessment Questions

4. The initial design process of identifying the sub-system and establishing a framework for sub-system control and communication is called _____.
5. A system model based on a shared database is sometimes called a _____.
6. The _____ of architecture models the interfacing of sub-systems.

4.5 Summary

- The main design activities in the software process are architectural design, system specification, interface design, component design, data structure design and algorithm design.
- Functional decomposition involves modeling a system as a set of interacting functional units,. Objects oriented decomposition models the system as a set of objects where an object is an entity with state and functions to inspect and modify that state.
- Functional oriented and object-oriented design are complimentary rather than opposing design strategies. Different perspectives may be applied at different levels of design abstraction.
- The software architecture is responsible for deriving an overall structural model of the system, which identifies sub-systems and their relationships. Architect may also design a control model for the system and decompose sub-systems into modules.
- Large systems rarely confirm to a single architectural model. They are heterogeneous and incorporate different models at different levels of abstraction.
- System decomposition models include repository models, cline-server models and abstract machine models. Repository models share data through common store. Client –server models distribute data. Abstract machine models are layered with each layer implemented using facilities provided its foundation layer.
- Examples of control models include centralized control and event models, In centralized model, control decisions are made depending on the system state, in event models external events control the system.
- Examples of modular decomposition models include data-flow and object models. Data-flow are functional, where as object models are

based on loosely coupled entities which maintain their own state and operations.

- Domain specific architecture models are abstraction over an application domain. Domain-specific models may be generic models which are constructed bottom-up from existing systems, or reference models which are idealized, abstract models of the domain.

4.6 Terminal Questions

1. Explain different system models.
2. Explain the different stages in software design.
3. What is meant by Architectural Design? Explain.

4.7 Answers

Self Assessment Questions

1. System models
2. Object oriented design
3. Computer Aided Software Engineering
4. Architectural design
5. Repository model
6. Abstract machine model

Terminal Questions

1. Examples of the different types of system model, which might be produced as part of the analysis process and the notations used to represent these models, are:
   I. **A data-processing model:** Data flow diagrams may be used to show how data is processed at different stages in the system.
   II. **A composition model**: entity-relation diagram may be used to show how some entities in the system are composed of other entities.
   III. **A classification model:** Objects class/inheritance diagrams may be used to show how entities have common characteristics.
   IV. **A stimulus-response model:** State transition diagrams may be used to show how the system reacts to internal and external events.
   V. **A process model:** Process models may be used to show the principal activities and deliverables involved in carrying out some process. (Refer Section 4.2)

2. Any software design problem must be tackled in three stages:
   I. **Study and understand the problem** without understanding effective software design is impossible. The problem should be examined from a number of different angles or viewpoints as these provide different insights into the design requirements.
   II. **Identify gross features of at least one possible solution**. It is often useful to identify a number of solutions and to evaluate them all. The choice of solution depends on the designer's experience, the availability of reusable components, and the simplicity of the derived solutions. Designers usually prefer familiar solutions even if these are not optimal, as they understand their advantages and disadvantages.
   III. **Describe each abstraction used in the solution.** Before creating formal documentation, the designer may write an informal design description. This may be analyzed by developing it in detail. Errors and omissions in the high-level design will probably be discovered during this analysis. These are corrected before the design is documented. (Refer section 4.3)
3. Large systems can be decomposed into sub-system that provide some related set of services. The initial design process of identifying this sub-system and establishing a framework for sub-system control and communication is called **Architectural design.** (Refer section 4.4)

Unit 5 Object Oriented Design

Structure:

5.1 Introduction

Object-oriented design transforms the analysis model created using object-oriented analysis into a design model that serves as a blueprint for software construction.

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. It must be possible to find pertinent objects, factor them into classes at the right granularity, define class interfaces hierarchies, and establish key relationships among them. The design should be specific to the problem and also general enough to address future problems and requirements.

Unlike conventional software design methods, OOD results in a design that achieves a number of different levels of modularity. Major system components are organized into sub-systems, a system-level “module”. Data and the operations that manipulate the data are encapsulated into objects-a modular form.

Objectives:

After studying this unit, you should be able to:

- explain the software design principles using object oriented approach
- describe classes, object classes and inheritance
- differentiate object-oriented and function oriented approaches

5.2 Object Oriented Design

Object–oriented design is a design strategy based on information hiding. It differs from the functional approach to design in that it views a software system as a set of interacting objects, with their private state, rather than as a set of functions that share a global state.

The characteristics of an object-oriented design (OOD) are:

(1) Objects are abstraction of system entities, which are responsible for managing their own private state and offering services to other objects.

(2) Objects are independent entities that may readily be changed because state and representation information is held within the objects. Changes to the representation may be made without reference to other system objects.

(3) System functionality is expressed in terms of operations or services associated with each object.

(4) Shared data areas are eliminated. Objects communicate by calling on services offered by other objects rather than sharing variables. This reduces overall system coupling. There is no possibility of unexpected modifications to shared information.

(5) Objects may be distributed and may execute either sequentially or in parallel. Decisions on parallelism need not be taken at an early stage of the design process.

Object-oriented systems are easier to maintain as the objects are independent. They may be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. There is a clear mapping between real-world entities and their controlling objects in the system. This improves the understandability and hence maintainability of the design.

Object-oriented analysis, design and programming are all part of **Object-oriented development** whereby an object-oriented strategy is used throughout the development process**.**

- **Object-oriented analysis:** is concerned with developing an object-oriented model of the application domain. The identified objects may or may not map directly into system objects.

- **Object-oriented design:** is concerned with developing an object-oriented model of a software system to implement the identified requirements. These requirements may or may not be structured around objects in the problem domain.
- **Object-oriented programming:** is concerned with realizing a software design using an object-oriented programming language. An object-oriented programming language supports the direct implementation of objects and provides object classes and inheritance.

Object-oriented concepts and design activities that are common to the object-oriented design process which is proposed by all method. These include:

- The identification of the objects in the system along with their attributes and operations.
- The organization of objects into an aggregation hierarchy which shows how objects are ‘ part-of’ other objects.
- The construction of dynamic ‘object-use’ diagrams that show which objects services are used by other objects.
- The specification of object interfaces.

5.2.1 Objects, Object Classes & Inheritance

An object is an entity that has a state and a defined set of operations, which operate, on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required. Objects are created according to some object class definition. An object class definition serves as a template for objects. It includes declarations of all the attributes and services which should be associated with an object of that class.

An object oriented design process is normally concerned with designing object classes. When the design is implemented, the required objects are created using these class definitions.

Objects communicate by requesting services from other objects and, if necessary, exchange information required for service provision. In some distributed systems, object communications are implemented directly as text messages, which are exchanged by objects. The receiving object parses the

message, identifies the service and the associated data and carries out the requested service.

For good design we have to hide information so the representation of the object should not be accessible from outside the object. When the object design is developed, the attributes should be accessed and modified through appropriate access and update functions. This allows the representation to be changed at a later stage in the design or implementation process without affecting other objects.

5.2.1.1 Inheritance

When objects are created they inherit the attributes and operations of their class. Object classes are themselves objects so inherit their attributes from some other class (their 'super-class'). Inheritance trees (class hierarchies) show how objects inherit attributes and services from their super-classes.

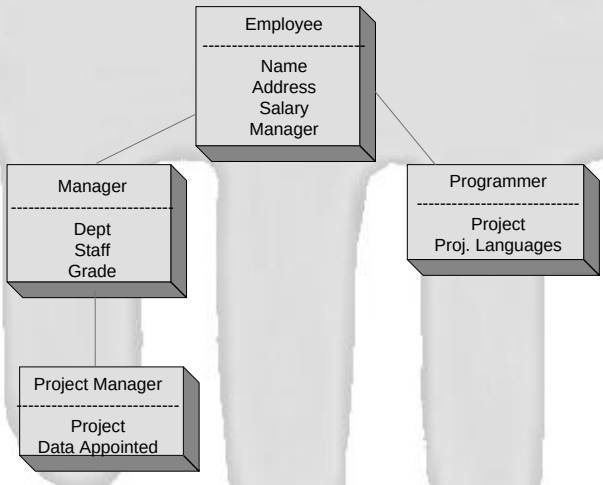


Fig. 5.1: A class hierarchy showing attributes and inheritance

In the class hierarchy shown in Figure 5.1 programmers is employees and inherits attributes from class Employee. The class Programmer then adds particular attributes specific to programmers such as the current project and known programming languages. Project managers are both managers and employees. They inherit their attributes from the class Manager which has already inherited attributes from Employee.

Inheritance is valuable for object-oriented modeling and object-oriented programming. However, if inheritance hierarchies are developed during the

design process, these may confuse rather than clarify the design. There are three reasons for this:

(1) Object classes are not self-contained. They cannot be understood on their own without reference to any super-classes.

(2) Designers have a tendency to reuse the inheritance graph created during analysis. This may lead to inefficient designs as this graph reflects the application domain rather than the system to be developed.

(3) The needs of analysis, design and implementation are all different. While the inheritance graphs may be similar, they are rarely identical. This is likely to confuse future maintainers of the system.

So there is not a strong case for developing inheritance graphs during design.

5.2.2 Object identification

The main problem in object-oriented design is identifying the objects that make up the system, their attributes and associated operations. There is no simple formula, which allows objects to be identified. Designers must use their skill and experience in this task.

There have been various proposals made about how to identify objects:

- **Use grammatical analysis** of a natural language description of a system. Objects and attributes are nouns, operations or services are verbs.
- **Use tangible entities** (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations such as offices, organizational units such as companies, and so on.
- **Use a behavioral approach** in which the designer first understands the overall behavior of the system. The various behaviors are assigned to different parts of the system and an understanding derived of who initiates and participates in these behaviors. Participants who play significant roles are recognized as objects.
- **Use a scenario-based analysis** where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes and operations. A method of analysis called CRC cards

whereby analysts and designers take on the role of objects is effective in supporting this scenario-based approach.

These approaches are not exclusive. Good designers may use all of them when trying to identify objects.

5.2.3 An object-oriented design example

The example illustrates object-oriented design is a system for creating weather maps using automatically collected meteorological data.

A weather data collection system is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations. Each weather station collects meteorological data over a period and produces summaries of that data. On request, it sends the collected, processed information to an area computer for further processing. Data on the air temperature, the ground temperature, the wind speed and direction, the barometric pressure and the amount of rainfall is collected by each weather station.

Figure 5.2 shows the architecture of Weather mapping system. Weather stations transmit their data to the area computer in response to a request from that machine. The area computer collates the collected data over period and produces summaries of that data. On request, it sends the collected, data and integrates it with reports from other sources such as satellites and ships. Using a digitized map database it then generates a set of local weather maps.

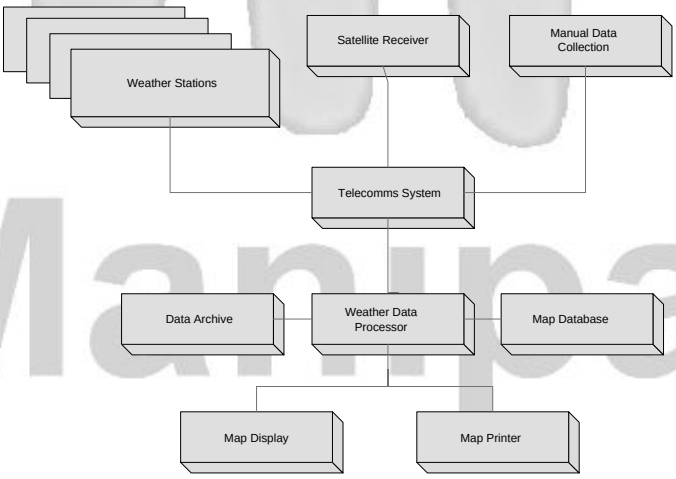


Fig. 5.2: Architecture of the weather mapping system

Ignoring satellite and manually collected data and consider only the weather station information in the rest of this example.

From the outline description of the system and the architectural diagram, four main abstract objects can be identified.

(1) **A Weather station** which collects information and communicates it for processing.

(2) **A Map database** which provides templates of maps for weather data to be added. Assume this is a database of survey information that allows maps of the area to be generated at various scales.

(3) **A Map** which is displayed and printed. Assume that a weather map is an outline of areas with superimposed weather information.

(4) **Weather data**, which is used to produce the map, and which is archived.

A description of the automatic weather station may be used to identify its objects:

A weather station is a package of software-controlled instruments, which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected every five minutes. When a command is issued to transmit the weather data, the weather station processes and summarizes the collected data. The summarized data is transmitted to the mapping computer when a request is received.

From this description, it is possible to identify some objects, attributes and operations of a weather station:

(1) Objects are air and ground thermometers, anemometer, wind vane, barometer and rain gauge. The instrument package may also be an object in its own right but this is not clear at this stage.

(2) Operations are collect data, perform data processing, and transmit data.

(3) Attributes are ‘summarized data’.

At this stage in the design process, knowledge of the application domain may be used to identify further objects and services. In this case, we know

that weather stations are often located in remote places. They include various instruments, which sometimes go wrong. Instrument failures should be reported automatically. This implies that attributes and operations to check the correct functioning of the instruments are necessary.

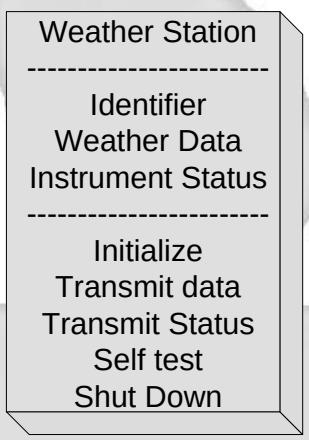


Fig. 5.3: A model of a class description for a weather station

Object class description for Weather station is shown in Figure 5.3. The state includes a unique station identifier, a weather data record (which will be described in more detail later), and an instrument status record. The weather station includes operations to initialize the state, to transmit weather data, to run a self-checking program, to transmit status information and to shut itself down. There is no explicit operation to collect weather information. Data collection starts automatically when the system is switched on.

The weather station object includes hardware control objects, which manage and control its instruments. The term 'hardware control object' means an object, which interacts directly with a hardware unit. It is a good design principle to associate a hardware control object with each piece of hardware in the system.

Hardware control objects are used to hide details of the system hardware. Say a hardware unit provides information by writing into a known memory location. The address of that location should be concealed in the hardware control object. If the hardware is redesigned and a different address used, there is no need to change the software, which interfaces, to the hardware control object.

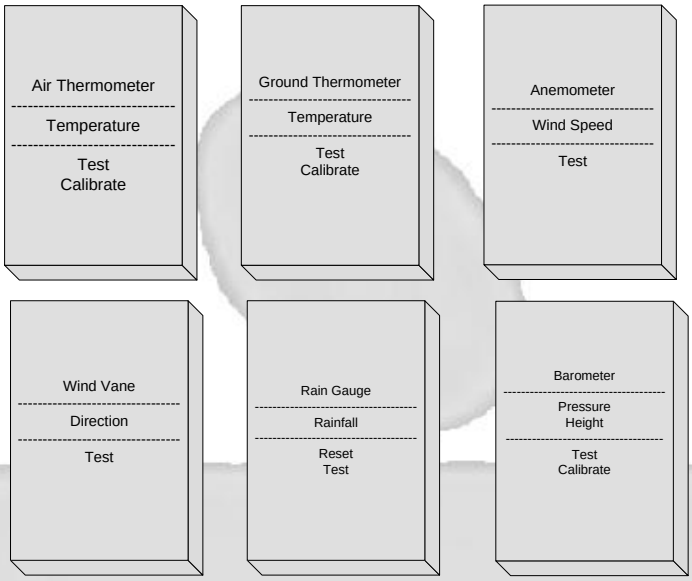


Fig. 5.4: Hardware control objects in the weather station

Figure 5.4 shows the possible design for the hardware control objects in the weather station. The object attributes represent the data collected by the instrument. All objects have a Test operation, which runs a self-test program on the instrument. As the rain gauge measures cumulative rainfall, it must have a Reset operation. The barometer object must have a Height attribute, as the barometric pressure reading must be adjusted to compensate for the height of the instrument.

For implementation, these objects could be arranged in an inheritance hierarchy with an object class such as Instrument with a Test operation at the top of the hierarchy. Developing a hierarchy at this stage, however, does not help our understanding of the design.

The following data will be collected by the weather station:

* Air temperature: Maximum, minimum and average temperature
* Ground temperature: Maximum, minimum and average temperature.
* Wind speed: Average speed, maximum gust speed
* Pressure: Average barometric pressure
* Rainfall: Cumulative rainfall
* Wind direction: Direction every five minutes

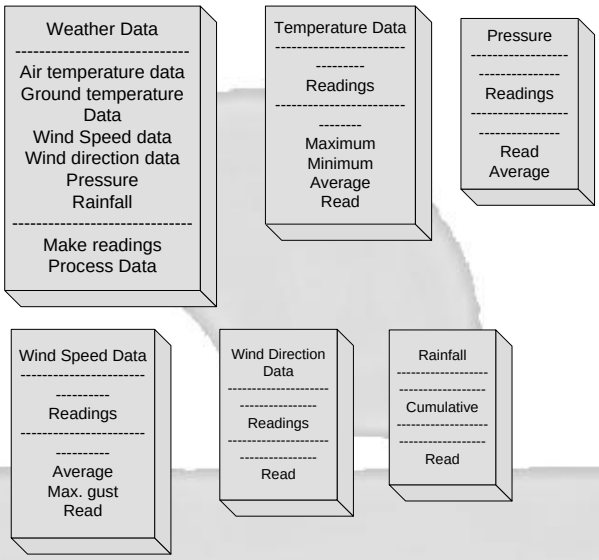


Fig. 5.5: Weather data and associated objects

Measurements should be made every five minutes and the above data computed from these measurements.

With this information, the object Weather data, which gathers and stores this information, can be defined. Weather data accumulates the data to be transmitted to the mapping system. Figure 5.5 shows the Weather data and associated objects. The attributes of Weather data are objects, which have an associated vector of readings that holds collected data. When a Read operation is called, a reading is made and added to this vector. Figure shows weather data and its associated objects. Air temperature data and Ground temperature data are both instances of the class Temperature data.

Weather data acts as a focus for the data collection. When a set of data is to be collected, Weather station requests Weather data to make readings. Weather data then calls all lower-level objects to evaluate the hardware control objects. The results are saved in the local Readings attribute.

The operation Process data in Weather data is initiated when a transmission of weather data is required. Process data calls the appropriate operations on lower-level objects (Maximum, Minimum, and so on). Using the raw weather data that is stored in Readings, these lower-level objects compute

the information (such as the maximum and minimum temperatures) required by Process data.

So far, the identified objects in the weather station system have been directly related to the collection of data. Other objects are required to handle communications and instrument status checked. The operations concerned with the transmission of weather data and instrument status suggest that there should be an object, which handles data communications (Comms). The attribute recording instrument status information suggests that the weather station instruments should be packaged in some way under a single object (Instruments). Data is collected at regular intervals so there is a need for a hardware clock and an associated hardware control object (clock).

Comms is active as discussed in the following section. It should be implemented as a continuously running process. It monitors the communications hardware for an incoming signal. When a signal is detected, the input command is entered into the input buffer. An internal operation parses this input and calls the appropriate operation in Weather station. When the weather station is switched on, Clock and Comms are started and initialized. The clock is synchronized when data is transmitted to the weather mapping computer.

5.2.4 Object aggregation

Various objects have been identified without considering the static structure of the system. Objects are organized into an aggregation structure that shows how one object is composed of a number of other objects. The aggregation relationship between objects is a static relationship. When implemented, objects, which are part of another object, may be implemented as sub-objects. Their definition may be included in the definition of the object of which they are a part.

5.3 Service Usage

For each operation in the weather station, an interaction diagram can be produced showing how a call to that operation triggers requests for service from other objects in the system. Figure 5.6 shows how the Transmit data operation in Weather station results in calls to other objects to compute the required information. The arrowed lines indicate that an object calls on a

service provided by another object (at the head of the arrow) with the service names indicated in a box on the line. The service name may be either an operation name or an attribute name. If it is an operation, the objects receiving the request should execute that operation. If it is an attribute name, the value of that attribute should be delivered.

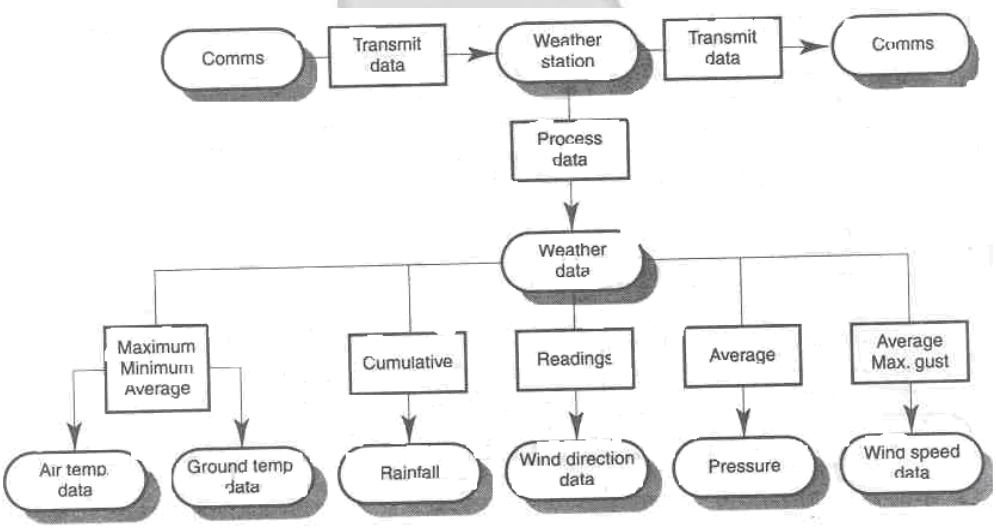


Fig. 5.6: Object interactions for the transmit data operation

The Comms object calls Weather station with a Transmit data command. These causes Weather station to request processed weather data, which is collected from other objects. Weather station then calls Communication to transmit the collected data.

When object interaction charts have been prepared for all operations, which are part of Weather station, a composite object interaction chart may be drawn. This summarizes all object-object communications. Obviously, in some systems, there are so many interactions that it is impractical to produce a composite chart. In such cases, designers must make their own judgement about which object interaction need be documented.

After identifying the object hierarchy and the object interactions, the next step is to develop the design of the object interface. Once the interface design has been established and agreed, it is then possible to implement each object, without reference to other object designs.

Self Assessment Questions

1. _____ is a design strategy based on information hiding.
2. A/An _____ is an entity that has a state and a defined set of operations, which operate, on that state.
3. Objects are organized into a/an _____ structure that shows how one object is composed of a number of other objects.

5.4 Object Interface Design

Object Interface design is concerned with specifying the detail of the object interfaces. This means defining the types of the object attributes and the signatures and the semantics of the object operations. If an object-oriented programming language is being used for implementation, it is natural to use it to express the interface design.

Designers should avoid interfaces representation information in their interface design. Rather, the representation should be hidden and object operations provided to access and update the data. If the representation is hidden, it can be changed without affecting the objects that use these attributes. This leads to a design which is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects, which use the stack.

5.4.1 Design evolution

An important advantage of an object-oriented approach to design is that it simplifies the problem of making changes to the design. The reason for this is that object state representation does not influence the design. Changing the internal details of an object is unlikely to affect any other system objects. Furthermore, because objects are loosely coupled, it is usually straightforward to introduce new objects without significant effects on the rest of the system.

To illustrate the robustness of the object-oriented approach, assume that pollution-monitoring capabilities are to be added to each weather station. This involves adding an air quality meter to compute the amount of various pollutants in the atmosphere. The pollution readings are transmitted at the same time as the weather data. To modify the design, the following changes must be made:

Figure 5.7 below shows weather station and the new objects to the system. The abbreviation NO in Air quality stands for nitrous oxide.

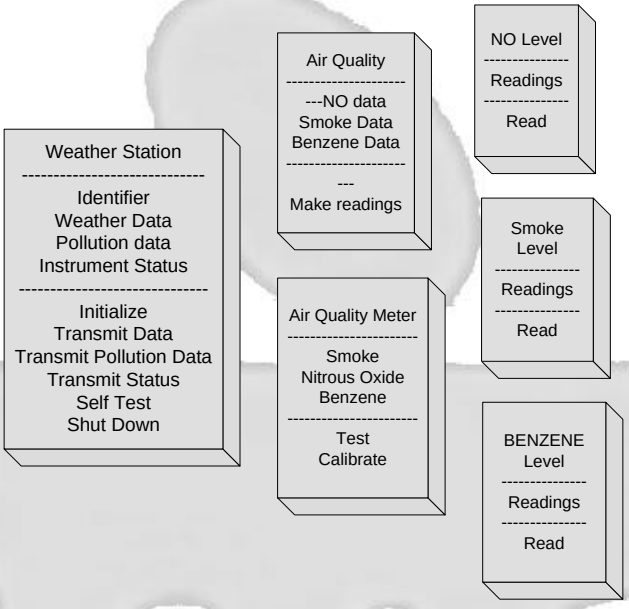


Fig. 5.7: New objects to support pollution monitoring

(1) An object Air quality should be introduced as part of Weather station at the same level as Weather data.

(2) An operation Transmit pollution data should be added to Weather station to send the pollution information to the central computer. The weather station control software must be modified so that pollution readings are automatically collected when the system is switched on.

(3) Objects representing the types of pollution, which can be monitored, should be added. Levels of nitrous oxide, smoke and benzene can be measured.

(4) A hardware control object Air quality meter should be added as a sub-object to Air quality. This has attributes representing each of the types of measurement, which can be made.

The addition of pollution data collection does not affect weather data collection in any way. Data representations are encapsulated in objects so they are not affected by the additions to the design.

5.4.2 Function oriented design

A function-oriented design strategy relies on decomposing the system into a set of interacting functions with a centralized system state shared by these functions as shown in figure 5.8 below. Functions may also maintain local state information but only for the duration of their execution.

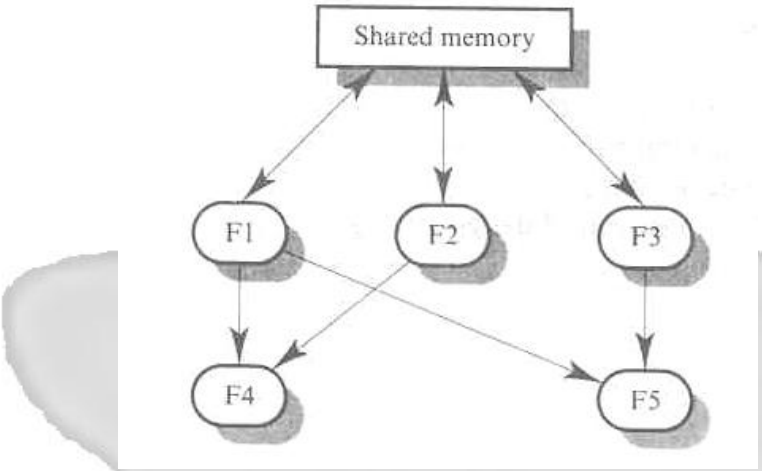


Fig. 5.8: A function-oriented view of design

Function-oriented has been practiced informally since programming began. Programs were decomposed into subroutines, which were functional in nature. In the late 1960s and early 1970s several books were published which described 'top-down' functional design. They specifically proposed this as a 'structured' design strategy. These led to the development of many design methods based on functional decomposition.

Function-oriented design conceals the details of an algorithm in a function but system state information is not hidden. This can cause problems because a function can change the state in a way, which other functions do not expect. Changes to a function and the way in which it uses the system state may cause unanticipated changes in the behavior of other functions.

A functional approach to design is therefore most likely to be successful when the amount of system state information is minimized and information sharing is explicit. Systems whose responses depend on a single stimulus or input and which are not affected by input histories are naturally function-oriented. Many transaction-processing systems and business data-processing systems fall into this class. In essence, they are concerned with

record processing where the processing of one record is not dependent on any previous processing.

An example of such a transaction processing system is the software that controls automatic teller machines, which are now installed outside many banks. The service provided to a user is independent of previous services provided so can be thought of as a single transaction. Figure 5.9 below illustrates a simplified functional design of such a system.

```
Loop
        Loop
          Print_input message ("Welcome – Please enter you Card ");
          Exit when card_input;
        End loop;
          Account_number = Read_card;

          Get_account_details (PIN, Account_balance, cash_available);
                If validate-card (PIN) then
                   Loop
                      Print_operation_select_message :
                         Case Get_button is
                                  When cash_only =>
                                  Dispense_cash (cash_available, amount-dispensed) ;
                                  When Print_balance =>
                                  Print_customer-balance (account-balance);
                                  When statement =>
                                  Order_Statement(Account_number);
                                  When Check-book =>
                                  Order_checkbook (account_number);
                         End case;
                Eject_card;
                Print (" Please take your card or press CONTINUE);
                Exit when Card-Removed;
                End loop;
                Update_account-information (Account-number, Amount-dispensed);
                Else
                Retain_card;
                End if;
End loop;
```

Figure 5.9: The functional design of software for an ATM

In this design, the system is implemented as a continuous loop and actions are triggered when a card is input. Functions such as Dispense_cash, Get_account_number, Order_statement, Order_checkbook and so on,

which implement system actions, can be identified. The system maintained by the program is minimal. The user services operate independently and do not interact with each other. An object-oriented design would be similar and would probably not be significantly more maintainable.

As object-oriented design has become more widely used, some people have suggested that Function-oriented design is obsolete, and should be superseded by an object-oriented approach.

The activities of Function-oriented design are:

(1) **Data-flow design** Model the system design using data-flow diagrams. This should show how data passes through the system and is transformed by each system function. This model may be derived from data-flow models developed during the analysis process.

(2) **Structural decomposition** Model how functions are decomposed into sub-functions using graphical structure charts.

(3) **Detailed design description** Describe the entities in the design and their interfaces. These descriptions may be recorded in a data dictionary. Also describe the control structure of the design using a program description language (PDL) which includes conditional statements and looping constructs.

As with all design processes, these activities are not carried out in sequence but are interleaved during the design process.

5.4.3 Data –flow design

Data-flow design is concerned with designing a sequence of functional transformations that convert system inputs into the required. The design is represented as data-flow diagrams. These diagrams illustrate how data flows through a system and how the output is derived from the input through a sequence of functional transformations.

Data-flow diagrams are a useful and intuitive way of describing a system. They are normally understandable without special training, especially if control information is excluded. They show end-to-end processing that is, the flow of processing from when data enters the system to where it leaves the system can be traced.

Data-flow design is an integral part of a number of design methods and most CASE tools support data-flow diagram creation. Different methods

may use different icons to represent data-flow diagram entities but their meanings are similar. The notation which use is based on the following symbols:

- **Rounded rectangles** represent functions, which transform inputs to outputs. The transformation name indicates its function.
- **Rectangles** represent data stores. Again, they should be given a descriptive name.
- **Circles** represent user interactions with the system which provide input or receive output.
- **Arrows** show the direction of data flow. Their name describes the data flowing along that path.
- The **keywords 'and' and 'or'.** These have their usual meanings as in Boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

This notation is shown in Figure 5.10 below which shows a data-flow design of a design report generator

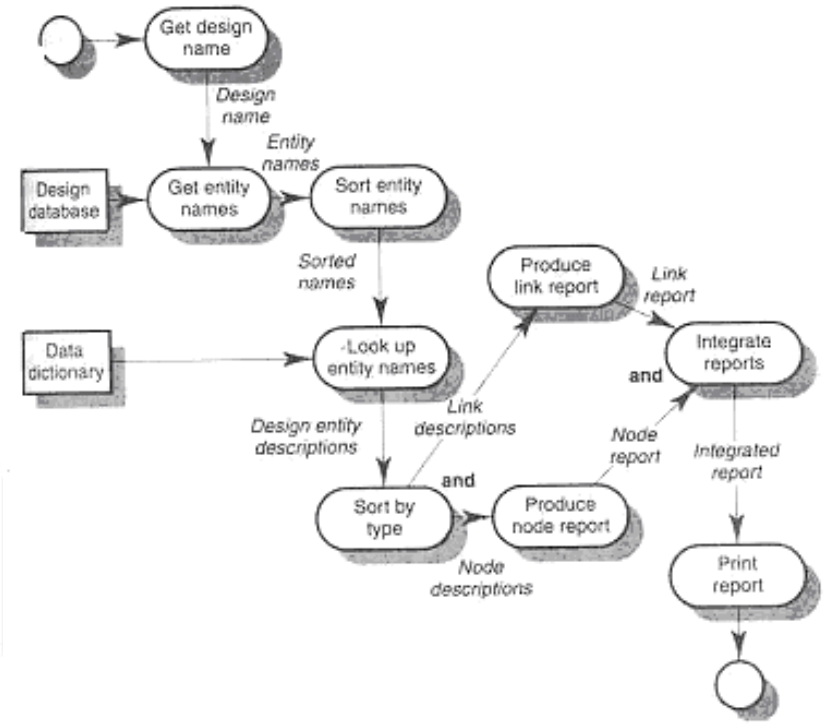


Fig. 5.10 data flow diagram of a design report generator

5.5 Structural Decomposition

It is useful to develop a structural system model. This structural model shows how a function is realized by a number of other functions, which it calls. Structure charts are a graphical way to represent this decomposition hierarchy. Like data-flow diagrams, they are dynamic rather than static system models. They show how the static blocks structure of a function or procedure.

A function is represented on a structure chart as a rectangle. The hierarchy is displayed by linking rectangles with lines. Inputs and outputs (which may be implemented either as parameters or shared variables) are indicated with annotated arrows. An arrow entering a box implies input, leaving a box implies output. Data stores are shown as rounded rectangles and user inputs as circles.

Three process steps, which follow these guidelines, can be identified for the transformation process from data-flow diagram to structure chart:

Identify system-processing transformations: These are the transformations in the diagram, which are responsible for central processing functions. They are not concerned with any input or output functions such as reading or writing data, data validation or filtering or output formatting. These transformations should be grouped under a single function at the first level in the structure chart.

Identify input transformations: These are concerned with reading data, checking it, removing duplicates, and so on. These should be grouped under a single function at the first level in the structure chart.

Identify output transformations: These are transformations, which prepare and format output or write it to the user's screen or other devices.

5.5.1 Detailed design

At this stage in the design process, the designer should know the organization of the design and what each function should do. Design entity description is concerned with producing a short design specification (sometimes called a minispec) of each function. This describes the function, its inputs and its outputs. Making this information explicit usually reveals flaws in the initial decomposition or functions, which have been omitted. The

data-flow diagrams and structure charts must be revisited and modified to incorporate the improved understanding of the design.

The best way to manage these functional descriptions is to maintain them in a data dictionary. They can also be used to record information about design entities. Maintaining names and descriptions in a data dictionary reduces the chances of mistakenly reusing names and provides design readers with insights into the designer's thinking.

Data dictionary entries can vary in detail from a short informal description to a specification of the function in a design description language. Figure 5.11 shows some of the data dictionary entries that might be made for the design report generator. As you can see, it includes information about data as well as the functions in the system.

Entity Name	**Type**	**Description**
Design Name	STRING	The name of the design assigned by the design engineer
Get Design name	FUNCTION	Input : Design name Function: This function communicates with the user to get the name of the design that has been entered in the design database. Output : Design name
Get Entity names	FUNCTION	Input: Design Name Function: Given a design name this function accesses the design database to find the names of the entities (nodes and Links) in that design. Output : Entity name
Sorted Names	ARRAY of STRING	A list of the names of the entities in a design held in the ascending alphabetical order

Fig. 5.11: Data dictionary entries for the design report generator

Some **CASE** tools may include facilities, which allow data dictionaries to be accessed at the same time as a design diagram. This allows information about individual entities to be viewed at the same time as the diagram showing all entities and their relationships. The tool may allow a design entity to be selected then display the corresponding information from the data dictionary. Figure 5.12 shown below is an example of this facility.

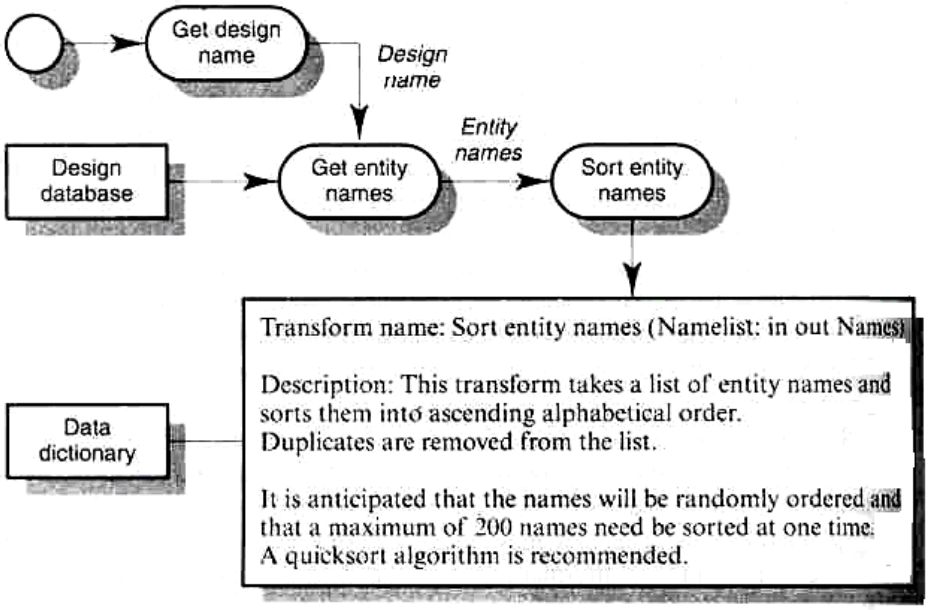


Fig. 5.12: Information about design entities from a data dictionary

The next stage of the functional design process is to produce detailed design for each part of the design. These detailed designs should include control information and more precise information about the data structures manipulated. The detailed design is expressed using some program description language, in some more detailed graphical notation or directly in a programming language.

Self Assessment Questions

4. _____ is concerned with specifying the detail of the object interfaces.
5. A _____ strategy relies on decomposing the system into a set of interacting functions with a centralized system state shared by these functions.
6. _____ is concerned with designing a sequence of functional transformations that convert system inputs into the required.

5.6 Summary

- The main design activities in the software process are architectural design, system specification, interface design, component design, data structure design and algorithm design.

- Functional decomposition involves modeling a system as a set of interacting functional units, Objects oriented decomposition models the system as a set of objects where an object is an entity with state and functions to inspect and modify that state.
- Functional oriented and object-oriented design are complimentary rather than opposing design strategies. Different perspectives may be applied at different levels of design abstraction.
- The software architecture is responsible for deriving an overall structural model of the system, which identifies sub-systems and their relationships. Architect may also design a control model for the system and decompose sub-systems into modules.
- Large systems rarely confirm to a single architectural model. They are heterogeneous and incorporate different models at different levels of abstraction.
- System decomposition models include repository models, cline-server models and abstract machine models. Repository models share data through common store. Client –server models distribute data. Abstract machine models are layered with each layer implemented using facilities provided its foundation layer.
- Examples of control models include centralized control and event models. In centralized model, control decisions are made depending on the system state, in event models external events control the system.
- Examples of modular decomposition models include data-flow and object models. Data-flow are functional, where as object models are based on loosely coupled entities which maintain their own state and operations.
- Domain specific architecture models are abstraction over an application domain. Domain-specific models may be generic models which are constructed bottom-up from existing systems, or reference models which are idealized, abstract models of the domain.
- Object oriented design is a means of designing with information hiding. Information hiding allows the information representation to be changed without other extensive system modifications.

- An object is an entity, which has a private state. It should have constructor and inspection functions allowing its state to be inspected and modified. The object provides services to other objects.
- Object identification is the major problem in object oriented design. One way to identify objects is to consider the nouns (objects) and verbs (operations) in short system descriptions. Other approaches are based on identifying and tangible entities in the application domain, on behavioral analysis and on scenario analysis.
- Function oriented design relies on identifying functions which transform their inputs to create outputs in most systems, functions share some global system state.
- The functional design process involves identifying data transformations of the system, decomposing functions into a hierarchy of sub-functions, describing the operation and interface of each system entity and documenting the flow of control in the system.
- Data flow diagram is a means of documenting end-to-end data flow through the system they do not include control information.
- Data-flow diagram can be implemented directly as a set of cooperating sequential processes. Each transform in the data flow diagram is implemented as a separate process.
- Functional design and object-oriented design usually result in totally different system decomposition. However, the most appropriate design strategy is often a heterogeneous one in which both functional and object-oriented approaches are used.

5.7 Terminal Questions

1. Explain Object Oriented Design in detail.
2. Explain Object Interface Design with suitable examples.
3. What do you mean by Structural Decomposition? Explain.

5.8 Answers

Self Assessment Questions

1. Object oriented design
2. Object

3. Aggregation
4. Object interface design
5. Function oriented design
6. Data flow design

Terminal Questions

1. Object–oriented design is a design strategy based on information hiding. It differs from the function approach to design in that it views a software system as a set of interacting objects, with own private state, rather than as a set of functions that share a global state. (Refer section 5.2)
2. Object Interface design is concerned with specifying the detail of the object interfaces. This means defining the types of the object attributes and the signatures and the semantics of the object operations. If an object-oriented programming language is being used for implementation, it is natural to use it to express the interface design. (Refer section 5.4)
3. It is useful to develop a structural system model. This structural model shows how a function is realized by a number of other functions, which it calls. Structure charts are a graphical way to represent this decomposition hierarchy. Like data-flow diagrams, they are dynamic rather than static system models. They show how the static blocks structure of a function or procedure. (Refer section 5.5)

Unit 6 Assessment of Process Life-Cycle Models

Structure:

6.1 Introduction

This unit discusses the essential purposes and roles of software engineering processes. It begins with criticism of existing models and general proposals that have been made for assessing and evaluating models. The critical role of time as a factor in development is considered, including not only the various scheduling constraints on time to develop, but also the business-driven parameter of time to market. The lack of an adequate integration between software and hardware technology, on the one hand, and business and social disciplines, on the other, is identified as a persistent shortcoming undermining the ability of the development process to attack real-world problems optimally.

Next, a series of questionable assumptions that have affected the historical development of software process models are considered, including suppositions about the primacy of the role of internal software factors; the relative independence of software development from the business process; separation of the software project as management enterprise from the software process; and a choice between process-centered versus architecture-centered development. These assumptions have illegitimately constrained and reduced the fundamental role that must be played by

people, money, interdisciplinary knowledge, and business goals in terms of their impact on effective problem solution.

Objectives:

After studying this unit, you should be able to:

- explain the overview of the assessment of process & dimension of time
- describe the need for a business model in software engineering
- discuss classic invalid assumptions
- explain the implication of the new business model & role of the problem solving process.

6.2 Overview of the Assessment of Process

The elements of a redefined software engineering process are identified based on the integration of critical process tasks or activities; required major interdisciplinary resources (people, money, data, exploratory and modeling tools, and methodologies); organizational goals; and the impact of time as components of an ongoing roundtrip approach to business-driven problem solving. The redefinition addresses limitations identified in the literature related to business evaluation metrics; the process environment and external drivers; and process continuation, as fundamental to process definition.

The idea of a software process model that fits every project seems far-fetched because any project has so many aspects that it is difficult to capture every potential aspect in a single perspective (Liu & Horowitz 1989). However, this has not prevented the development of process models that have attempted to capture the essential benefits of previous software models in a unified manner. Humphrey (1988) observed that “since the software engineering process used for a specific project should reflect [that project’s] particular needs, a framework is needed to provide consistency between projects.” Liu and Horowitz (1989) argued against unification in models, but proposed essential features that every successful model should have. These included the ability to describe the development process as a design process; address parallel processing in large-scale projects; map the diverse set of conditions that exist prior to development activities; debug the process by locating failed activities and resources; and allocate sufficient resources for each activity in the development project.

Some have argued against attempting to structure and manage the software development process because of the overwhelming differences that exist across different projects, firms, and cultures. However, Blackburn, Scudder, and Van Wassenhove (1996) argued to the contrary, observing that worldwide similarities in management of the process are more prevalent than differences. Considerable effort has been made to establish custom solutions based on existing process models. Although a few of these efforts have tried to tailor or match process models to specific project needs, many have attempted to provide evaluation criteria or metrics; mechanisms for evolution and improvement; unified frameworks or taxonomies; and supporting tools and environments. Other studies have tried to address the general issue of process description or abstraction by constructing a conceptual process framework, rather than by evaluating existing process models (Armitage & Kellner 1994). These have served as a basis for the process representation or transformation that has assisted in the review, development, and improvement of process models. An understanding of these efforts will contribute to one of the main objectives, which is to build a more comprehensive taxonomy of process models.

In an overall sense, process models are used to enable effective communication; facilitate process reuse; support process evolution; and facilitate process management.

- Humphrey and Kellner (1989) suggested that, to evaluate the effectiveness of a process model, one should consider its ability to represent the real-world application as well as the way in which work is actually done; provide a flexible, understandable, and powerful framework for representing and improving the software development process, and be refineable or resolvable to any required level of detail or specification.
- Curtis, Kellner, and Over (1992) identified five uses for process models: facilitating human understanding and communication, supporting process improvement, supporting process management, automating process guidance, and automating execution support. All of these uses can also be considered as evaluation criteria for process models as well.
- Sutton (1988) asserted that, for a process model to be effective, it must exhibit “multidimensional” characteristics, including the ability for

decomposition adequately to capture the details of the work to be done; the ability to provide complete coverage of all the activities of the software life cycle; the ability to reflect the distributed nature of the development process including the potential for sequential and parallel processing; and the ability to incorporate related interdisciplinary models from areas such as project and configuration management, software evaluation, and software acquisition into a single system development.

- Madhavji and colleagues (1994) proposed a method for eliciting and evaluating process models that entailed understanding the organizational environment (organizational, process, and project issues); defining objectives, including model and project-oriented objectives; planning the elicitation strategy, developing process models, validating process models, analyzing process models, post-analysis, and packaging. According to these authors, the basic reasons for using software process models were to produce software of high quality that met budget and time requirements and to do so as far as possible by means of automated tools.
- Khalifa and Verner (2000) focused on the Waterfall and prototype models in their empirical study, emphasizing the factors driving the usage of specific process models: depth and breadth of use and facilitating conditions (the size of the development team, organizational support, and the speed with which new methodologies were adopted).
- According to Boehm and Belz (1990), the critical process aspects were requirements growth, the need to understand complex requirements, the need for robustness, available technology, and architectural understanding. They used these as a baseline for a software process model elicitation procedure.
- Blackburn et al. (1996) identified the five most influential factors in the development process as development time, project characteristics, team size, allocation of time in project stages, and development language selection. The approach was based on a strong correlation between process optimization and software product metrics from a project management prospective.

Madhavji et al. observed that recognized benefits of life-cycle models included the ability to enhance process understanding, determine global

activities, reduce cost, improve quality, methods, and tool effectiveness, and improve stakeholder satisfaction. Using estimation techniques, the models addressed problems of resource management such as time and manpower. They also provided predictive capabilities with respect to primary performance measures and captured some of the variability and uncertainty associated with the software development process (Martin *&* Raffo 1997). However, the models tended to fall short on overall problem comprehension, detailed description, and the ability to adapt or tailor to changing project requirements. They focused more on product engineering than the many elemental process building blocks essential to project management and control (Curtis et al. 1992).

Krasner et al. (1992) criticized the models for their tendency to "focus on series of artifacts that exist at the end of phases of the life cycles, rather than on the processes that are conducted to create the artifacts" in the first place. According to Madhavji et al., these traditional process models led to low software process maturity and difficulties in managing and controlling software processes. Their over-reliance on the Waterfall Model encumbered them with its negative side effects, such as enforcing one-way development by managers, inhibiting creativity based on design and requirements trade-off, and corrupting measurement and tracking systems in processes (Humphrey & Kellner 1989). The conventional models also tended to impose extensive documentation requirements without providing commensurate added value (Krasner et al. 1992).

Humphrey and Kellner (1989) attributed the problems with conventional process models to inaccurate representations on their part of the behavioral aspects of what occurs during software development because of an overly intense focus on task sequencing. Boehm (1996), on the other hand, attributed their weaknesses to factors such as a lack of user-interface prototyping, fixed requirements, inflexible point solutions, high-risk downstream capabilities, and off-target initial releases. According to Boehm, recognition of such problems "led to the development of alternative process models such as risk-driven, reuse-driven, legacy-driven, demonstration-driven, design-to-COT-driven, and incremental, as well as hybrids of any of these with the waterfall or evolutionary development models."

Ropponen and Lyytinen (2000) elaborate on the need for risk management in process model assessment, including risks related to scheduling and timing, system functionality, subcontracting, requirements management, resource usage and performance, and personal management. Madhavji et al. proposed combining process-detailed understanding and process support to address change or volatility in process-centered software environments. They identified several categories or perspectives from which a process model could be viewed in terms of its static and dynamic properties, process steps, artifacts, roles, resources, and constraints.

The analysis by Martin & Raffo (1997) recognized two major approaches in software development: process models and system dynamics. The latter is important in developing an intuitive understanding of how a project will behave under different management polices and alternatives and benefits significantly from simulation techniques. Abdel–Hamid and Mad-nick (1989) used system dynamics to model project risks such as delays, pressures, and unknown problems at different project levels. However, Raffo and Martin expanded this idea by introducing a continuous simulation framework (1997) that was consistent with the process improvement paradigm inspired by the CMM model (Martin & Raffo 1997; Paulk et al. 1993). Indeed, Boehm recognized the Abdel–Hamid and Madnick model as a realistic contribution to quantitative models of software project dynamics, although he was still concerned about the lack of a quantitative model of software life-cycle evolution (Boehm 1984). Clearly, project management and software economics perspectives have gained greater attention as critical elements in the assessment of process models.

Some of the research on model assessment has focused on classifying process models. Blum (1994) arranged development methods according to a matrix, depending on their focus of interest (the problem or product involved) and the form of representation used (conceptual or formal). Boehm & Port (1999) and Boehm & Belz (1990) addressed the conflicts that occur when a combination of product, process, property, and success models is adopted, leading to model clashes. They proposed taxonomy of model clashes in an effort to resolve the resulting conflicts.

Over the past decade, the trend towards process improvement has been increasing and turning away from fixed, conventional process models. Thus,

Bandinelli et al. (1995) observed that “there has been an increasing interest in the development of frameworks and guidelines to support the evaluation of software process maturity and to identify strategies and key areas of improvement.” These authors built on experiences learnt from the Capability Maturity Model, Bootstrap, Kaizen, QIP, SPMS, and other models in developing a feedback-loop model for software development organizations.

The model was intended to address problems with discrepancies; descriptions, comprehension, visibility, and traceability among the different process forms (desired, perceived, observed, and actual process). They used the feedback-loop model as a baseline in experiments aimed at improving process maturity. Two inferences can be made from their research. A process model is no longer a fixed model that fits in with a fixed problem definition, but instead dynamically evolves over time in response to changes in the problem until at some point it stabilizes. Second, the ability to capture a real-world situation (actual process) was and still is the most significant issue in assessing process models. The closer that a representation is to the actual situation, the more likely it is to be effective.

Basili and Rombach (1988) proposed the improvement-oriented TAME process model, which is based on a goal–question–metrics (GQM) approach. It includes separate components for characterizing the current status of a project environment, integrating planning for improvement into the execution of projects, executing the construction and analysis of projects, recording project experiences into an experience base, and distributing information across the model and its components. They claim such component integration distinguishes their model from traditional process models that have only partially addressed such issues. They assert that even recently developed process models have not been able to “completely integrate all their individual components in a systematic way that would permit sound learning and feedback for the purpose of project control and improvement of corporate experience.”

Kadary and colleagues (1989) raised important questions about the need for or even possibility of a generic paradigm for software life cycles, aspects of such a generic model, and its potential role in industrial practice. Their challenge is difficult because the issue is not yet well structured and one can think of many alternatives that need further testing and assessment.

The research on assessments may be summarized as follows:

- *Metric-oriented assessments* framed or synthesized processes and provided standards and metrics for further process enhancement and evaluation, as described in the work of Humphrey *&* Kellner (1989); Sutton (1988); and Curtis et al. (1992). The metrics took the form of factors or goals, as in Boehm and Belz (1990); Madhavji et al. (1994); Khalifa and Verner (2000); and Blackburn et al. (1996). Some assessments suggested elicitation procedures or plans as well (Jaccheri, Picco, *&* Lago 1998; Madhavji et al. 1994).
- Unified model or *taxonomy-driven assessments* surveyed as many models as possible in an attempt to build a classification or taxonomy (Blum 1994) or make comprehensive conclusions regarding a unified process model derived through a broad selection and understanding of process models (Jacobson et al. 1999).
- *Process improvement assessments* come from the perspective that existing models are insufficient and need enhancements and new architectures, as described in Bandinelli et al. (1995); Basili and Rombach (1988); El-Emam and Birk (2000); and Baumert (1994). The Capability Maturity Model has been the official reference platform for this approach, in addition to efforts to integrate it with ISO9000 standards. Some of the assessments have focused on dramatic change rather than incremental development.
- Tool support and *software environment-based assessments* incorporated automated tools into process modeling. Some have even proposed frameworks for process model generation (Boehm *&* Belz 1990). These approaches have focused more on software development environments and included tool support to build more sophisticated process models using CASE tools and automation, as described in Osterweil (1997) and Ramanathan and Soumitra (1988). This and the process improvement category overlap substantially.

6.3 The Dimension of Time

Time has been the critical factor in software development from its beginnings; the original motivation for interest in computing was the computer's ability to carry out tasks faster than could be done otherwise. Computer

hardware provided fast processing power and high-speed memories provided fast storage. Software adapted this technology to the needs of individuals and organizations to address problems in a timely manner. It took only a while to recognize that building effective software required more than just the time needed to write the source code for a software product. Experience underscored the obvious: software was only valuable when it met people's needs and created value. Software came to be viewed as a system that emerged during the course of multiple, evolutionary, interdisciplinary life-cycle phases, rather than a one-shot effort composed from a largely technical perspective.

Accordingly, the objective of development shifted dramatically, from saving time in the short term to saving time in the long term, with software production recognized as a lengthy process that was engaged in developing solutions compliant with stakeholder requirements. This decisive attitudinal change was the first step in transitioning software development from coding to engineering, where business goals drove software construction and not vice versa.

Of course, the short-term effect of the time factor was not cost free. Software economics has underscored the importance of the time value of money in assessing the actual costs and benefits of a software project in terms of discounted cash flow, net present value, return on investment, and break-even analysis. Additionally, business and technology have undergone dramatic – even revolutionary – changes during the historic time-line of software development, creating new demands and facilitating new capabilities. From any perspective, time repeatedly plays a key role in software development and its evolution.

Thus, a firm's failure to respond to new business requirements within an adequate *time to market* can result in serious losses in sales and market share; failing to exploit new enabling technologies can allow advantageous advances to be exploited by competitors. Although it is derived from a business context, this time-to-market notion now plays a major role in software process paradigms. The implication is that short-term cycle time must become shorter and, at the same time, the features and expected quality of the final system must be retained. This is the new challenge faced by software development: building quality systems faster. The required

acceleration of the software development process entails an extensive body of methodologies and techniques such as reusability; CASE tools; parallel development; and innovative approaches to project management.

Self Assessment Questions

1. In an overall sense, process models are used to enable effective _____.
2. _____ and _____ perspectives have gained greater attention as critical elements in the assessment of process models.
3. _____ has been the critical factor in software development from its beginnings.

6.4 The Need for a Business Model in Software Engineering

Software engineering faces several dilemmas. It has comprehensive goals, but limited tools. It demands broad perspectives, but depends on narrowly focused practitioners. It places a high premium on quality, but often has insufficient inputs to its problem-solving process. As a field, software engineering has yet to define theories and frameworks that adequately combine the disciplines of software and hardware technology with related business and social science disciplines to attack real-world problems optimally. Despite advances, software engineering tends to remain code driven and is burdened in testing for bugs, program errors, and verification, even though reusable objects, reusable applications, and CASE tools have long been available.

The engineering of software entails inviting software technology to help tackle human problems rather than just shoehorning human problems into a software solution. This requires reordering the relation between people and computers; computer programs are understood to play an important but limited role in problem-solving strategy. Such an approach to software engineering would still be software driven in the sense that it was driven by the need to develop software for automated as opposed to manual problem solving. However, it would view problems and evaluate solutions from a broadly interdisciplinary perspective in which software was understood and used as a tool.

Requirements engineering is supposed to address the problem part of software engineering, but it is part of the traditional view that looks at the problem-solving process as a phase in the software development life cycle,

rather than at the software development life-cycle as part of the problem-solving process. The software development life cycle never ends with a solution, but only with a software product. Although one may assume that a software product should be the solution, in practice this never happens because software systems are only part of a total organizational context or human system; one cannot guarantee that these solutions are effective independently of their context.

6.5 Classic Invalid Assumptions

Four unspoken assumptions that have played an important role in the history of software development are considered next.

6.5.1 First Assumption: Internal or External Drivers

The first unspoken assumption is that *software problems are primarily driven by internal software factors*. Granted this supposition, the focus of problem solving will necessarily be narrowed to the software context, thereby reducing the role of people, money, knowledge, etc. in terms of their potential to influence the solution of problems. Excluding the people factor reduces the impact of disciplines such as management (people as managers); marketing (people as customers); and psychology (people as perceivers). Excluding the money factor reduces the impact of disciplines such as economics (software in terms of business value cost and benefit); financial management (software in terms of risk and return); and portfolio management (software in terms of options and alternatives). Excluding the knowledge factor reduces the impact of engineering; social studies; politics; language arts; communication sciences; mathematics; statistics; and application area knowledge (accounting, manufacturing, World Wide Web, government, etc).

It has even been argued that the entire discipline of software engineering emerged as a reaction against this assumption and represented an attempt to view software development from a broader perspective. Examples range from the emergence of requirements engineering to the spiral model to human – computer interaction (HCI). Nonetheless, these developments still viewed non-software-focused factors such as ancillary or external drivers and failed to place software development in a comprehensive, interdisciplinary context. Because software development problems are

highly interdisciplinary in nature, they can only be understood using interdisciplinary analysis and capabilities. In fact, no purely technical software problems or products exist because every software product is a result of multiple factors related to people, money, knowledge, etc., rather than only to technology.

6.5.2 Second Assumption: Software or Business Processes

A second significant unspoken assumption has been that the *software development process is independent of the business processes in organizations*. This assumption implied that it was possible to develop a successful software product independently of the business environment or the business goals of a firm. This led most organizations and business firms to separate software development work, people, architecture, and planning from business processes. This separation not only isolated the software-related activities, but also led to different goals, backgrounds, configurations, etc., for software as opposed to business processes. As a consequence, software processes tended to be driven by their internal purposes, which were limited to product functionality and not to product effectiveness.

This narrow approach had various negative side effects on software development. For example, the software process was allowed to be virtually business free. Once the product was finalized, it was tested and validated only for functionality, as opposed to being verified for conformity to stakeholder goals. As a result, even if the product did not effectively solve the underlying business problems or create a quantifiable business value for the organization, it could still pass its test. Because software development was not synchronized with the business process, software problems could be “solved” without actually solving business problems.

6.5.3 Third Assumption: Processes or Projects

A third unspoken assumption was that the *software project was separate from the software process*. Thus, a software process was understood as reflecting an area of computer science concern, but a software project was understood as a business school interest. If one were a computer science specialist, one would view a quality software product as the outcome of a development process that involved the use of good algorithms, data base design, and code. If one were an MIS specialist, one would view a

successful software system as the result of effective software economics and software management.

This dichotomy ignored the fact that the final product was identical regardless of who produced it or how it was produced. The assumption reinforced the unwise isolation of project management from the software development process, thus increasing the likelihood of product failure. In contrast to this assumption, interdisciplinary thinking combines the process with the project; computer science with the MIS approach; and software economics with software design and implementation in a unified approach. Just as in the case of the earlier assumptions, this assumption overlooks the role of business in the software development process.

6.5.4 Fourth Assumption: Process Centered or Architecture Centered

There are currently *two broad approaches in software engineering; one is process centered and the other is architecture centered*. In process-centered software engineering, the quality of the product is seen as emerging from the quality of the process. This approach reflects the concerns and interests of industrial engineering, management, and standardized or systematic quality assurance approaches such as the Capability Maturity Model and ISO. The viewpoint is that obtaining quality in a product requires adopting and implementing a correct problem-solving approach. If a product contains an error, one should be able to attribute and trace it to an error that occurred somewhere during the application of the process by carefully examining each phase or step in the process.

In contrast, in architecture-centered software engineering, the quality of the software product is viewed as determined by the characteristics of the software design. Studies have shown that 60 to 70 percent of the faults detected in software projects are specification or design faults. Because these faults constitute such a large percentage of all faults within the final product, it is critical to implement design-quality metrics. Implementing design-quality assurance in software systems and adopting proper design metrics have become key to the development process because of their potential to provide timely feedback. This allows developers to reduce costs and development time by ensuring that the correct measurements are taken from the very beginning of the project before actual coding commences. Decisions about the architecture of the design have a major impact on the

behavior of the resulting software – particularly the extent of development required, reliability, reusability, understandability, modifiability, and maintainability of the final product, characteristics that play a key role in assessing overall design quality.

However, an architecture-centered approach has several drawbacks. In the first place, one only arrives at the design phase after a systematic process. The act or product of design is not just a model or design architecture or pattern, but a solution to a problem that must be at least reasonably well defined. For example, establishing a functional design can be done by defining architectural structure charts, which in turn are based on previously determined data flow diagrams, after which a transformational or transitional method can be used to convert the data flow diagrams into structure charts. The data flow diagrams are outcomes of requirements analysis process based on a preliminary inspection of project feasibility. Similarly, designing object-oriented architectures in UML requires first building use-case scenarios and static object models prior to moving to the design phase.

A further point is that the design phase is a process involving architectural, interface, component, data structure, and database design (logical and physical). The design phase cannot be validated or verified without correlating or matching its outputs to the inputs of the software development process. Without a process design, one could end up building a model, pattern, or architecture that was irrelevant or at least ambivalent because of the lack of metrics for evaluating whether the design was adequate. In a comprehensive process model, such metrics are extracted from pre-design and post-design phases. Finally, a process is not merely a set of documents, but a problem-solving strategy encompassing every step needed to achieve a reliable software product that creates business value. A process has no value unless it designs quality solutions.

6.6 Implications of the New Business Model

The following consequences result when one refocuses from engineering software for the sake of the technological environment to engineering software for people's sake:

- Solutions will evolve only from carefully understood problems. The resulting solutions will be guided by their originating problems and

considered successful only if they are able to solve those problems. The solution is never solely the software product, but everything needed to solve the problem.

- Problems will not be defined in terms of what the people want the software to do for them. Problem definition will address the relevant human needs, regardless of the role of the software in meeting those needs. Subsequent to an interdisciplinary definition of the problem, an interdisciplinary solution will be proposed that will utilize the available, relevant human, financial, informational, technological, and software resources.
- The iterative software development process will become part of the synchronized business process and will in turn deliver business process total solutions. Thus, the business process will shape the software process in terms of its goals, metrics, and requirements.

Self Assessment Questions

4. The engineering of software entails inviting _____ to help tackle human problems rather than just shoehorning human problems into a software solution.
5. _____ is supposed to address the problem part of software engineering.
6. In a comprehensive process model, metrics are extracted from _____ and _____ phases.

6.7 Role of the Problem-Solving Process in this Approach

A solution represents the final output from a problem-solving process. To obtain reliable solutions, the problem-solving process must receive all the requisite inputs. The more comprehensive, carefully defined and well-established these inputs are, the more effective the solutions will be. Regardless of whether one uses a manual or computerized system to tackle a problem, the problem-solving process can properly operate only when it has sufficient relevant data, a well-defined problem, and appropriate tools.

6.7.1 Data

The importance of data, raw facts, or statistical summaries of raw facts in solving problems is decisive for the scientific method. Obviously, without adequate data, it is difficult to measure or estimate the “distance” from the current situation to a desired situation. Two basic problems that arise with

data are:

- Data may be insufficient to provide a deep understanding of a problem domain or situation. It is hard to establish a solid point of reference in the context of insufficient data. Ambiguity may be substantial and uncertainty exacerbated.
- Data may be excessive; making it difficult to identify or distinguish which data is relevant or significant to the problem under consideration. An excess of data can lead to time wasted pursuing false directions, thus causing delays in solution and possibly allowing further development of system-degrading problems.

Defining an accurate context for business problems is critical to identifying and obtaining relevant data. Effective data capture requires data collection and data mining. Appropriate tools are needed to ensure the quality and speed of the collection and mining processes. In data collection, relevant data is explored to provide a comprehensive picture of the problems; this requires creative thinking and the use of diverse resources, disciplines, and skills. Interdisciplinary tools are essential. In data mining, the collected data is further examined and filtered to obtain the significant data. Once again, interdisciplinary tools are essential to extracting the relevant data.

6.7.2 Problem Definition

Once sufficient data is available, problems may at least potentially be properly defined. Problem definition is the foundation of an effective problem-solving process. It entails structuring, interpreting, and analyzing data. Subsequent to the initial data analysis, one can arrive at a reliable problem definition that captures the essential aspects of the situation. Problem definition depends heavily on data, so if data is inaccurate, incomplete, irrelevant, or insufficient, problems cannot be appropriately defined. On the other hand, determining which data to seek depends on the problem's initial definition, which sets the context or circumstance in which data is gathered.

6.7.3 Tools and Capabilities

Tools can support the problem and solution level of the problem-solving process, helping to automate and accelerate the entire process. Given adequate tools, accurate and reliable data can be determined and

maintained. Tools can assist in data mining raw facts and elicit the important, relevant data. In the present context, tools encompass all the available capabilities that can be brought to bear during a problem-solving process; these tools are as critical as a knowledge base and personnel abilities. CASE tools are popular because of their applicability in the software development process, but the role of tools transcends software development. They encompass analysis, design, and implementation and extend to support the entire business process of an organization.

Interdisciplinary thinking is a necessity because a one-sided view of a problem will not reveal a comprehensive picture of a business or organizational problem. In an interdisciplinary approach to problem solving, a comprehensive diagnosis of the business problem is an essential part of the process. Interdisciplinary thinking not only permits a full analysis of a business problem, but also enables problem-solvers to take advantage of existing interdisciplinary abilities. An understanding of different disciplines is the basic enabling factor that allows incorporation of interdisciplinary thinking in problem solving. Bringing this knowledge to bear requires identifying and eliminating sources of ignorance about a problem. Interdisciplinary thinking also reflects an appreciation of the role of diversity. In fact, it can be viewed as a means for integrating diversity in a productive and effective manner. It allows recognition, measurement, and utilization of differences in terms of their positive role in problem solving. The following sections identify how ignorance and diversity influence problem solving.

6.8 Redefining the Software Engineering Process

The definition of the development process has long been disputed in software engineering literature. Computer scientists tend to view the process as one of providing a theoretical framework with the objective of systematically producing cost-effective software products. Project managers view the process as a way of partitioning projects into activities to provide guidelines for project managers. The problem with both views is their narrow focus on the activities that should be involved and on how the activities should be ordered or scheduled. Both views lack effective ways for optimizing the process to achieve real-world problem solving, which requires additional components beyond those that organize an identified set of

activities in a particular order. These components include:

- A means for evaluating the process against the business goals of an organization
- A means for recognizing and responding to the diverse environmental, external, or interdisciplinary factors in the context of which the process functions or operates
- A means for identifying in what way the different process activities are interrelated or interdependent in terms of fulfilling organizational goals

Although many efforts have been made to establish or define metrics for assessing the quality of the software process and product, these metrics never seem to be part of the underlying process definition and rarely have clear connections to the external drivers in the surrounding environment. Consequently, software process definitions have generally lacked environmental interactivity and business purpose, although there have been some notable exceptions. For example, the Capability Maturity Model approach was introduced by the Software Engineering Institute partially in an attempt to define a process model responsive to changes and pressures from environmental factors. Overall, however, most approaches have paid little attention to ensuring that interdisciplinary resources are integrated into software process activities to advance business processes and goals.

6.8.1 Round-Trip Problem-Solving Approach

The software engineering process represents a *round-trip framework* for problem solving in a business context in several senses.

- The software engineering process is a problem-solving process entailing that software engineering should incorporate or utilize the problem-solving literature regardless of its interdisciplinary sources.
- The value of software engineering derives from its success in solving business and human problems. This entails establishing strong relationships between the software process and the business metrics used to evaluate business processes in general.
- The software engineering process is a round-trip approach. It has a bidirectional character, which frequently requires adopting forward and reverse engineering strategies to restructure and reengineer information systems. It uses feedback control loops to ensure that specifications are

accurately maintained across multiple process phases, reflective quality assurance is a critical metric for the process in general.

- The non-terminating, continuing character of the software development process is necessary to respond to ongoing changes in customer requirements and environmental pressures.

6.8.2 Activities

The software engineering process comprises a set of interrelated activities that mutually require and support each another. Although the activities vary in terms of their names, labels, degrees of abstraction, or resolution, they always include the following steps:

- A well-defined process and project, as well as a well-defined problem identified through diagnosis and analysis
- A well-defined solution obtained through design and software architecture and based on the problem definition
- An accurate and precise execution of the defined solution obtained through implementation and installation
- Well-defined testing processes that use business and quality assurance metrics: testing, validation, verification, and quality assurance
- Continual improvement or adjustment of the implemented solution in response to customers, changes, competition, reengineering, and maintenance

6.8.3 Goals

The software engineering process must be guided, assessed, and evaluated by ongoing references to the business goals of an organization, which guide the entire process from beginning to end. These define the business case and provide the foundation for the requirements analysis process. They determine the economic and organizational feasibility of the software project. They serve as metrics to assess the performance and quality of the process and of the generated solution. Finally, they motivate continual improvement in the software engineering process.

6.8.4 Interdisciplinary Resources

The software engineering process integrates and uses interdisciplinary resources to execute its activities and meet its goals. Interdisciplinary resources encompass multiple disciplines and a diverse range of knowledge about people, money, data, tools, application knowledge, methodologies,

time, and goals. This inclusive approach implies effectively executing each activity in the process and requires appropriate consideration of the pertinent interdisciplinary resources. Utilization of interdisciplinary resources is closely related to process performance and product quality. Failure to integrate interdisciplinary resources can significantly affect the success of process or project management, accuracy in problem definition, and the effectiveness of the final solution. The integration of interdisciplinary resources represents a critical recognition: namely, the importance of interdisciplinary thinking in software engineering in contrast to the prevailing attitude in conventional approaches. Interdisciplinary resources encompass multiple disciplines and a diverse range of knowledge about people, money, data, tools, application knowledge, methodologies, time, and goals, as described earlier.

6.8.5 Time

The software engineering process depends on time as a critical asset as well as a constraint or restriction on the process. Time can be a hurdle for organizational goals, effective problem solving, and quality assurance.

Managed effectively, time can support the competitive advantage of an organization, but time is also a limitation, restricting or stressing quality and imposing an obstacle to efficient problem solving. Time is the major concern of various stakeholders in the software engineering process, from users, customers, and business managers to software developers and project managers.

Time is closely correlated with money and cost, tools, and the characteristics of development methodologies like Rapid Application Development that aim primarily at reducing time and accelerating the software engineering process. These methodologies exhibit characteristics such as reusability, which emphasizes avoiding reinventing the wheel, object-oriented analysis, design, and implementation. Examples include assembly from reusable components and component-based development, business objects, distributed objects, object-oriented software engineering and object-oriented business process reengineering, utilizing unified modeling languages (UML), and commercial-off-the-shelf software. Other characteristics are automation (via CASE tools), prototyping, outsourcing, extreme programming, and parallel processing.

A redefined software engineering process must integrate the critical activities, major interdisciplinary resources (people, money, data, tools, and methodologies), organizational goals, and time in an ongoing round-trip approach to business-driven problem solving. This redefinition must address limitations identified in the literature related to business metrics, the process environment and external drivers, and process continuation, as fundamentals of process definition. A conceptual framework should emphasize the following characteristics for interdisciplinary software engineering. It must address exploring resources, external drivers, and diversity in the process environment to optimize the development process. It must overcome knowledge barriers in order to establish interdisciplinary skills in software-driven problem-solving processes. It must recognize that organizational goals determine the desired business values, which in turn guide, test, and qualify the software engineering process.

The process activities are interrelated and not strictly sequential. Irrelevant activities not related to or that do not add value to other activities should be excluded. The optimized software engineering process must be iterative in nature with the degree of iteration ranging from internal feedback control to continual process improvement. The software engineering process is driven by time, which is a critical factor for goals, competition, stakeholder requirements, change, project management, money, evolution of tools, and problem-solving strategies and methodologies.

Self Assessment Questions

7. Effective data capture requires _____ and _____.
8. _____ thinking is a necessity because a one-sided view of a problem will not reveal a comprehensive picture of a business or organizational problem.
9. UML stands for _____.

6.9 Summary

The unit started with the importance of considering the time frame in the software development process. We then discussed about the need for a business model and the main four unspoken assumptions involved in the software engineering process. Implications of the new business model and the role of problem solving have been discussed towards the end of the unit.

6.10 Terminal Questions

1. Explain different kinds of assessment techniques.
2. Give the importance of dimension of time in software development.
3. Explain the importance of need for business model in software engineering.

6.11 Answers

Self Assessment Questions

1. Communication
2. Project management, Software economics
3. Time
4. Software technology
5. Requirements engineering
6. Pre-design, post-design
7. Data collection, data mining
8. Interdisciplinary
9. Unified Modeling Language

Terminal Questions

1. Metric oriented assessment, Unified Model or Taxonomy driven assessment, Process improvement assessment, and Software environment based assessment. (Refer section 6.2).
2. Time has been the critical factor in software development from its beginnings. The original motivation for interest in computing was the computer's ability to carry out tasks faster than could be done otherwise. (Refer section 6.3)
3. Software engineering faces several dilemmas. It has comprehensive goals, but limited tools. It demands broad perspectives, but depends on narrowly focused practitioners. It places a high premium on quality, but often has insufficient inputs to its problem-solving process. (Refer section 6.4)

Unit 7 Configuration Management

Structure:

7.1 Introduction

Large software systems may be considered as configuration of components. During their right time, these systems evolve. Many different versions, made up of different component configurations of the system are created. **Configuration management (CM)** is the process, which controls the changes made to a system, and manages the different versions of the evolving software product.

Configuration management involves the development and application of procedures and standards for managing an evolving system product. Procedures should be developed for building systems and releasing them to customers. Standards should be developed for recording and processing proposed system changes and identifying and storing different versions of the system.

The process of changing a system after it has been delivered and is in use is called **software maintenance**. The changes may involve simple changes to correct coding errors, more extensive changes to correct design errors or significant enhancement to correct specification errors or accommodate new requirements. Maintenance therefore, in this context, means evaluation. It is the process of changing the system to maintain its ability to survive.

Software re-engineering is concerned with taking existing legacy systems and re implementing them to make it more maintainable. As part of this re-

engineering process, the system may be re-documented or restructured. It may be translated to a more modern programming language, implemented on a distributed platform rather than mainframe or its data may be migrated to a different database management system.

Software Refactoring is the process of factoring the design module.

Objectives:

After studying this unit, you should be able to:

- explain the concept of configuration management
- describe software maintenance and software re-engineering
- discuss software refactoring

Configuration management planning

Configuration management takes over control of systems after they have been developed. However, planning this management process must start during system development. A configuration management plan should be developed as part of the over all planning process.

The CM plan should include at least the following information:

(1) The definition of what entities are to be managed and a formal scheme for identifying these entities.

(2) A statement of who takes responsibility for the configuration management procedures and for submitting controlled entities to the configuration management team.

(3) The configuration management policies, which are used for, change control and version management.

(4) A description of the records of the configuration management process which should be maintained.

(5) A description of the tools to be used for configuration management and the process to be applied when using these tools.

(6) A definition of the configuration database which will be used to record configuration information.

Other information such as the management of software from external suppliers and the CM process auditing procedures may also be included in the CM plan.

An important part of the CM plan is the definition of responsibilities. It should define who is responsible for the delivery of each document or software component to quality assurance and configuration management. It may also define the reviewers of each document. The person responsible for document delivery need not be the same as the person responsible for producing the document. To simplify interface, it is often convenient to make project managers or team leaders responsible for all of the documents produced by their team.

7.2 Change Management

The change management process should come into effects when the software or associated documentation is put under the control of the configuration management team. Change management procedures should be designed to ensure that the costs and benefits of change are properly analyzed and that changes to a system are made in a controlled way.

Change management processes involve technical change analysis, cost benefit analysis and change tracking. The pseudo-code, shown in table below defines a process, which may be used to manage software system changes:

The first stage in the change management process is to complete a change request form (CRF). This is a formal document where the requester sets out the change required to the system. As well as recording the change required, the CRF records the recommendations regarding the change, the estimated costs of the change and the dates when the change was requested, approved, implemented and validated. It may also include a section where the maintenance engineer outlines how the change is to be implemented.

The information provided in the change request form is recorded in the CM database. These steps are shown in table 7.1.

Table 7.1: Steps in Change Management

```
Request change by completing a change request form
Analyze change request
If change is valid then
    Assess how change might be implemented
    Asses change cost
    Record change to change control board
    Submit request to change control board
  If change is accepted then
      Repeat
        Make changes to software
        Record changes and link to associated change request
        Submit changed software for quality approval
      Until software quality is adequate
      Create new system version
  Else
      Reject change request
Else
      Reject change request
```

Once a change request form has been submitted, it is analyzed to check that the change is valid. Some change requests may be due to user misunderstandings rather than system faults; others may refer to already known faults. If the analysis process discovers that a change request is invalid duplicated or has already been considered the change should be rejected. The reason for the rejection should be returned to the person who submitted the change request.

For valid changes, the next stage of the process is change assessment and costing. The impact of the change on the rest of the system must be checked. A technical analysis must be made of how to implement the change. The cost of making the change and possibly changing other system components to accommodate the change is then estimated. This should be recorded on the change request form. This assessment process may use the configuration database where component interrelation is recorded. The impact of the change on other components may then be assessed.

Unless the change involves simple correction of minor errors on screen displays or in documents, it should then be submitted to a change control board (CCB) who decide whether or not the change should be accepted.

The change control board considers the impact of the change from a strategic and organizational rather than a technical point of view. It decides if the change is economically justified and if there are good organizational reasons to accept the change.

The term 'change control board' sounds very formal. It implies a rather grand group which makes change decisions. Formally structured change control boards which include senior client and contractor staff are a requirement of military projects. For small or medium-sized projects, however, the change control board may simply consist of a project manager plus one or two engineers who are not directly involved in the software development. In some cases, there may only be a single change reviewer who gives advice on whether or not changes are justifiable.

When a set of changes has been approved, the software is handed over to the development of maintenance team for implementation. Once these have been completed, the revised software must be revalidated to check that these changes have been correctly implemented. The CM team, rather than the system developers, is responsible for building a new version or release of the software.

Change requests are themselves configuration items. They should be registered in the configuration database. It should be possible to use this database to discover the status of change requests and the change requests, which are associated with specific software components.

As software components are changed, a record of the changes made to each component should be maintained. This is sometimes called the derivation history of a component. One way to maintain such a record is in a standardized comment prologue kept at the beginning of the component. This should reference the change request associated with the software change.

The change management process is very procedural. Each person involved in the process is responsible for some activity. They complete this activity then pass on the forms and associated configuration items to someone else. The procedural nature of this process means that a change process model can be designed and integrated with a version management system. This model may then be interpreted so that the right documents are passed to the right people at the right time.

7.3 Version and Release Management

Version and release management are the processes of identifying and keeping track of different versions and releases of a system. Version managers must devise procedures to ensure that different versions of a system may be retrieved when required and are not accidentally changed. They may also work with customer liaison staff to plan when new releases of a system should be distributed.

A system version is an instance of a system that differs, in some way, from other instances. New versions of the system may have different functionality, performance or may repair system faults. Some versions may be functionally equivalent but designed for different hardware or software configurations. If there are only small differences between versions, one of these is sometimes called a variant of the other.

A system release is a version that is distributed to customers. Each system release should either include new functionality or should be intended for a different hardware platform. Normally, there are more versions of a system than releases. Some versions may never be released to customers. For example, versions may be created within an organization for internal development or for testing.

A release is not just an executable program or set of programs. It usually includes:

(1) Configuration files defining how the release should be configured for particular installations.

(2) Data files which are needed for successful system operation.

(3) An installation program which is used to help install the system on target hardware.

(4) Electronic and paper documentation describing the system.

All this information must be made available on some medium, which can be read by customers for that software. For large systems, this may be magnetic tape. For smaller systems, floppy disks may be used. Increasingly, however, releases are distributed on CD-ROM disks because of their large storage capacity.

When a system release is produced, it is important to record the versions of the operating system, libraries, compilers and other tools used to build the

software. If it has to be rebuilt at some later date, it may be necessary to reproduce the exact platform configuration. In some cases, copies of the platform software and tools may also be placed under version management.

Some automated tool almost always supports version management. This tool is responsible for managing the storage of each system version.

7.3.1 Version identification

Identifying versions of a system appears to be straightforward. The first version and release of a system is simply called 1.0, subsequent versions are 1.1, 1.2 and so on. At some stage, it is decided to create release 2.0 and the process starts again at version 2.1, 2.2 and so on. System releases normally correspond to the base versions, that is, 1.0, 2.0, 3.0 and so on. The scheme is a linear one based on the assumption that system versions are created in sequence. Version management tools such as SCCS support this approach to version identification.

In Figure 7.1 Version 1.0 has spawned two versions, 1.1 and 1.1a. Version 1.1 has also spawned two versions namely 1.2 and 1.1b. Version 2.0 is not derived from 1.2 but from 1.1a. Version 2.2 is not a direct descendant of version 2 as it is derived from version 1.2.

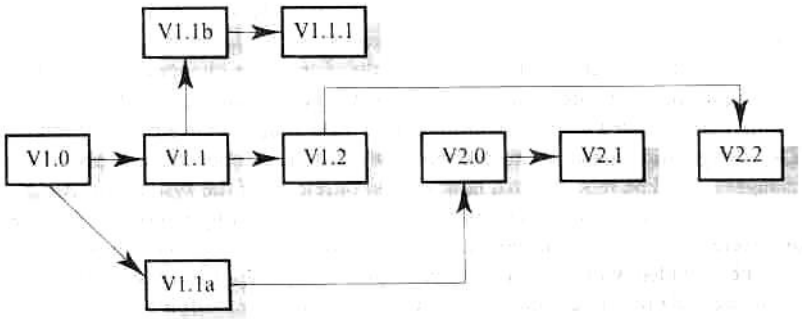


Fig. 7.1: Version Derivation Structure

An alternative to a numeric naming structure is to use a symbolic naming scheme. For example, rather than refer to Version 1.1.2, a particular instance of a system might be referred to as V1/VMS/DB server. This implies that this is a version of a database server for a Digital computer running the VMS operating system. This has some advantages over the linear scheme but, again, it does not truly represent the derivation structure.

7.3.2 Release management

New versions of a system may be created to fix reported faults or as part of the development process. In general, creating a new system version involves creating new source code and building the system. Creating a release, however, is more complex and expensive. As well as creating new source code and system building, data and configuration files may have to be prepared and new documentation written. The release must be packaged and distributed to customers.

Over the lifetime of a system, changes are likely to be proposed on a fairly regular basis. Corrective changes are intended to fix faults. Perfective changes are intended to implement new requirements or to improve system maintainability. Adaptive changes are intended to change the system to make it operate in a new environment. The configuration manager must decide how often the components affected by these changes should be rebuilt into a new version or release of the system.

When a new release of a system is created, the changes, which have been made, may introduce new faults or bring other existing faults to light. The more changes to a system, the more new faults will be introduced. Therefore, if a release incorporates a large number of changes, it is likely that there will be a correspondingly large number of new faults. These have to fix in the next system release.

Lehman's fifth law, the Law of Conservation of Familiarity, suggests that over the lifetime of a system, the incremental system change in each release is approximately constant. This 'law' was derived by analyzing systems over many years and measuring the number of system modules, which were modified in each release.

Lehman suggested that if a lot of new functionality was introduced in one release of a system, it would be necessary to issue another release fairly quickly. This would be required to correct errors that have resulted from the system changes and to improve the performance of the delivered release. Over the lifetime of a system, this was seen to be a self-regulating process. There was a limit to the rate at which new functionality could be introduced.

This suggests that it is unwise to change too much of a system's functionality at once. Otherwise an excessive number of faults may be

introduced. A good change strategy is to interleave fault repair releases and release which change the system's functionality as in figure 7.2 below.

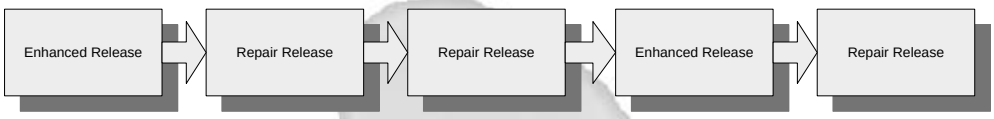


Fig. 7.2: System release strategy

Release management is complicated by the fact that customers may not actually want a new release of the system. Some system users may be happy with an existing system version. They may consider that it is not worth the cost of changing to a new release.

7.3.3 Version management tools

Version management involves managing large amounts of information and ensuring that system changes are recorded and controlled. There are several CASE tools available to support this process. For UNIX platforms, the most widely used version management systems are SCCS.

All version management systems provide basic set of capabilities although some have more sophisticated facilities than others. Examples of the capabilities, which may be included in a version management system, are:

(1) **Version and release identification** Managed versions may be assigned identifiers automatically when they are submitted to the system. Some systems support attributes value assignment for identification.

(2) **Controlled change** Versions of components must be checked out explicitly for change. The identifier of the engineer making the change is recorded. When re-submitted, a new version is created and tagged with the owner's identifier. The old version is never over-written.

(3) **Storage management** To reduce the storage space required by different versions which are largely the same, version management systems provide storage management facilities so that versions are described by their differences from some master version.

(4) **Change history recording** all of the changes made to a particular system or component may be recorded and listed.

The UNIX version management systems SCCS and RCS are fundamentally similar RCS offers more capabilities. RCS saves the source code of the most recent version of a system as the master version. This is created from an earlier master version. When a new master is created, the previous version is deleted and replaced by a specification of the differences between it and the new master version. This difference specification is called a delta. Rather than having to store all source code of all masters, RCS need only save a single master version and a set of deltas.

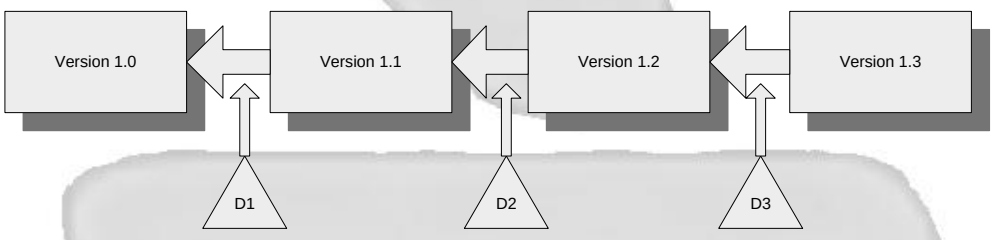


Fig. 7.3: Deltas in RCS

The main differences between RCS and SCCS are the method of storage management used and the version annotation capabilities. SCCS stores the first version of a system with further versions specified as deltas from it. It does not allow as much user-supplied information to be associated with a system version. RCS is generally more efficient for version creation. The versions that are requested most often are usually those, which created most recently.

Self Assessment Questions

1. _____ is the process, which controls the changes made to a system, and manages the different versions of the evolving software product.
2. The process of changing a system after it has been delivered and is in use is called _____.
3. _____ is concerned with taking existing legacy systems and re-implementing them to make it more maintainable.

7.4 Software Maintenance

The process of changing a system after it has been delivered and is in use is called **software maintenance**. The changes may involve simple changes to correct coding errors. Maintenance means evolution. It is the process of changing a system to maintain its ability to survive.

There are three different types of software maintenance:

(1) **Corrective maintenance** is concerned with fixing reported errors in the software. Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve the rewriting of several program components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.

(2) **Adaptive maintenance** means changing the software to some new environment such as a different hardware platform or for use with a different operating system. The software functionality does not radically change.

(3) **Perfective maintenance** involves implementing new functional or non-functional system requirements. Software customers as their organization or business changes generate these.

7.4.1 The maintenance process

The maintenance process is triggered by a set of change requests from system users, management or customers. The cost and impact of these changes are assessed. If the proposed changes are accepted, a new release of the system is planned. This release will usually involve elements of adaptive, corrective and perfective maintenance. The changes are implemented and validated and a new version of the system is released. The process then iterates with a new set of changes proposed for the new release. Figure 7.4 shows an overview of this process.

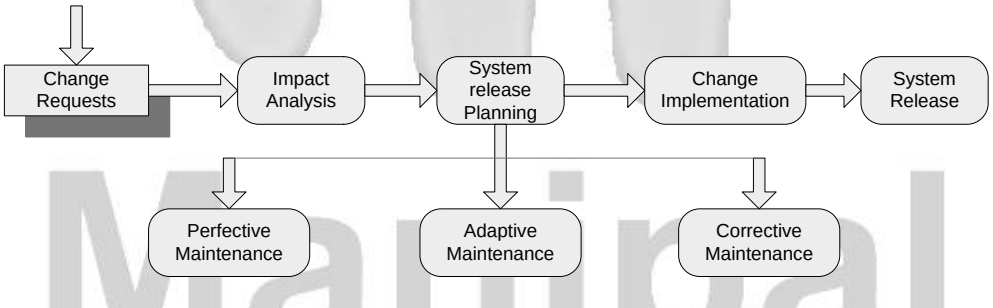


Fig. 7.4: An overview of the maintenance process.

Rather than viewing maintenance as a separate process, it should normally be considered as an iteration of the development process. New requirements must be formulated and validated, components of the system

must be redesigned and implemented and part or all of the system must be tested. This implies a process model as shown in Figure 7.5.

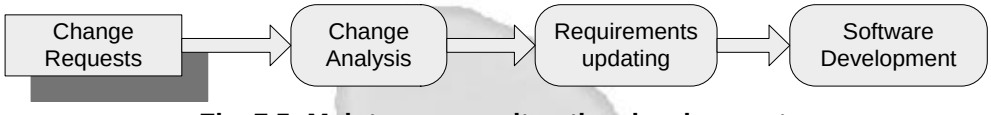


Fig. 7.5: Maintenance as iterative development.

7.4.2 System documentation

The system documentation includes all of the documents describing the implementation of the system from the requirement specification to the final acceptance test plan. Documents, which may be produced to aid the maintenance process, include

- The requirements document and as associated rationale.
- A document describing the overall system architecture.
- For each program in the system, a description of the architecture of that program.
- For each component, a specification and design description.
- Program source code listings which should be commented. If meaningful names are used and gotos are avoided, much of the code should be self-documenting with no need for explanatory comments. Program comments need only explain complex sections of code and provide a rationale for the coding method used.
- Validation documents describing how each program are validated and how the validation information relates to the requirements.
- A system maintenance guide that describes known problems with the system and that describes which parts of the system are hardware and software dependent. The guide should also explain how evolution of the system has been taken into account in its design.

  System documentation should be structured, with overviews leading the reader into more formal and detailed descriptions of each aspect of the system. It is important that documents are clear and readable otherwise they will not be used. While the standard of presentation need not match that of user manuals, it must be at such a level that poor grammar, spelling and document layout do not discourage readers.

7.4.3 Maintenance costs

Maintenance costs vary widely from one application domain to another. For other classes of system, such as embedded real-time systems, maintenance costs may be up to four times higher than development costs. The high reliability and performance requirements of these systems may require modules to be tightly linked and hence difficult to change.

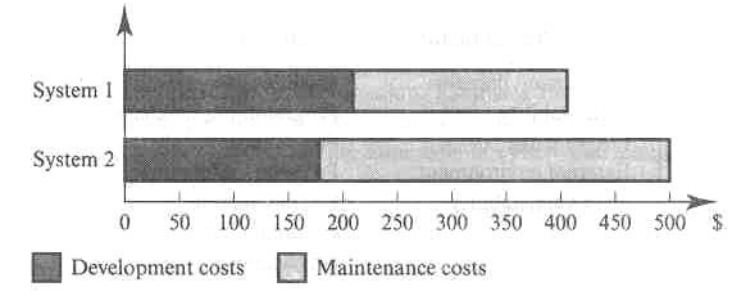


Fig. 7.6: Development and maintenance costs

Above Figure 7.6 shows how overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system. Because of the potential reduction in costs in understanding, analysis and testing, there is a significant multiplier effect when the system is developed for maintainability. For System 1, extra development costs of $25,000 are invested in making the system more maintainable. This results in a saving of $ 100,000 in maintenance costs. This assumes that a percentage increase in development costs results in a comparable percentage decrease in overall system costs.

Maintenance costs are related to a number of product, process and organizational factors. The principal technical and non-technical factors, which affect maintenance, are (Table 7.2):

1. **Module independence:** It should be possible to modify one component of a system without affecting other system components.
2. **Programming language:** Programs written in a high-level programming language are usually easier to understand (and hence maintain) than programs written in a low-level language.
3. **Programming style:** The way in which a program is written contributes to its understandability and hence the ease with which it can be modified.

4. **Program validation and testing:** Generally, the more time and effort spent on design validation and program testing, the fewer errors in the program. Consequently, corrective maintenance costs are minimized.
5. **The quality of program documentation:** If a program is supported by clear, complete yet concise documentation, the task of understanding the program can be relatively straightforward. Program maintenance costs tend to be less for well-documented systems than for systems supplied with poor or incomplete documentation.
6. **The configuration management techniques used:** One of the most significant costs of maintenance is keeping track of all system documents and ensuring that these are kept consistent. Effective configuration management can help control this cost.
7. **The application domain:** If the application domain is clearly defined and well understood, the system requirements are likely to be complete. Relatively little perfective maintenance may be necessary. If the application is in a new domain, it is likely that the initial requirements will be modified frequently, as users gain a better understanding of their real needs.
8. **Staff stability:** Maintenance costs are reduced if system developers are responsible for maintaining their own programs. There is no need for other engineers to spend time understanding the system. In practice, however, it is very unusual for developers to maintain a program throughout its useful life.
9. **The age of the program:** As a program is maintained, its structure is degraded. The older the program, the more maintenance it receives and the more expensive this maintenance becomes.
10. **The dependence of the program on its external environment:** If a program is dependent on its external environment it must be modified as the environment changes. For example, changes in a taxation system might require payroll, accounting, and stock control programs to be modified.
11. **Hardware stability:** If a program is designed for a particular hardware configuration that does not change during the program's lifetime, no maintenance due to hardware changes will be required. However, this situation is rare. Programs must often be modified to use new hardware which replaces obsolete equipment.

12.

Table 7.2: Factors involved in Maintenance Costs

Non-technical factors	Technical factors
Application domain	Module independence
Staff stability	Programming language
Program age	Programming style
External environment	Program validation
Hardware stability	Documentation
	Configuration
	Management

7.4.4 Maintainability measurement

Maintainability metric can help management to make an informed decision on whether a component should be maintained or completely rewritten to reduce future maintenance costs.

Maintainability metrics do not measure the cost of making a particular change to a system nor do they predict whether or not a particular component will have to be maintained. Rather, they are based on the assumption that the maintainability of a program is related to its complexity. The metrics measures some aspects of the program complexity. It is suggested that high complexity values correlate with difficulties in maintaining a system component.

Examples of process metrics, which may be useful for assessing maintainability, are:

1. **Number of requests for corrective maintenance.** If the number of failure reports is increasing, this may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. This may indicate a decline in maintainability.
2. **Average time required for impact analysis.** This reflects the number of program components that are affected by the change request. If this time increases, it implies more and more components are affected and maintainability is decreasing.
3. **Average time taken to implement a change request.** This is not the same as the time for impact analysis although it may correlate with it.

The activities involved are making changes to the system and its documentation rather than simply assessing what components are affected. This change time depends on the difficulty of programming so those non-functional requirements such as performance are met. If the time to implement a change increases, this may indicate a decline in maintainability.

4. **Number of outstanding change requests.** If this number increase with time, it may imply a decline in maintainability.

7.5 Software Reengineering

An application has served the business needs of a company for 10 or 15 years. During that time it has been corrected, adapted and enhanced many times. If the application is unstable because but every time a change is attempted, unexpected and serious side effects will occur. Un-maintainable software is not a new problem. In fact, a software maintenance team has generated the broadening emphasis on software reengineering.

Software Maintenance

The maintenance is described by four activities that are undertaken after a program is released for use.

- Corrective maintenance
- Adaptive maintenance
- Perfective or enhancement maintenance
- Preventive maintenance or re-engineering

Only about 20 percent of all maintenance work is spent “fixing mistakes “. The remaining 80 percent is spent adapting existing systems to change in their external environment, making enhancements requested by users, and reengineering an application for future use.

Software Reengineering Process Model

Reengineering takes time; it costs significant amounts of money; and it absorbs resources that might be otherwise occupied on immediate concerns. For all of this reason, reengineering is not accomplished in a few months or even a few years. Reengineering of information systems is an activity that will absorb information technology resources for many years. That’s why every organization needs a practical strategy for software reengineering.

Reengineering is a rebuilding activity, and we can better understand the reengineering of information systems if we consider an analogous activity; the rebuilding of a house. Consider the following situation; if you purchased a house in another state. You've never actually seen the property, but you acquired it at a reasonable low price, with the warning that it might have to be completely rebuilt. How would you proceed?

- Before you can start rebuilding, it would seem reasonable to inspect the house. To determine whether it is in need of rebuilding, you would create a list of criteria so that your inspection would be systematic.
- Before you tear down and rebuilt the entire house, be sure that the structure is weak. If the house is structurally sound, it may be possible to "remodel" without rebuilding.
- Before you start rebuilding, be sure you understand how the original was built. Take a peek behind the walls. Understand the wiring, the plumbing, and the structural internals. Even if you trash them all, the insight you'll gain will serve you well when you start construction.
- If you begin to rebuild, use only the most modern, long-lasting materials. This may cost a bit more now, but it will help you to avoid expensive and time-consuming maintenance later.
- If you decide to rebuild, be disciplined about it. Use practices that will result in high quality-today and in the future.

These principles focus on the rebuilding of a house, they apply equally well to the reengineering of computer-based systems and applications.

To implement these principles, we apply software reengineering process model that defines size activities, shown in figure. In some cases, these activities occur in a linear sequence, but this is not always the case. For e.g., it may be that reverse engineering may have to occur before document restricting can commence.

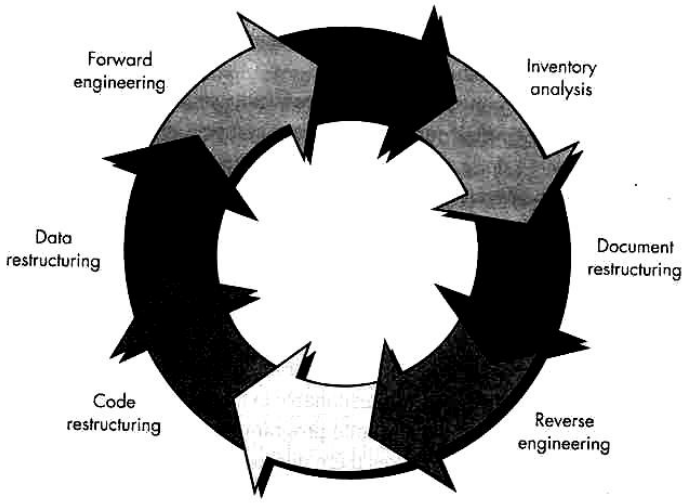


Fig. 7.7: A software reengineering process model

The reengineering paradigm show in figure 7.7 is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revised. For any particular cycle, the process can terminate after any one of these activities.

Inventory analysis: Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application by storing the information according to business critically, longevity, current maintainability, and other locally important criteria, candidate for re-engineering work.

It is important to note that the inventory should be revisited on a regular cycle. The status of applications (e.g. Business critically) can change as a function of time, and as a result, priorities for reengineering will shift.

Document restructuring Weak documentation is the trademark of many legacy systems. But what do we do about it? What are our options?

1. Creating documentation is more time consuming. If the system works, we'll live with what we have. In some cases, this is correct approach. It is not possible to re-create documentation for hundreds of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change.

2. Documentation must be updated, but we have limited resources. We'll use a "document when touched" approach. It may not be necessary to fully re-document an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.
3. The system is business critical and must be fully re documented. Even in this case, an intelligent approach is to pare documentation to an essential minimum.

A software of organization must choose the one that is most appropriate for each case.

Reverse Engineering: The term reverse engineering has it origins in the hardware world. A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing secrets. These secrets could be easily understood if the competitor's design and manufacturing specifications were obtained. But these documents are proprietary and unavailable to the company doing the reverse engineering in essence, successfully derives one or more design and manufacturing specifications for a product by examining actually specimens of the product.

Reverse engineering for software is quite similar in most cases, however, the program to be reverse engineered is not a competitors. Rather, it is the company's own work. The 'secrets' to be understood are obscure because no specification was ever developed. Therefore, reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of obstruction than source code. Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.

Code restructuring: The common type of reengineering is code restructuring. Some legacy systems have relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. In such cases, the code within the suspect modules can be restructured.

To accomplish this activity, the source code is analyzed using are structuring tool, Violations of structured programming constructs are noted

and code is then restructured. The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced, internal code documentation is updated.

Data structuring: A program with weak data architecture will be difficult to adapt and enhance. In fact, for many applications, data architecture has more to do with the long-term viability of a program that the source code itself.

Unlike code restructuring, which occurs at a relatively low level of abstraction, data structuring is a full-scale reengineering activity. In most cases, data restructuring begins with are reverse engineering activity. Current data architecture is dissected and necessary data models are defined. Data object and attributes are identified, and existing data structures are reviewed for quality.

When data structure is weak (e.g. Flat files are currently implemented, when a relational approach would greatly simplify process), the data reengineered. Because data architecture has a strong influence on program architecture and algorithms that populate it, changes to the data will invariably result in either architectural or code-level changes.

Forward Engineering: In an ideal world, applications should be rebuilt using an automated ‘reengineering engine’. The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality. In the short term, it is unlikely that such an ‘engine’ will appear, but CASE vendors have introduced told shat provide a limited subset of these capabilities that addresses specific application domain. More important these reengineering tools are becoming increasingly more sophisticated.

Forward engineering, also called renovation or reclamation, not only recovers design information from existing software, but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software re-implements the function of the existing system and also adds new functions and/or improves overall performance.

7.6 Software Refactoring

Software Refactoring is the process of factoring the design module, each modules factored into the components which is more readable and easier to understand. The related components of a program may be dispersed through the code. The program modularization is usually carried out manually by analyzing the software. To re-factorize the program, the components must be identified and function of the components deduced. Software refactoring is the program refining the factored modules, their specifications and abstract data in to manageable parts.

Refactoring: Improving the Design of Existing Code

As the application of object technology – particularly the Java programming language – has become commonplace, a new problem has emerged to confront the software development community. Significant numbers of poorly designed programs have been created by less-experienced developers, resulting in applications that are inefficient and hard to maintain and extend. Increasingly, software system professionals are discovering just how difficult it is to work with these inherited, "non-optimal" applications. For several years, expert-level object programmers have employed a growing collection of techniques to improve the structural integrity and performance of such existing software programs. Referred to as "Refactoring," these practices have remained in the domain of experts because no attempt has been made to transcribe the lore into a form that all developers could use until now. In Refactoring: Improving the Design of Existing Software, renowned object technology mentor Martin Fowler breaks new ground, demystifying these master practices and demonstrating how software practitioners can realize the significant benefits of this new process.

With proper training a skilled system designer can take a bad design and rework it into well-designed, robust code. In this book, Martin Fowler shows you where opportunities for refactoring typically can be found, and how to go about reworking a bad design into a good one. Each refactoring step is simple – seemingly too simple to be worth doing. Refactoring may involve moving a field from one class to another, or pulling some code out of a method to turn it into its own method, or even pushing some code up or down a hierarchy. While these individual steps may seem elementary, the cumulative effect of such small changes can radically improve the design. Refactoring is a proven way to prevent software decay.

Refactoring Improves the Design of Software

Without refactoring, the design of the program will decay. As people change code – changes to realize short-term goals or changes made without a full comprehension of the design of the code – the code loses its structure. It becomes harder to see the design by reading the code. Refactoring is rather like tidying up the code. Work is done to remove bits that aren't really in the right place. Loss of the structure of code has a cumulative effect. The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. Regular refactoring helps code retain its shape.

Poorly designed code usually takes more code to do the same things, often because the code quite literally does the same thing in several places. Thus an important aspect of improving design is to eliminate duplicate code. The importance of this leads to future modifications to the code. Reducing the amount of code won't make the system run any faster, because the effect on the footprint of the programs rarely is significant. Reducing the amount of code does, however, make a big difference in modification of the code. The more code there is, the harder it is to modify correctly. There's more code to understand. You change this bit of code here, but the system doesn't do what you expect because you didn't change that bit over there that does much the same thing in a slightly different context. By eliminating the duplicates, you ensure that the code says everything once and only once, which is the essence of good design.

Refactoring and Performance

A common concern with refactoring is the effect it has on the performance of a program. To make the software easier to understand, you often make changes that will cause the program to run more slowly. This is an important issue. Software has been rejected for being too slow, and faster machines merely move the goalposts. Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. The secret to fast software, in all but hard real-time contexts, is to write tunable software first and then to tune it for sufficient speed.

There are three general approaches in writing fast software. The most serious of these is time budgeting, used often in hard real-time systems. In this situation, as you decompose the design you give each component a budget for resources – time and footprint. That component must not exceed

its budget, although a mechanism for exchanging budgeted times is allowed. Such a mechanism focuses hard attention on hard performance times. It is essential for systems such as heart pacemakers, in which late data is always bad data. This technique is overkill for other kinds of systems, such as the corporate information systems with which we usually work.

Self Assessment Questions

4. The _____ includes all of the documents describing the implementation of the system from the requirement specification to the final acceptance test plan.
5. _____ can help management to make an informed decision on whether a component should be maintained or completely rewritten to reduce future maintenance costs.
6. _____, also called renovation or reclamation, not only recovers design information from existing software, but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.

7.7 Summary

- Configuration management is the management of system change. When a system is maintained the role of the CM team is to ensure that the changes are incorporated in a controlled way.
- In a large project, a formal document-naming scheme should be established and used as a basic for managing the project documents.
- The CM team should be supported by a configuration database that records information about system changes and change request which are outstanding projects should have some formal means of requesting system changes.
- When setting up a configuration management scheme, a consistent scheme of version identification should be established. Version attributes such as the customer name and target platform may be used to identify particular versions.
- System releases should be phased. A release which provides major new system functionality should be followed by a release which is mostly concerned with fault removal and performance enhancement.

- There are three principle types of software maintenance. These are perfective maintenance where new functionality is added to the system; adaptive maintenance where the system is adapted to new environments and corrective maintenance, which is system repair.
- The cost of software maintenance usually exceeds the cost of software development. Typically, maintenance costs are at least 505 of lifetime system costs for business systems and even more for embedded systems.
- System documentation for maintenance should include a system requirements document, design documents and validation documents. It must be kept up to date when the system is changed.
- Several technical and non-technical factors affect maintenance costs. These include application factors, environmental factors, personnel factors, programming language factors and documentation.
- Software re-engineering tools and techniques will become significant part of the maintenance process. Re-engineering takes the information obtained and restructures the program to achieve higher quality and therefore better maintainability for the feature

7.8 Terminal Questions

1. What is Change Management? Explain.
2. Give the importance of version and release management.
3. What is software maintenance? Explain its significance.

7.9 Answers

Self Assessment Questions

1. Configuration management
2. Software maintenance
3. Software re-engineering
4. System documentation
5. Maintainability metric
6. Forward engineering

Terminal Questions

1. The change management process should come into effects when the software or associated documentation is put under the control of the configuration management team. Change management procedures should be designed to ensure that the costs and benefits of change are properly analyzed and that changes to a system are made in a controlled way. (Refer section 7.2)
2. Version and release management are the processes of identifying and keeping track of different versions and releases of a system. Version managers must devise procedures to ensure that different versions of a system may be retrieved when required and are not accidentally changed. They may also work with customer liaison staff to plan when new releases of a system should be distributed. (Refer section 7.3)
3. The process of changing a system after it has been delivered and is in use is called software maintenance. The changes may involve simple changes to correct coding errors. (Refer section 7.4)

Unit 8 Software Testing Techniques

Structure:

8.1 Introduction

The importance of software testing and its implications with respect to software quality cannot be overemphasized.

The development of software systems involves a series of Production activities where opportunities for injection of human fallibilities are enormous. Errors may begin to occur at the very inception of the process where the objectives ... may be erroneously or imperfectly specified, as well as line later design and development stages ... Because of human inability to perform and communicate with perfection, software development is accompanied by a quality assurance activity.

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation. The increasing visibility of software as a system element and the attendant "costs' associated with a software failure are motivating forces for well-planned, thorough testing. It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing. In the extreme, testing of human-rated software (e.g., flight control, nuclear reactor monitoring) can cost three to five times as much as all other software Engineering steps combined.

In this chapter, we discuss software testing fundamentals and techniques for software test case design. Software testing fundamentals define the overriding objectives for software testing. Test case design focuses on a set of techniques for the creation of test cases that meet overall testing objectives.

Objectives:

After studying this unit, you should be able to:

- explain the testing fundamentals
- describe the testing principles
- discuss testing for specialized environments, architectures and applications

8.2 Software Testing Fundamentals

Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. The engineer creates a series of test cases that are intended to 'demolish' the software that has been built. In fact, Testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.

Software engineers are by their nature constructive people. Testing requires the developer discard preconceived notions of the 'correctness' of software just developed and overcome a conflict of interest that occurs when errors are uncovered. Beizer describes this situation effectively when he states:

There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top- down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For failing to achieve inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For failing to be telepathic?

For not solving human communications problems that have been kicked around for forty centuries?

Should testing instill guilt? Is testing really destructive? The answer to these questions is "No!" However, the objectives of testing are somewhat different than we might expect.

Testing Objectives

In an excellent book on software testing, Glen Myers states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

These objectives imply a dramatic change in viewpoint. They move counter to the commonly held view that a successful test is one in which no errors are found. Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met. In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects, it can only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

8.3 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggests a set of testing principles that have been adapted for use in this book:

- **All tests should be traceable to customer requirements:** As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- **Tests should be planned long before testing begins:** Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
- **The Pareto principle applies to software testing:** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The probe" of course, is to isolate these suspect components and to thoroughly test them.
- **Testing should begin "in the small" and progress toward testing "in the large":** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors integrated clusters of components and ultimately in the entire system.
- **Exhaustive testing is not possible:** The number of path permutations even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
- **To be most effective, testing should be conducted by an independent third party:** By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing). The software engineer who created the system is not the best person to conduct all tests for the software.

Testability

In ideal circumstances, a software engineer designs a computer program, a system, or a product with "testability" in mind. This enables the individuals charged with testing to design effective test cases more easily. But what is testability? James Bach describes testability in the following manner.

Software testability is simply how easily [a computer program] can be tested. Since testing is so profoundly difficult, it pays to know what can be done to streamline it. Some times programmers are willing to do things that will help the testing process and a check-list of possible design points, features, etc., can be useful in negotiating with them.

There are certainly metrics that could be used to measure testability in most of its aspects. Sometimes, testability is used to mean how adequately a particular set of the tests will cover the product. It's also used by the military to mean how easily a tool can be checked and repaired in the field. Those two meanings are not the same as *software testability.* The checklist that follows provides a set of characteristics that lead to testable software.

Operability

"The better it works, the better it can be tested."

- The system has few bugs (bugs add analysis and reporting overhead to the test process).
- No bugs block the execution of tests.
- The product evolves in functional stages (allows simultaneous development and testing).

Observability

"What you see is what you test."

- Distinct output is generated for each input.
- The system states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected through self-testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

Controllability

"The more we control the software, the more the testing can be automated and optimized.'

- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.

Decomposability

"By controlling the scope of testing, we can more quickly isolate problems and perform smarter re-testing.'

- The software system is built from independent modules.
- Software modules can be tested independently.

Simplicity

"The less there is to test, the more quickly we can test it, Functional simplicity (e.g., the feature set is the minimum necessary to meet Requirements).

- Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).
- Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability

"The fewer the changes, the fewer the disruptions to testing."

- Changes to the software are infrequent.
- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.
- The software recovers well from failures.

Understandability

"The more information we have, the more effectively we test.

- The design is well understood.
- Dependencies between internal, external, and shared components are well understood.
- Changes to the design are communicated.
- Technical documentation is instantly accessible.
- Technical documentation is well organized.

- Technical documentation is specific and detailed.
- Technical documentation is accurate.

A software engineer to develop a software configuration (i.e., programs, data, and documents) that is amenable to testing can use the attributes suggested by Bach.

And what about the tests themselves? Kaner, Falk, and Nguyen suggest the following attributes of a "good" test:

1. A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.
2. A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different). For example, a module of the *SafeHome* software is designed to recognize a user password to activate and deactivate the system. In an effort to uncover an error in password input, the tester designs a series of tests that input a sequence of passwords. Valid and invalid passwords (four numeral sequences) are input as separate tests. However, each valid/invalid password should probe a different mode of failure. For example, the invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, say 1235, would have the same purpose as 1234 and is therefore redundant. However, the invalid input 8081 or 8180 has a subtle difference, attempting to demonstrate that an error exists for passwords "close to" but not identical with the valid password.
3. A good test should be "best of breed". In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

4. A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

Self Assessment Questions

1. Testing is a process of executing a program with the intent of finding a _____.
2. The Pareto principle implies that _____ percent of all errors uncovered during testing will likely be traceable to _____ percent of all program components.
3. A good test should be neither too _____ nor too _____.

8.4 White-Box Testing

White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

A reasonable question might be posed at this juncture: "Why to spend time and energy worrying about (and testing) logical minutiae when we might better expend effort ensuring that program requirements have been met?' Stated another way, why don't we spend all of our energy on black-box tests? The answer lies in the nature of software defects:

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement function, conditions, or controls that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while 'special case' processing tends to fall into the cracks.
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. The logical flow of a program is sometimes counterintuitive, meaning that our unconscious

assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.

- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur.
- Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.

Each of these reasons provides an argument for conducting white-box tests. Black-box testing, no matter how thorough, may miss the kinds of errors noted here. White- box testing is far more likely to uncover them.

8.5 Control Structure Testing

The basis path testing technique is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

1. Condition Testing
2. Data Flow Testing
3. Loop Testing

8.6 Black-Box Testing

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black- box testing tends to be applied during later stages of testing. Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing and (2) test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Graph-Based Testing Methods

The first step in black-box testing is to understand the objeCtS6 that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another." Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

To accomplish these steps, the software engineer begins by creating a graph-a collection of nodes that represent objects; links that represent the relationships between objects; node weights that describe the properties of a node (e.g., a specific data value or state behavior); and link weights that describe some characteristic of a link.

Equivalence partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be

derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence for an input condition. Using concepts introduced in the preceding section if a set of objects can be linked by relationships that are symmetric, transitive and reflexive, an equivalence class is present. An equivalence class represents of valid or invalid states for input conditions. Typically, an input condition is specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence classes are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

As an example, consider data maintained as part of an automated banking application. The user can access the bank using a personal computer, provide a six-digit password and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking applications data in the form

Area code – blank or three-digit number

Prefix – three-digit number not beginning with 0 or 1

Suffix – four-digit number

Password – six digit alphanumeric string

Commands – check, deposit, bill pay, and the like

The input conditions associated with each data element for the banking applications can be specified as

area code: Input condition, Boolean-the area code may or may not be present.

Input condition, range-values defined between 200 and 999, with specific exceptions.

prefix: Input condition, range-specified value >200

Input condition, value-four-digit length

password: Input condition, Boolean-a password may or may not be present.

Input condition, value-six-character string.

command: Input condition, set-containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each main data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class is exercised at once.

Self Assessment Questions

4. _____, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases.
5. The _____ technique is one of a number of techniques for control structure testing.
6. Black-box testing, also called as _____, focuses on the functional requirements of the software.

8.7 Boundary Value Analysis

For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of

the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range, bounded by values *a* and *b*, test cases should be designed with values a and b and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

Comparison Testing

There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical. In such applications redundant hardware and software are often used to minimize the possibility of error. When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification. In such situations, each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

Using lessons learned from redundant systems, researchers have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered

computer-based system. These independent versions form the basis of a black-box testing technique called comparison testing or back-to-back testing.

When multiple implementations of the same specification have been produced, test cases designed using other black-box techniques (e.g., equivalence partitioning) are provided as input to each version of the software. If the output from each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference. In most cases, the comparison of outputs can be performed by an automated tool.

Comparison testing is not foolproof. if the specification from which all versions have been developed is in error, all versions will likely reflect the error. in addition, if each of the independent versions produces identical but incorrect results, condition testing will fail to detect the error.

Testing for specialized environments, architectures and applications

As computer software has become more complex, the need for specialized testing approaches has also grown. The white-box and black-box testing methods discussed are applicable across all environments, architectures, and applications, but unique guidelines and approaches to testing are sometimes warranted. In this section we consider testing guidelines for specialized environments, architectures, and applications that are commonly encountered by software engineers.

8.8 Testing GUIs

Graphical user interfaces (GUIs) present interesting challenges for software engineers. Because of reusable components provided as part of GUI development environments, the creation of the user interface has become less time consuming and more precise. But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI.

Due to the large number of permutations associated with GUI operations, testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years.

Testing of Client/Server Architectures

Client/server (C/S) architectures represent a significant challenge for software testers. The distributed nature of client/server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of C/S architectures and the software that reside within them considerably more difficult than stand-alone applications. In fact, recent industry studies indicate a significant increase in testing time and cost when C/S environments are developed.

8.9 Testing Documentation and Help Facilities

The term software testing conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. From the definition of software, it is important to note that testing must also extend to the third element of the software configuration-documentation.

Errors in documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more frustrating than following a user guide or an on-line help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. it is for this reason that that documentation testing should be a meaningful part of every software test plan.

Documentation testing can be approached in two phases. The first phase, review and inspection, examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the use of the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used

to define various classes of input and associated interactions. Program usage is then tracked through the documentation. The following questions should be answered during both phases:

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Is the document table of contents and index accurate and complete?
- Is the design of the document (layout, typefaces, indentation, and graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted and areas of document ambiguity or weakness are defined for potential rewrite.

Testing for Real-time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix-time. Not only does the test case designer have to consider white and black-box test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. in many situations, test data provided when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.

For example, the real-time software that controls a new photocopier accepts operator interrupts (i.e., the machine operator hits control keys such as RESET or DARKEN) with no error when the machine is making copies (in the "copying" state). These same operator interrupts, if input when the machine is in the "jammed" state, cause a display of the diagnostic code indicating the location of the jam to be lost (an error).

In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.

Comprehensive test case design methods for real-time systems have yet to evolve. However, an overall four-step strategy can be proposed:

Task testing:
The first step in the testing of real-time software is to test each task independently. That is, white-box and black-box tests are designed and executed for each task. Each task is executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior.

Behavioral testing:
Using system models created with CASE tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built. Using a technique that is similar to equivalence partitioning, events (e.g., interrupts, control signals) are categorized for testing. For example, events for the photocopier might be user interrupts (e.g., reset counter), mechanical interrupts (e.g., paper jammed), system interrupts (e.g., toner low), and failure modes (e.g., roller overheated) Each of these events is tested individually and the behavior of the executable system is examined to detect errors that occur as a consequence of processing associated with these events. The behavior of the system model (developed during the analysis activity) and the executable software can be compared for conformance. Once each class of events has been tested, events are presented to the system in random order and with random frequency. The behavior of the software is examined to detect behavior errors.

Intertask testing

Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.

System testing

Software and hardware are integrated and a full range of system tests are conducted in an attempt to uncover errors at the software/hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential. Using the state transition diagram and the control specification, the tester develops a list of all possible interrupts and the processing that occurs as a consequence of the interrupts. Tests are then designed to assess the following system characteristics:

- Are interrupt priorities properly assigned and properly handled?
- Is processing for each interrupt handled correctly?
- Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?
- Does a high volume of interrupts arriving at critical times create problems in function or performance?

In addition, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

Self Assessment Questions

7. _____ leads to a selection of test cases that exercise bounding values.
8. GUI stands for _____.
9. The first step in the testing of real-time software is to test each _____ independently.

8.10 Summary

- The primary objective for test case design is to derive a set of tests that have the highest likelihood for uncovering errors in the software. To accomplish this objective, two different categories of test case design techniques are used: white-box testing and black-box testing.

- White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure coverage.
- Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.
- Hetzel describes white-box testing as 'testing in the small.' His implication is that the white-box tests that we have considered in this chapter are typically applied to small program components (e.g., modules or small groups of modules). Black-box testing, on the other hand, broadens our focus and might be called 'testing in the large.'
- Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough tests coverage.
- Equivalence partitioning divides the input domain into classes of data that are likely to exercise specific software function. Boundary value analysis probes the program’s ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems will small numbers of input parameters.
- Specialized testing methods encompass a broad array of software capabilities and application areas. Testing for graphical user interfaces, client/server architectures, documentation and help facilities, and real-time systems each require specialized guidelines and techniques.
- Experienced software developers often say, 'Testing never ends, it just gets transferred from you [the software engineer] to Your Customer. Every time your customer uses the program, a test is being conducted.' By applying test case design, the software engineer can achieve more complete testing and thereby uncover and correct the highest number of errors before the 'customer's tests begin.

8.11 Terminal Questions

1. Give brief account of Software Testing Fundamentals.
2. Explain various principles involved in Software Testing.
3. What is Boundary Value Analysis (BVA)? Explain.

8.12 Answers

Self Assessment Questions

1. Error
2. 80, 20
3. Simple, Complex
4. White-box testing
5. Basis path testing
6. Behavioral testing
7. Boundary Value Analysis (BVA)
8. Graphical User Interface
9. Task

Terminal Questions

1. Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. (Refer section 8.2)
2. Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggests a set of testing principles that have been adapted for use. (Refer section 8.3)
3. For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values. (Refer section 8.7)

Unit 9 Software Testing Assurance

Structure:

9.1 Introduction

This chapter introduces test Verification and Validation (V & V) with their specific definitions. Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing. Testing plays an extremely important role in V & V.

The Test plan section describes the overall strategy for integration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. There are five different types of test strategies like Top-down testing, Bottom-up Testing, Thread testing, Stress testing, Back-to-back testing which are explained in detail.

A software engineer must understand the basic principles that guide software testing. A detailed study of testing method and tools are also

discussed here through, reviews, requirements, designs, programs, and software changes. An additional requirements in testing OO system, is also discussed in this chapter. Different types (system, acceptance and regression) are also detailed out.

Metrics Collection, Computation, and Evaluation, is also an integral part of this chapter.

Objectives:

After studying this unit, you should be able to:

- explain the basic principles of Verification and Validation
- follow the test plan documentation
- document and review the various test plans and strategies

9.2 Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this in another way:

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

The definition of V&V encompasses many of the activities that we have referred to as software quality assurance (SQA).

Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective formal technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

Miller relates software testing to quality assurance by stating that

"the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems."

9.2.1 Validation Testing

At the culmination of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests – validation testing – may begin.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that *validation succeeds when the software functions in a manner that can be reasonably expected by the customer*. At this point a battle-hardened software developer might protest: Who or what is the arbiter of reasonable expectations?

Reasonable expectations are defined in the Software Requirements Specification – a document that describes all user-visible attributes of the software. The Specification contains a section called Validation criteria.

9.2.2 Validation Test Criteria

Software validation is achieved through a series of black box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and the procedure are designed to ensure that all functional requirements are satisfied, all performance requirements are achieved, documentation is correct and human-engineered, and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of the two possible conditions exists:

1. The function or performance characteristics conform to specification and are accepted, or
2. A deviation from specification is uncovered and a deficiency list is created.

Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled completion. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

9.3 Test Plan

9.3.1 Test Documentation

Table 9.1: Test Specification Outline

Test Specification outline
I. Scope of testing
II. Test plan
A. Test phases and builds
B. Schedule
C. Overhead software
D. Environment and resources
III. Test procedure n. (description of test for build n)
A. Order of integration
1. Purpose
2. Modules to be tested
B. Unit tests for modules in build
1. Description of tests for module m
2. Overhead software description
3. Expected results
C. Test environment
1. Special tools or techniques
2. Overhead software description
D. Test case data
E. Expected results for build n
IV. Actual test results
V. References
VI. Appendices

An overall plan for integration of the software and a description of specific tests are documented in a Test Specification. The specification is deliverable in the software engineering process and becomes part of the software configuration. Table 9.1 presents a *Test Specification* outline that may be used as a framework for this document.

Scope of testing summarizes the specific functional, performance, and internal design characteristics that are to be tested. Testing effort is bounded, criteria for completion of each test phase are described, and schedule constraints are documented.

The Test plan section describes the overall strategy for integration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for a computer graphics-oriented CAD system might be divided into the following test phases:

- *User interaction* (command selection; drawing creation; display representation; error processing and representation)
- *Data manipulation and analysis* (symbol creation; dimensioning; rotation; computation of physical properties)
- *Display processing and generation* (two-dimensional displays; three-dimensional displays; graphs and charts)
- *Database management* (access; update; integrity; performance)

Each of these phases and sub-phases (denoted in parentheses) delineates a broad functional category within the software and can generally be related to a specific domain of the program structure. Therefore, program builds (groups of modules) are created to correspond to each phase.

The following criteria and corresponding tests are applied for all test phases: *Interface integrity.* Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

Functional validity. Tests designed to uncover functional errors are conducted.

Information content. Tests designed to uncover errors associated with local or global data structures are conducted.

Performance. Tests designed to verify performance bounds established during software design are conducted.

These criteria and the tests associated with them are discussed in this section of the Test Specification.

A schedule for integration, overhead software, and related topics are also discussed as part of the *Test plan* section. Start and end dates for each phase are established and "availability windows" for unit-tested modules are defined. A brief description of overhead software (stubs-and drivers) concentrates on characteristics that might require special effort. Finally, the test environment and resources are described.

A detailed testing procedure that is required to accomplish the *test plan* is described in the Test procedure section. Referring back to Test Specification outline item, the order of integration and corresponding tests at each integration step are described. A listing of all test cases (annotated for subsequent reference) and expected results is also included.

A history of actual test results, problems, or peculiarities is recorded in the fourth section of the *Test Specification*. Information contained in this section can be vital during software maintenance. Appropriate references and appendices are presented in the final two sections.

Like all other elements of a software configuration, the *Test Specification* format may be tailored to the local needs of a software development organization. It is important to note, however, that an integration strategy, contained in a Test plan, and testing details, described in a Test procedure, are essential ingredients and must appear.

Self Assessment Questions

1. _____ refers to the set of activities that ensure that software correctly implements a specific function.
2. _____ refers to a set of activities that ensure that the software that has been built is traceable to customer requirements.
3. Software validation is achieved through a series of _____ that demonstrate conformity with requirements.

9.4 Test Strategies

9.4.1 Top-Down Testing

Top-down testing (figure 9.1) tests the high levels of a system before testing its detailed components. The program is represented as a single abstract component with sub components represented by stubs. Stubs have the same interface as the component but very limited functionality. After the top-level component has been tested, its stub components are implemented and tested in the same way. This process continues recursively until the bottom level components are implemented. The whole system may then be completely tested.

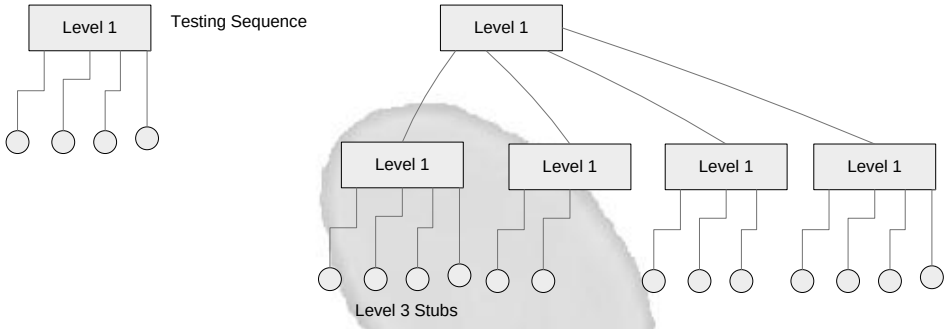


Fig. 9.1: Top-Down Testing

Top-down testing should be used with top-down program development so that a system component is tested as soon as it is coded. Coding and testing are a single activity with no separate component or module-testing phase.

If top-down testing is used unnoticed design errors might be detected at an early stage in the testing process. As these errors are usually structural errors, early detection means that they can be corrected without undue costs. Early error detection means that extensive redesign and re-implementation may be avoided. Top-down testing has the further advantage that a limited, working system is available at an early stage in the development. This is an important psychological boost to those involved in the system development. It demonstrates the feasibility of the system to management. Validation, as distinct from verification, can begin early in the testing process as a demonstrable system can be made available to users.

Strict top-down testing is difficult to implement because of the requirement that program stubs, simulating lower levels of the system, must be produced.

The main disadvantage of top-down testing is that test out put may be difficult to observe. In many systems, the higher levels of that system do not generate output but, to test these levels, they must be forced to do so. The tester must create an artificial environment to generate the test results.

9.4.2 Bottom-Up Testing

Bottom – up testing is the converse of top down testing. It involves testing the modules at the lower levels in the hierarchy, and then working up the

hierarchy of modules until the final module is tested. The advantage of bottom-up testing is the disadvantages of the top-down testing and vice versa.

When using bottom-up testing (figure 9.2), test drivers must be written to exercise the lower-level components. These test drivers simulate component's environment and are valuable components in their own right. If the components being tested are reusable components, the test-drivers and test data should be distributed with the component. Potential re-users can run these tests to satisfy themselves that the component behaves as expected in their environment.

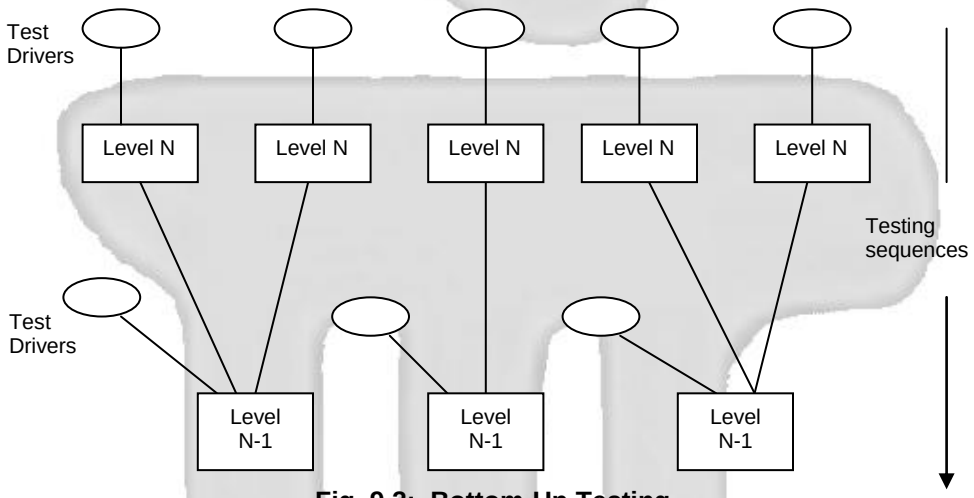


Fig. 9.2: Bottom-Up Testing

If top-down development is combined with bottom-up testing, all parts of the system must be implemented before testing can begin. Architectural faults are unlikely to be discovered until much of the system has been tested. Correction of these faults might involve the rewriting and consequent re-testing of low-level modules in the system.

A strict top-down development process including testing is an impractical approach, particularly if existing software components are to be reused. Bottom-up testing of critical, low-level system components is almost always necessary.

Bottom-up testing is appropriate for object-oriented systems in that individual objects may be tested using their own test drivers they are then integrated and the object collection is tested. The testing of these collections should focus on object interactions.

9.4.3 Thread testing

Thread testing is a testing strategy, which was devised for testing real-time systems. It is an event-based approach where tests are based on the events, which trigger system actions. A comparable approach may be used to test object-oriented systems as they may be modeled as event driven systems.

Thread testing is a testing strategy, which may be used after processes, or objects have been individually tested and integrated in to sub-systems. The processing of each possible external event 'threads' its way through the system processes or objects with some processing carried out at each stage. Thread testing involves identifying and executing each possible processing 'thread'.

Of course, complete thread testing may be impossible because of the number of possible input and output combinations. In such cases, the most commonly exercised threads should be identified and selected for testing.

9.4.4 Stress testing

Some classes of system are designed to handle specified load. For example, a transaction processing system may be designed to process up to 100 transactions per second; an operating system may be designed to handle up to 200 separate terminals. Tests have to be designed to ensure that the system can process its intended load. This usually involves planning a series of tests where the load is steadily increased.

Stress testing continues these tests beyond the maximum design load of the system until the system fails. This type of testing has two functions:

(1) It tests the failure behavior of the system.
(2) It stresses the system and may cause defects to come to light, which would not normally manifest themselves.

Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded as the network becomes swamped with data, which the different processes must exchange.

9.4.5 Back-to-back testing

Back-to-back testing may be used when more than one version of a system is available for testing. The same tests are presented to both versions of the

system and the test results compared. Difference between these test results highlights potential system problems (figure 9.3).

Back-to-back testing is only usually possible in the following situations:

(1) When a system prototype is available.
(2) When reliable systems are developed using N-version programming.
(3) When different versions of a system have been developed for different types of computers.

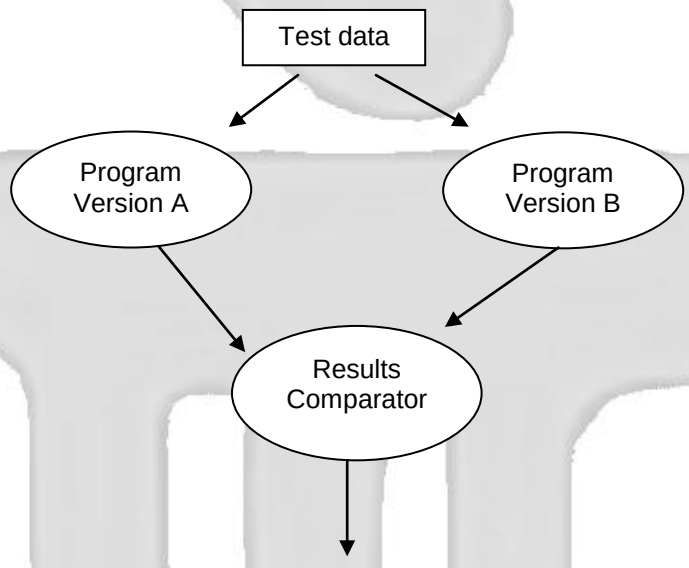


Fig. 9.3: Back to back testing

Steps involved in back-to-back testing are:

Step 1: prepare a general-purpose set of test case.

Step 2: run one version of the program with these test cases and save the results in more than one files

Step 3: run another version of the program with the same test cases, saving the results to a different file.

Step 4: automatically compare the files produced by the modified and unmodified program versions.

If the programs behave in the same way, the file comparison should show the output files to be identical. Although this does not guarantee that they are valid (the implementers of both versions may have made the same

mistake), it is probable that the programs are behaving correctly. Differences between the outputs suggest problems, which should be investigated in more detail.

9.5 Principles of Testing

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. The following are the list of testing principles.

- **All tests should be traceable to customer requirements.** As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- **Tests should be planned long before testing begins.** Test planning can begin as soon as the requirement model is complete. Detailed definition of test can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
- **The Pareto principle applies to software testing.** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem of course, is to isolate these suspect components and to thoroughly test them.
- **Testing should begin "in the small" and progress toward "in the large".** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
- **Exhaustive testing is not possible.** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
- **To be most effective, testing should be conducted by an independent third party.** By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing).

Software engineer who created the system is not the best person to conduct all the tests for the software.

9.6 Testing Methods and Tools

9.6.1 Testing through reviews

Formal technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high-quality software.

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 and defects that can then removed. Softer reviews "purify" the software engineering activities that we have called analysis, design, and coding.

Many different types of reviews can be conducted.

- An informal meeting around the coffee machine is a form of review, if technical problems are discussed.
- A formal presentation of software design to audience of customers, management, and technical staff is, also a form of review.
- Formal technical reviews some times called a *walkthrough* or an *inspection*. This is most effective filter from a quality assurance standpoint.

9.6.2 Black-box testing (Functional testing)

Black box testing alludes to tests that are conducted at the software interface. Although they are designed to uncover errors, black box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information (e.g., a database) is maintained. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

In the black-box approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason black-box testing is also known as functional testing.

9.6.3 White box testing (glass-box testing)

White-box testing of software is predicated on close examination of procedural detail. Providing test cases that exercise specific sets of conditions and/or loops tests logical paths through the software. The “status of the program” may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box methods, the software engineer can derive test cases that

1. Guarantee that all independent paths within a module have been exercised at least once,
2. Exercise all logical decisions on their true and false sides,
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise internal data structures to ensure their validity.

Designing white-box test cases requires thorough knowledge of the internal structure of software, and therefore white-box testing is also called the *structural testing*.

Testing programs testing in the small and testing in the large

During testing, the program to be tested is executed with a set of test cases, and the output of the program for the test cases is evaluated to determine if the program is performing as expected. Due to its approach, dynamic testing can only ascertain the presence of errors in the program; the exact nature of the errors is not usually decided by testing. Testing forms the first step in determining the errors in a program. Clearly, the success of testing is in revealing errors in programs depend critically on test cases.

Testing a large system is a complex activity, and like any complex activity it has to be broken in to smaller activities. Due to this, for a project, incremental testing *is generally performed, in which components and subsystems of the system are tested separately before integrating them to form the system, for system testing. This form of testing, though necessary to ensure quality for a large system, introduces new issues of how to select components for testing and how to combine them to form subsystems and systems.*

9.6.4 Testing software changes

The changes that will affect software engineering will be influenced from four simultaneous sources:

1. The people who do not work,
2. The process that they apply,
3. The nature of information,
4. The under laying computing technology.

Self Assessment Questions

4. _____ tests the high levels of a system before testing its detailed components.
5. _____ involves testing the modules at the lower levels in the hierarchy, and then working up the hierarchy of modules until the final module is tested.
6. _____ is a testing strategy, which was devised for testing real-time systems.

9.7 Additional requirements in testing OO Systems

OOA-object oriented analysis-is based upon the concepts like objects and attributes, classes and numbers, wholes and parts.

In order to build an analysis model five basic principles were applied:

1. The information domain is modeled;
2. Function is described;
3. Behavior is represented;
4. Data, functional and behavioral models are partitioned to expose greater details; and
5. Early models represent the essence of the problem while later models provide implementation details

To accomplish this, a number of tasks must occur:

- Basic user requirements must be communicated between the customer and the software engineer.
- Classes must be identified (i.e., attributes and methods are defined).
- A class hierarchy must be specified.
- Object–to–object relationships (object connections) should be represented.
- Object behavior must be modeled.

- The above tasks are reapplied iteratively until the model is complete.

It is important to note that there is no universal agreement on the “concepts” that serve as a foundation for OO.

9.8 System Testing

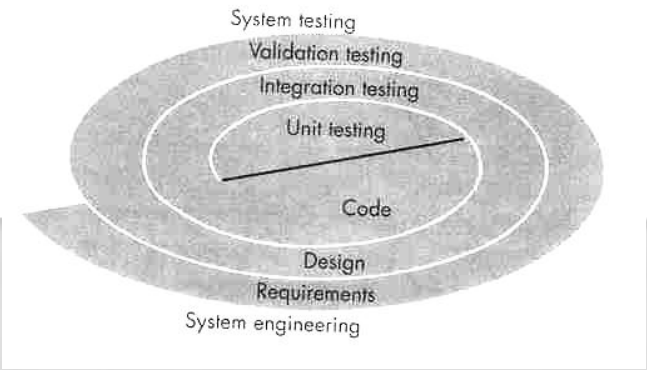


Fig. 9.4: System testing

A strategy for software testing may also be viewed as in the context of the spiral (Figure 9.4) unit testing begins at the vertex of the spiral and concentrated on each unit by moving outward along the spiral to integration testing, where the focus is on design and the construction of software architecture. Taking another run outward on the spiral, we encounter validation testing, where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at system testing, where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn. System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

9.9 Acceptance Testing

Testing is usually relied on to detect the faults, in addition to the faults introduced during coding phase itself. Due to this, different levels of testing are used in testing process; each level of testing aims to test different aspects of the system.

The basic levels are unit testing, integration testing and system and *acceptance testing.* These different levels of testing attempt to detect different types of faults.

The relation of faults introduced in different phases and the different levels of testing is shown in figure 9.5.

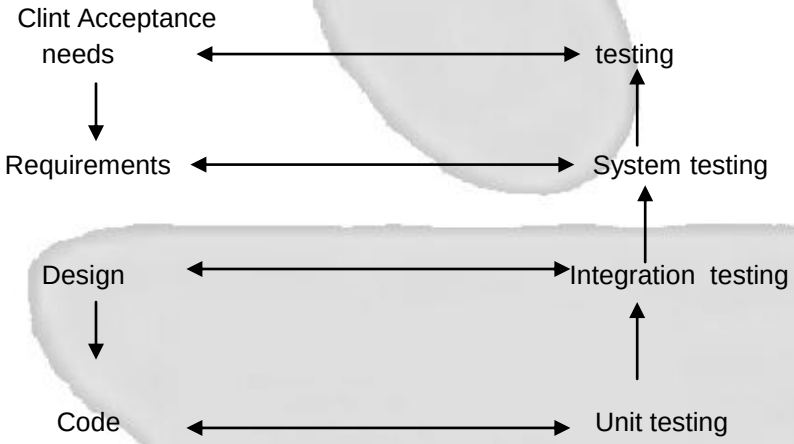


Fig. 9.5: Acceptance testing

In acceptance testing, the entire software system is tested. The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements. This is essentially a validation exercise, and in many situations it is the only validation activity.

Acceptance testing is sometimes performed with realistic data of the client to demonstrate the software working satisfactorily. Testing here focuses on the external behavior of the system; the internal logic of the program is not emphasized. Consequently, mostly functional testing is performed at these levels.

9.10 Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with function that previously worked flawlessly. In the context of an integration test strategy*, regression testing is* the re-execution of some subset of tests that

have already been conducted to ensure that changes have not propagated unintended side effects.

Regression testing is the activity that helps to ensure that changes (due to testing or other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools*. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and caparison.

The regression test suite contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

9.11 Metrics Collection, Computation, and Evaluation

The process for establishing a baseline is illustrated in (Figure 9.6). Ideally, data needed to establish a baseline has been collected in an on-going manner. Sadly, this is rarely the case. Therefore, data collection requires an historical investigation of past projects to reconstruct required data. Once data have been collected (unquestionably the most difficult step), metrics computation is possible. Depending on the breadth of data collected, metrics can span a broad range of LOC or FP measures. Finally, computed data must be evaluated and applied in estimation. Data evaluation focuses on the underlying reasons for the results obtained. Are the computed averages relevant to the project at hand? What extenuating circumstances invalidate certain data for use in this estimate? These and other questions must be addressed so that metrics data are not used blindly.

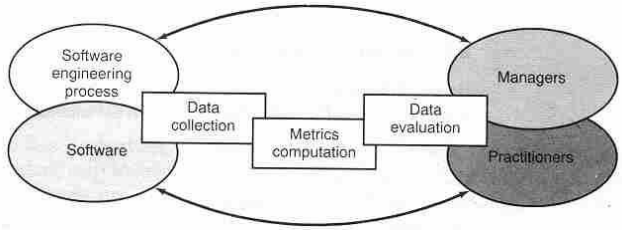


Fig. 9.6: Metrics Collection process

COST DATA INPUT

DESCRIPTION	UNITS	Example Data
labor cost	$/person-month	$7,744
labor year	hrs/year	1560

DATA FOR METRICS COMPUTATION

DESCRIPTION	UNITS	Example Data
project name or identifier	alphanumeric	Proj#1
release type (development/maint)	alphanumeric	maintenance
number of project staff	people	3
effort	person-hours	4800
effort, computed	person-months	36.9
elapsed time to completion	months	13.0
source lines newly developed	KLOC	11.5
source lines modified (existing code)	KLOC	0.4
source lines delivered (includes existing LOC)	KLOC	33.4
source lines reused (from other programs/libraries)	KLOC	0.8
number of separate programs	programs	1
technical documentation	pages	265
user documentation	pages	122
number of errors (1st year after release)	errors	26
maintenance effort-modifications (1st year)	person-hours	3550
maintenance effort-errors (1st year)	person-hours	1970
PROJECT DATA		
Percentage of total project time spent on:		
problem analysis & specification	%	18%
design	%	20%
coding	%	23%
testing	%	25%
other_describe	%	14%

FUNCTION-ORIENTED DATA INPUT

DESCRIPTION	UNITS	DATA
Information Domain		
1. no. user inputs	inputs	24
2. no. user outputs	outputs	46
3. no. use inquiries	inquiries	8
4. no. files	files	4
5. no. external interfaces	interfaces	2
Weighting Values		
Use first number for simple, second for average, and last for complex		
1. input weights	3, 4, 6	4
2. outputs	4, 5, 7	4
3. inquiries	3, 4, 6	6
4. files	7, 10, 15	10
5. interfaces	5, 7, 10	5
Processing Complexity Factors		
Factors rated from 0 to 5 where:		
no influence (0); incidental (1); moderate (2); average (3); significant (4); essential (5)		
1. backup and recovery required	0,1,2,3,4,5	4
2. data communication required	0,1,2,3,4,5	1

Fig. 9.7: Software metrics: collection, computation and evaluation

3. distributed processing function	0,1,2,3,4,5	0
4. performance critical	0,1,2,3,4,5	3
5. heavily utilized operating environment	0,1,2,3,4,5	3
6. online data entry	0,1,2,3,4,5	5
7. input transaction with multiple screens	0,1,2,3,4,5	4
8. master files updated online	0,1,2,3,4,5	4
9. input, output, files, inquiries complex	0,1,2,3,4,5	3
10. internal processing complex	0,1,2,3,4,5	3
11. code designed to be reusable	0,1,2,3,4,5	2
12. conv./installation included in design	0,1,2,3,4,5	2
13.system design for multiple installations	0,1,2,3,4,5	4
14. maintainability/ease of use	0,1,2,3,4,5	5

SIZE-ORIENTED METRICS

DESCRIPTION	UNITS	DATA
Productivity and Cost Metrics:		
project name	alphanumeric	Proj #1
output	KLOC/p-m	0.905
cost-all maintained code	$/KLOC	$22,514
cost excluding reused	$/KLOC	$24,028
elapsed time	months/KLOC	1.0
documentation	pages/KLOC	30
documentation	pages/p-m	10
documentation	$/page	$739
Quality Metrics:		
defects	errors/KLOC	2.0
cost of errors	$/error	$376
maint. errors/total maint.	ratio	0.36
maint. mods./total maint.	ratio	0.64
maint. effort/dev. effort	ratio	1.15

Note: Size oriented metrics computed above use "KLOC maintained," that is, source lines newly developed, modified and reused only.

FUNCTION-ORIENTED METRICS

DESCRIPTION	UNITS	DATA
Function-Point Computations		
unadjusted function points		378
total degree of influence		43
complexity adjustment		1.08
function points (FP)		408
Productivity and Cost Metrics FP		
project name	alphanumeric	Proj # 1
output	FP/p-m	11.1
cost	$/FP	$700
elapsed time	FP/month	31.4
documentation	pages/FP	0.9
Functionality		
program size	FP/program	408
function to size	FP/KLOC-maintained	32
Quality Metrics		
defects	errors/FP	0.064
maint. effort-errors	person-days/FP	0.817
maint. effort-mods.	person-days/FP	1.472

Fig. 9.7 (continued)

Figure 9.7 presents a spreadsheet model for collection and computation of historical software baseline data. Note that the model includes cost data, size-oriented data, and function-oriented data, enabling computation of both LOC- and FP-oriented metrics. It should be noted that it is not always possible to collect all data requested in this model. If we apply such a model to a number of past projects, a software metrics baseline will have been established.

9.12 Test and QA plan

To ensure that the final product produced is of high quality, some quality control activities must be performed throughout the development. Correcting of errors in the final stages can be very expensive, especially if they originated in the early phases. The purpose of the software quality assurance plans (SQAP) is to specify all the work products that need to be produced during the project, activities that need to be performed for checking the quality of each of the work products and the tools and methods that may be used for the SQA activities.

SQAP takes a broad view of quality. It is interested in the quality of not only the final product but also the intermediate products, even though in a project we are ultimately interested in the quality of the delivered product. This is due to the fact that in a project it is very unlikely that the intermediate work products are of poor quality, but the final product is of high quality. The sentence is incomplete. For this reason, an SQAP will contain QA activities throughout the project.

The SQAP specifies the tasks that need to be undertaken at different times in the life cycle to improve the software quality and how they are to be managed. These tasks will generally include reviews and audits. Each task should defined with an entry and exit criterion, that is, the criterion that should be satisfied to initiate the task and the criterion that should be satisfied to terminate the task. Both criteria should be stated so that they can be evaluated objectively. The responsibilities for different tasks should also be identified.

The documents that should be produced during software development to enhance software quality should also be specified by the SQAP. It should identify all documents that govern the developments, verification, validation,

use, and maintenance of the software and how these documents are to be checked for adequacy.

9.13 Managing Testing Functions

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

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

- Testing begins at the module level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

Test management tools are used to control and coordinate software testing for each of the major testing steps. Tools in this category manage and co-ordinate regression testing, perform comparisons that ascertain differences between actual and expected output, and conduct batch testing of programs with interactive human-computer interfaces.

In addition to the functions noted above, many test management tools also serve as generic test drivers. A test driver reads one or more test cases from a testing file, formats the test data to conform to the needs of the software under test and then invokes the software to be tested. Testing tools in this sub category are customized by the tester to meet specialized testing needs.

Finally, test managers sometimes work in conjunction with requirements tracing tools to provide requirements coverage analysis for testing. Reading each test case in sequence, the requirement coverage analyzer attempts to determine (based on information that describes the purpose of the test case) which software requirements are addressed by the test. A cross

reference matrix is often used to indicate which tests address, what requirements

Self Assessment Questions

7. In _____ the software and other system elements are tested as a whole.
8. _____ is the activity that helps to ensure that changes (due to testing or other reasons) do not introduce unintended behavior or additional errors.
9. _____ are used to control and coordinate software testing for each of the major testing steps.

9.14 Summary

- Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
- Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.
- The Test plan section describes the overall strategy for integration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software.
- There are five different types of test strategies namely, Top-down testing, Bottom-up Testing, Thread testing, and Stress testing, Back-to-back testing.
- A software engineer must understand the basic principles that guide software testing. The following testing principles have been highlighted: All tests should be traceable to customer requirements, Tests should be planned long before testing begins, The Pareto principle applies to software testing
- Testing should begin “in the small” and progress toward “in the large”. Exhaustive testing is not possible. To be most effective, testing should be conducted by an independent third party.
- A brief discussion of testing OO system is also highlighted.

- System testing verifies that all elements mesh properly and that overall system function/performance is achieved. This has been explained using a spiral model.
- In acceptance testing the entire software system is tested. The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements. This is essentially a validation exercise, and in many situations it is the only validation activity.
- In the context of an integration test strategy*, regression testing is* the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Regression testing is the activity that helps to ensure that changes (due to testing or other reasons) do not introduce unintended behavior or additional errors. data collection requires an historical investigation of past projects to reconstruct required data. Once data have been collected (unquestionably the most difficult step), metrics computation is possible. Depending on the breadth of data collected, metrics can span a broad range of LOC or FP measures. Finally, computed data must be evaluated and applied in estimation. Data evaluation focuses on the underlying reasons for the results obtained.

9.15 Terminal Questions

1. Explain Verification and Validation (V&V).
2. Describe the various testing strategies.
3. Give a brief account of Regression Testing.

9.16 Answers

Self Assessment Questions

1. Verification
2. Validation
3. Black-box tests
4. Top-down testing
5. Bottom-up testing
6. Thread testing
7. System testing
8. Regression testing
9. Test management tools

Terminal Questions

1. Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. (Refer section 9.2)
2. Top-down testing, Bottom-up testing, White-box testing, Black-box testing etc. (Refer section 9.4).
3. Regression testing is the activity that helps to ensure that changes (due to testing or other reasons) do not introduce unintended behavior or additional errors. (Refer section 9.10)

Unit 10 Software Testing Strategies

Structure:

10.1 Introduction

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

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

- Testing begins at the component level 2 and works 'outward' towards the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

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

progress must be measurable and problems must surface as early as possible.

Objectives:

After studying this unit, you should be able to:

- organize for software testing
- explain various software testing strategies
- discuss top-down integration and bottom-up integration methods

10.2 Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests militates against thorough testing.

10.3 Software Testing Strategy

The software engineering process may be viewed as the spiral illustrated in Figure 10.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software- are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.

A strategy for software testing may also be viewed in the context of the spiral. Unit testing begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses outwards along the spiral to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter validation testing, where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at system testing, where the software and other system elements are

tested as a whole. To test computer software, we spiral out along stream-lines that broaden the scope of testing with each turn.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in Figure 10.1. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name is unit testing. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Black-box test case design techniques are the most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of high-order tests are conducted. Validation criteria (established during requirements analysis) must be tested. Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements. Black-box testing techniques are used exclusively during validation.

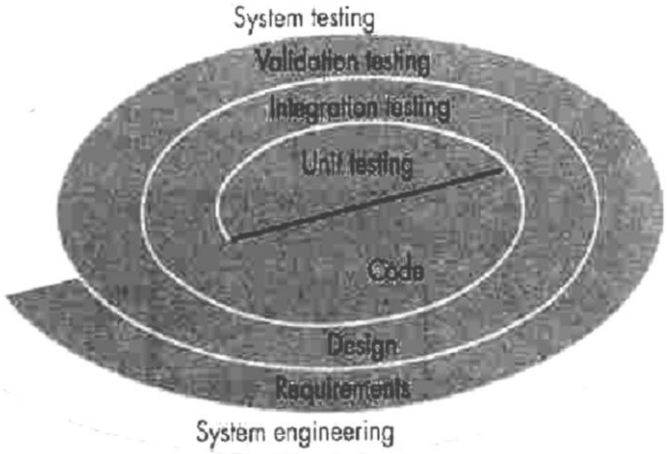


Fig. 10.1: Software Testing Strategy

Self Assessment Questions

1. _____ begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code.
2. _____ addresses the issues associated with the dual problems of verification and program construction.
3. _____ provides final assurance that software meets all functional, behavioral, and performance requirements.

10.4 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design-the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

10.4.1 Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in figure 10.2. The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

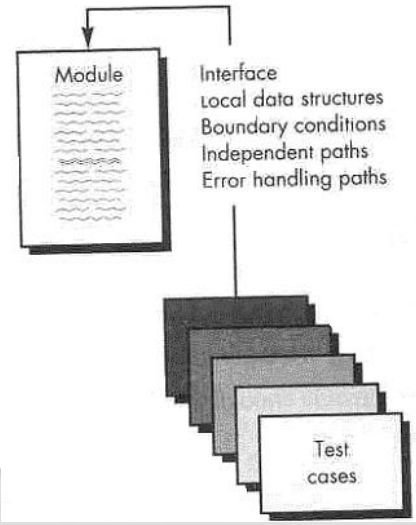


Fig. 10.2: Unit Test Considerations

10.5 Top-down Integration Testing

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

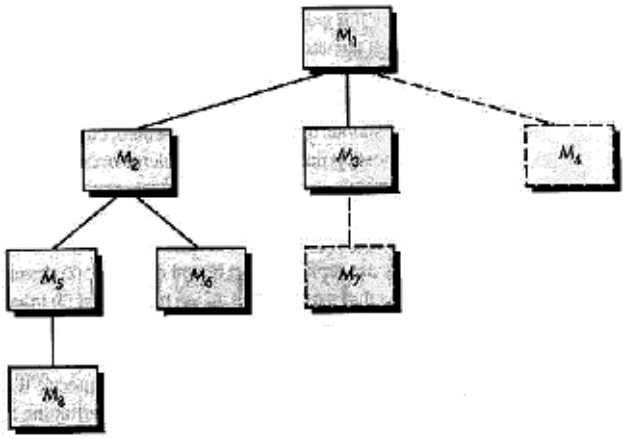


Fig. 10.3: Top-down Integration Testing

10.6 Bottom-up Integration Testing

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

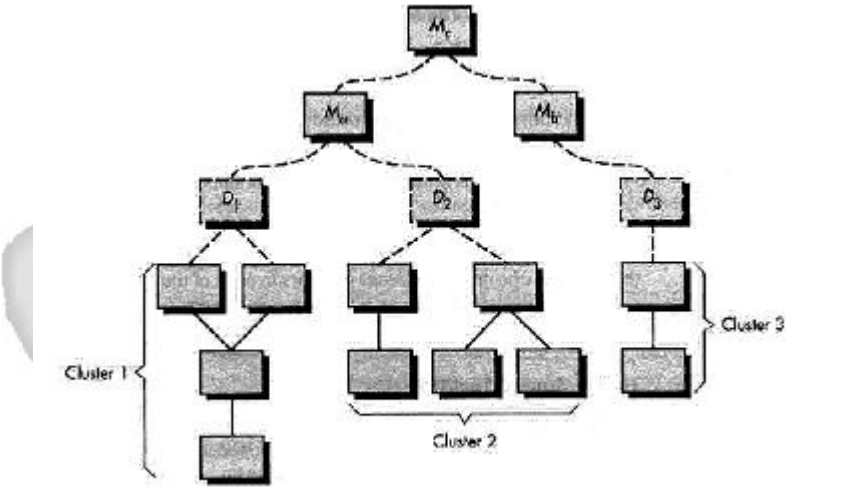


Fig. 10.4: Bottom-up Integration Testing

Self Assessment Questions

4. The unit test is _____ oriented, and the steps can be conducted in parallel for multiple components.
5. Top-down integration testing is a/an _____ approach to construction of program structure.
6. _____, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

10.7 Summary

- Software testing accounts for the largest percentage of technical effort in the soft ware process.
- Yet we are only beginning to understand the subtleties of systematic test planning, execution, and control.
- The objective of software testing is to uncover errors.

- To fulfill this objective, a series of test steps-unit, integration, validation, and system tests-are planned and executed.
- Unit and integration tests concentrate on functional verification of a component and incorporation of components into a program structure.
- Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system.
- Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases.
- With each testing step, the level of abstraction with which software is considered is broadened.
- Unlike testing (a systematic, planned activity), debugging must be viewed as an art.
- Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error.
- Of the many resources available during debugging, the most valuable is the counsel of other members of the software engineering staff.

10.8 Terminal Questions

1. Write a note on Software Testing Strategy
2. What is Unit Testing? Explain.
3. Explain the process of Top-down integration and Bottom-up Integration.

10.9 Answers

Self Assessment Questions

1. Unit testing
2. Integration testing
3. Validation testing
4. White-box
5. Incremental
6. Bottom-up integration testing

Terminal Questions

1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software- are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn. (Refer section 10.3)
2. Unit testing focuses verification effort on the smallest unit of software design-the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components. (Refer section 10.4)
3. Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated. (Refer sections 10.5 and 10.6)

Unit 11 People and Software Engineering

Structure:

11.1 Introduction

Multidisciplinary thinking helps us understand problems better and therefore solve problems more effectively. Previous units have illustrated this at the *process* level and examined process structure, process models, process activities, and problem analysis as initial components of the problem-solving process. This unit considers multidisciplinary thinking at the resource level, specifically in terms of its *people* dimension (see Figure 11.1).

Objectives:

After studying this unit, you should be able to:

- explain traditional software engineering
- discuss the importance of people in problem solving process
- describe human driven software engineering
- explore people factor in multidisciplinary aspects

11.2 Traditional Software Engineering

Traditionally, software engineering has considered people as a resource only if they were explicitly involved in carrying out software development tasks – analysis to design for implementation. In interdisciplinary software engineering, the concept of people as a resource extends beyond those who are immediately involved to encompass all the individuals who play a

significant role in the problem-solving process, regardless of whether they are officially affiliated with the development team. This more inclusive concept comprises those informal but nonetheless critical human resources without whose cooperation the problem cannot be adequately solved: informal resources engaged through a process of collaboration rather than formal affiliation. Examples of collaborative human resources include such stakeholders as customers, managers, and group clients.

Extended knowledge. This refers to applying an extended knowledge base to the entire problem-solving process, thus allowing the problem to be viewed from many alternative angles so that the solution receives a sufficiently broad analysis.

11.3 The Importance of People in the Problem-Solving Process

People are at the core of problem solving because business problems are solved by people for people. The problem solvers are not just the software developers. Business problem solving is collaborative and requires ongoing management support, commitment, and understanding. It also requires significant cooperation from the relevant organizational units and employees of a business. In some cases, an organization may constitute a team to work formally with a development group. This is especially so for software solutions that must comply with organizational quality management rules and standards. The structure of the group and its efficiency of communications, style of management, and cohesiveness are critical factors in the effectiveness of the team.

11.3.1 The roles of users in problem definition

The term '*user'* is too narrow to reflect adequately the variety of stake-holders who can affect or be affected by the evolution of a problem's definition and its eventual solution. Figure 11.1 provides an overview of the kinds of people involved in this process.

At the staff level, for example, many kinds of direct and indirect users need to be considered. A system operator, responsible for inputting and updating data, monitoring progress, and generating reports, represents the simplest example of a direct staff user. Salesmen interoperate with systems at a mobile or remote level in order to access inventory, check prices, and close deals. Inventory personnel track data, input updates, and initiate new

requests. The accounting department interacts with the system at the financial level. Marketing, legal, personnel, and other administrative departments also input relevant data, output reports, and monitor progress from their specialized perspectives. System engineers, database managers, and software or hardware specialists use the system and view its problem domain from very different viewpoints.

The current or potential customers of an enterprise are also key stakeholders and may be direct or indirect users of a system. A user receiving only outputs from a system, such as payroll reports or account, strongly affects the nature of their expectations about solutions. As indicated previously, problem complexity is typically low at the operational level because operational problems are likely to be well structured. However, at the tactical and strategic management levels, problems tend to be semi-structured or ill structured. Management requirements vary widely from department to department because their variation in needs and different contexts lead to diverse problem definition views.

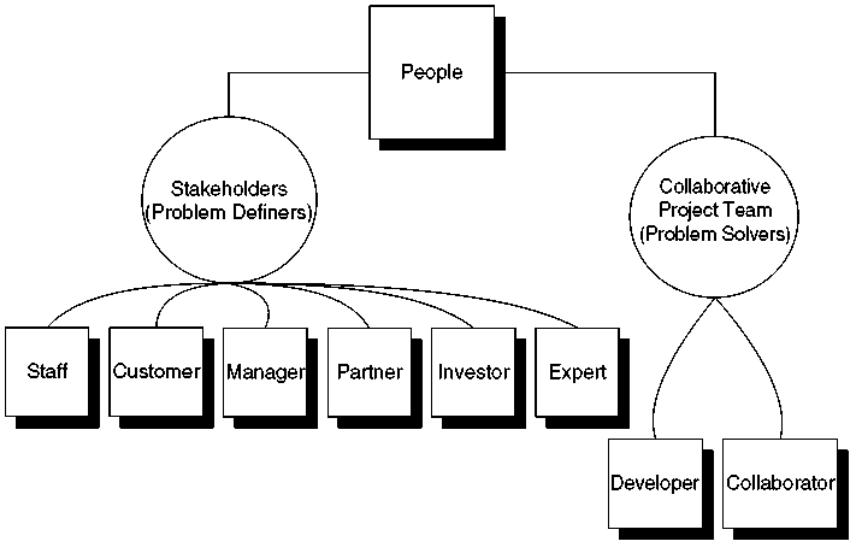


Fig. 11.1: The Role of Users in Problem Solving

Thus, financial managers want systems that help assess the financial performance of the organization. Research and development managers want the ability to examine the quality of products and services and track operational costs throughout an organization. Production managers want

software solutions that support resource planning in terms of required labor and materials and that assist them in reducing errors and maximizing productivity. Marketing managers look forward to software solutions that provide descriptive and inferential statistics across geographic locations, and among various salesmen and different products and brands.

The business partners or collaborators in the supply chain are other essential stakeholders. Many organizations currently link their corporate intranets to extranet systems accessible to their business partners. Although security and privacy considerations apply, these extranets can be extended to larger geographic or metropolitan areas through metro-nets. Defining a problem at an intranet level is obviously less complex than defining a problem at a metro-net or global level, and the risks involved may have a very different character.

Investors and owners are also significant stakeholders who make demands on a system in terms of financial goals, net present values, breakeven points, and return on investment. Experts and consultants are another stakeholder class who can strongly affect problem definition. Whether experts are sought from within an organization or from outside, their experiences make their viewpoints of key value to the entire process, possibly representing the difference between project success and failure and minimally offering insights that can save time and money.

Self Assessment Questions

1. Traditionally, software engineering has considered _____ as a resource only if they were explicitly involved in carrying out software development tasks – analysis to design for implementation.
2. The term _____ is too narrow to reflect adequately the variety of stake-holders who can affect or be affected by the evolution of a problem's definition and its eventual solution.
3. Many organizations currently link their corporate _____ to extranet systems accessible to their business partners.

11.4 Human-Driven Software Engineering

The most critical resource in the problem-solving process is *people*. Whether they are staff members, customers, managers, partners, investors, or experts and whether their involvement is direct or indirect, their role in

functional and interface requirements at the problem definition and solution construction levels is essential. Thus, if one is to achieve success across the software development process, the following people-driven issues must be addressed effectively:

- **Stakeholders.** The various requirements of all the stakeholders must be satisfied.
- **Customers.** To attract customers away from alternatives, the product must not just be competitive and of high quality, it must also be in time to market, have appropriately attractive features, and be priced well.
- **Development team.** It is given that the development team must be qualified and skilled, but the team must also have sufficient multidisciplinary skills to truly meet the underlying project requirements.
- **Project manager.** The project manager must have interdisciplinary skills beyond the customary prerequisite ability to manage, coordinate, control, plan, and communicate effectively.
- **Partners.** Partners are an essential part of supply chain management. The partners may be identified as stakeholders or as a component of supply chain management.
- **Groups.** There are groups of developers, groups of customers, groups of managers, and so on. These groups must all exhibit successful communication, collaboration, and management mechanisms.

To utilize human resources efficiently, one must identify and locate the people who are important to truly understand a problem and assisting in its solution; who are able to document the information needed in order to build a knowledge inventory for the entire problem-solving process; and who can bring this information to bear to guide the proposed solution of the problem. One must also obtain feedback in order to validate and verify that needs and expectations are reflected in the proposed solution. In addition, it is necessary to train those who will work on the development team or collaborate at the organizational level to accomplish the system goals and deliver the expected business value. Figure 11.2 illustrates the role of the people factor in the problem-solving process.

11.5 The People Factor – Multidisciplinary Aspects

The multidisciplinary aspects of the people factor manifest themselves at the problem and the solution level. At the problem level, the issue is which

people-related disciplines can help one better understand the underlying problem. At the solution level, the main concerns are the people-related disciplines that enable one to address problem solving better. Table 11.1 offers an overview of these issues.

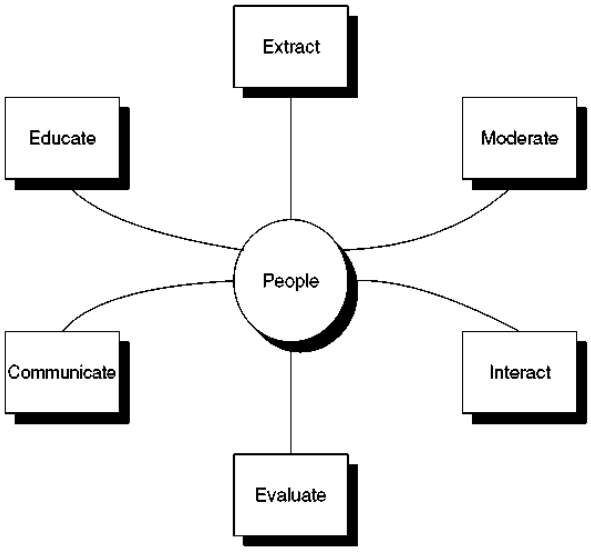


Fig. 11.2: The Role of people factor in Problem Solving

Interdisciplinary Driver	*Relevant Discipline*	*Utilization*
Customers	Marketing	Discover more subcategories of customers and other stakeholders Explore systems requirements from a marketing perspective
Development team	Project management	Structuring the development team
Users	Cognitive psychology	User interface design

Table 11.1: The People Factor – Multidisciplinary Aspects

In information systems, making successful IT projects is a required common denominator for overall business success. Despite this, many organizations experienced a high failure rate in IT projects, which made the improvement of IT project management even more critical. However, although extensive research and development of new methodologies for IT projects was

conducted during this period, little corresponding improvement appeared to take place in IT development. The implication is that IT project management is not a simple matter of identifying requisite skills and applying methodologies, but an emerging discipline that still demands extensive further research.

Before proceeding, it is necessary to recap briefly what a project is. A project can be thought of as a group of tasks and activities performed within a definable period and meeting a specific set of objectives. A project involves a temporary assemblage of resources brought together to solve a specific problem. Tatnall and Shackleton (1885), Rosenau (1888), and Meredith and Mantel (1885) identify several characteristic features of projects. Projects are unique. The degree of uniqueness may vary, but all projects are essentially one-of-a-kind, nonrecurring undertakings.

Projects vary in size but exhibit characteristics that distinguish them from other types of work efforts. For example, projects have specific objectives, must be completed within a given budget, and are carried out by teams. The assignment of people to a project team may be on a fulltime or part-time basis, depending on the specific needs of the project. Projects must be completed within a specific time period and have well-defined beginnings and ends. Correct project definition is critical to project management. The project definition helps establish a clear scope for the project and serves as a basis for project planning.

The steps needed to define a project begin with describing the opportunities that brought about the project in the first place; supplying a description of the background that established the need for the project; and then defining the goals for the project. After identifying the stakeholders and available resources, one must also identify any related projects that will affect or be affected by the project under consideration. One then identifies the criteria for deciding whether a project is viable, including understanding project constraints, assumptions, and risks, as well as the implication of such constraints and assumptions for the project risks.

Project management can be defined as a set of principles, methods, tools, and techniques for planning, organizing, staffing, directing, and controlling project-related activities in order to achieve project objectives within time and under cost and performance constraints. The project management

process faces the often daunting task of assembling a project team that has the expertise needed to implement a project, establishing the technical objectives of the project, dynamically managing changes in order to meet requirements, and planning and controlling the project so that it completes on schedule and within budget. Project management is applicable in any situation in which resources must be efficiently applied to achieve demanding goals under significant time and cost constraints, and serious ramifications will occur if the expected results are not met on time, on budget, at required standards, and to stakeholder satisfaction. One can classify project management activities according to the phase of the project:

- *Project conception.* The purpose of the conceptual phase is to determine the feasibility of the project. Objectives are examined in the context of the business environment, alternatives are defined and evaluated, and preliminary estimates of cost, schedule, and risk are done. This phase culminates in a decision as to whether to proceed with the project.
- *Planning.* The performance, cost, and schedule estimates are refined to a point at which detailed plans for project execution can be made. Budgets and schedules are developed, the project team is formed, and a project management system is established to guide the management of the project.
- *Execution.* The program manager's responsibility is to manage the resources necessary to accomplish the objectives. The emphasis of responsibilities shifts from planning to control.
- *Termination.* The project activities are phased out. This can be triggered by premature termination or by successful achievement of the goals. In either case, certain activities are necessary to wrap up the project.

The success of project management depends on factors ranging from managerial leadership and the availability of business and technical documents that properly establish and communicate plans, to organizational or institutional support for developing the managerial skills that enhance people and project management. The most frequently cited management-related difficulties in project management include poorly defined goals and specifications, lack of an adequate project plan, and unrealistic deadlines and budgets.

The effectiveness of the project manager is critical to project success. The qualities that a project manager must possess include an understanding of negotiation techniques, communication and analytical skills, and requisite project knowledge. Control variables that are decisive in predicting the effectiveness of a project manager include the manager's competence as a communicator, skill as a negotiator, and leadership excellence, and whether he or she is a good team worker and has interdisciplinary skills. Project mangers are responsible for directing project resources and developing plans, and must be able to ensure that a project will be completed in a given period of time. They play the essential role of coordinating between and interfacing with customers and management. Project mangers must be able to:

- Optimize the likelihood of overall project success
- Apply the experiences and concepts learned from recent projects to new projects
- Manage the project's priorities
- Resolve conflicts
- Identify weaknesses in the development process and in the solution
- Identify process strengths upon completion of the project
- Expeditiously engage team members to become informed about and involved in the project

Studies of project management in Mateyaschuk (1888), Sauer, Johnston, and Liu (1888), and Posner (1887) identify common skills and traits deemed essential for effective project managers, including:

- Leadership
- Strong planning and organizational skills
- Team-building ability
- Coping skills
- The ability to identify risks and create contingency plans
- The ability to produce reports that can be understood by business managers
- The ability to evaluate information from specialists
- Flexibility and willingness to try new approaches

Feeny and Willcocks (1888) claimed that the two main indicators of a project manager's effectiveness are prior successful project experience and the

credibility with stakeholders. The underlying rationale for this is that such conditions, taken together, help ensure that the project manager has the necessary skills to execute a project and see it through to completion and that the business stakeholders will continue to support the project. Research also suggests that the intangibility, complexity, and volatility of project requirements have a critical impact on the success of software project managers.

11.6 The Team Factor

A *'team'* can be defined as a group of individuals who have been organized for the purpose of working together to achieve a set of objectives that cannot be effectively achieved by the individuals working alone. The effectiveness of a team may be measured in terms ranging from its outcomes to customer acceptance, team capability, and individual satisfaction. Organizational and individual inputs significantly affect the team's inputs. The team work process is characterized by the efforts exerted towards the goal, the knowledge and skills utilized, the strategy adopted, and the dynamics of the group. Team construction and management are a critical challenge in software-driven problem solving. They require:

- Goal identification
- Strategy definition
- Task management
- Time management
- Allocation of resources
- Interdisciplinary team composition
- Span of control
- Training
- Team communication
- Team cohesiveness
- Quality assurance and evaluation

The main characteristics of successful teams include:

- *Shared goal.* There must be a shared awareness of the common team goal among all the team members. This shared goal is the objective that directs, guides, and integrates the individual efforts to achieve the intended results.

- *Effective collaboration.* A team must work as a team. This entails collaborating, individuals making contributions, exchanging their ideas and knowledge, and building interpersonal relationships and trust. The project environment should facilitate and encourage effective collaboration and interoperation.
- *Individual capabilities.* Each team member must be trained and guided so as to be able to cooperate with the other team members towards the common goal.

Some other characteristics of well-functioning teams include:

- Sharing the mission and goal
- Disseminating complete information about schedules, activities and priorities
- Developing an understanding of the roles of each team member
- Communicating clear definitions of authority and decision-making lines
- Understanding the inevitability of conflicts and the need to resolve them
- Efficiently utilizing individual capabilities
- Effectively deploying meetings
- Accurately evaluating the performance of each team member
- Continually updating individual skills to meet evolving needs

Additional indicators of effective operation include a high level of project management involvement and participation, a focus on purpose, shared responsibilities, a high degree of communication, strategically oriented thinking, and rapid response to challenges and opportunities. These team performance characteristics require every team member to contribute ideas, operate in an environment that contains a diversity of skills, appreciate the contributions of others, share knowledge, actively inquire to enhance understanding, participate energetically, and exercise flexibility.

11.7 The Customer Factor

It is a truism that, in a customer-focused economy, software engineering must also be customer driven. This section considers some characteristics and techniques typical of a customer-driven software development environment. These include:

- *Customer-driven development is requirements intensive and features driven.* Because customer needs are the highest priority, they must be

carefully gathered, identified, specified, visualized, and internally prioritized among themselves. As a consequence, requirements engineering becomes the key strategic phase across the software engineering process.

- *Customer-driven development is iterative in nature.* Iterative development is essential because it allows extensive feedback and development response to the feedback.
- *Customer-driven development aims to develop killer applications.* The only way to survive in a highly competitive market is to develop winning applications – not ordinary applications that merely pass the test of basic viability.
- *Customer-driven development strongly values time to market.* Time means opportunity, so applications must be engineered expeditiously enough to capture time-dependent marketing opportunities.
- *Customer-driven development attempts to achieve multi-stakeholder satisfaction via win-win situations.* Every software development activity involves many participants, each of whom has his or her goals and views of what constitutes value. Therefore, the effective reconciliation of conflicts over system requirements becomes a key factor in assuring customer satisfaction.
- *Customer-driven development focuses on quality in products and services.* Quality assurance implies managing software processes in such a way that the developer and the customer are satisfied with the quality and consistency of the goods or services produced or provided.
- *Customer-driven development views customers as partners – not merely as buyers.* In order to assure that customer expectations are met, customers should team up with developers at each phase of the software development process. This can significantly minimize risk and reduce cycle time throughout the development process.
- *Customer-driven development is customizable, personalized, and adaptable to individual needs and changes in needs.* No two businesses or individuals are identical (demands and needs vary and evolve even across a single organization), so recognizing individual differences and organizational diversity is crucial to providing effective solutions.

- *Customer-driven development is driven by cognitive psychology.* Cognitive psychology can be thought of as the language for the source code of the software customer's mind. Therefore, a customer-driven software development approach should examine the extent to which software design accurately reflects the needs of customers as perceived by the customers.
- *Customer-driven development is informative and accessible.* Designing a software solution in the "customer age" requires full customer service and support in terms of well-documented help, interactive Web assistance, and state-of-the-art means of communication. Applications that do not provide support information are subject to customer complaints, dissatisfaction, and rejection.
- *Security and privacy are concerns in any customer-driven solution.* To earn customer trust, software engineers must design reliable systems that are less likely to be vulnerable to privacy invasions or security hackers. Security and privacy are key concerns of software customers.

Self Assessment Questions

4. One must obtain _____ in order to validate and verify that needs and expectations are reflected in the proposed solution.
5. In information systems, making successful IT projects is a required common denominator for overall _____.
6. A _____ can be defined as a group of individuals who have been organized for the purpose of working together to achieve a set of objectives that cannot be effectively achieved by the individuals working alone.

11.8 Summary

The unit mainly deals with the most important factor of the interaction of people with that of the process of software engineering. The role of the people in the development process and the contributions of the users towards the problem definition phase are extremely important. There aspects of the interaction and contribution of people towards the development phase has been discussed in this unit.

11.9 Terminal Questions

1. Give the importance of people in problem solving process.
2. Explain People Factor in Multidisciplinary aspects.
3. What do you mean by Customer Driven software development?

11.10 Answers

Self Assessment Questions

1. People
2. User
3. Intranets
4. Feedback
5. Business success
6. Team

Terminal Questions

1. People are at the core of problem solving because business problems are solved by people for people. The problem solvers are not just the software developers. Business problem solving is collaborative and requires ongoing management support, commitment, and understanding. It also requires significant cooperation from the relevant organizational units and employees of a business. (Refer section 11.3)
2. The multidisciplinary aspects of the people factor manifest themselves at the problem and the solution level. At the problem level, the issue is which people-related disciplines can help one better understand the underlying problem. At the solution level, the main concerns are the people-related disciplines that enable one to address problem solving better. (Refer section 11.5)
3. It is a truism that, in a customer-focused economy, software engineering must also be customer driven. (Refer section 11.7)

Unit 12 Software Technology and Problem Solving

Structure:

12.1 Introduction

Information technology has ubiquitously influenced business and affected management approaches to problem solving. A key manifestation of this technology is the software technology that has pervaded all aspects of life, from household appliances to entertainment devices, communication media, productivity tool-ware, learning systems, and portable devices that operate under the control of embedded, factory preprogrammed chips with settings and parameters controllable through easy-to-use user interfaces. The quintessential software characteristics of flexibility and adaptability have enabled manufacturers to create customized systems that respond to changing customer needs and allow tailoring technology to endlessly diverse business requirements. Problem-solving strategies increasingly depend on software technology as an enabling mechanism and for facilitating decision-making processes. In this context, software technology includes the complete software environment utilized in problem solving, covering application systems, the knowledge base, hardware facilities, and technical resources.

The introduction of information processing has changed the way in which people and organizations address problems. The previous unit considered how problem-solving approaches are closely related to how software development is done. This unit considers how the availability of software tools influences how problem solving is done. Software serves as the critical enabling technology that automates routine problem-solving activities and

interactions, facilitates visualization, supports collocated and distant collaboration, etc.

Because software is enabled by technology, advances in problem solving have become coupled with the rapid advances in technology. Software tools are now pervasively used to support classic problem-solving tasks from data exploration to communication. A similar pervasive adaptation of software and business processes is seen in the rapid re-conceptualization of business operations reflected in the e-business revolution that is reshaping entire industries. The impact of the dramatically increasing portability of computing on business processes and the affect of enhanced digitally driven connectivity on development issues such as product cycle time will also be considered.

Objectives:

After studying this unit, you should be able to:

- explain the software technology as enabling business tool
- discuss the limitations of software technology in business
- describe the problem solving approaches

12.2 Software Technology as Enabling Business Tool

The application of software technology to problem solving exhibits characteristics that are fundamental to business organizations. Software technology allows for the acceleration of the problem-solving process by automating tasks and reducing the need for repetition. This can lead to reducing human errors significantly and thus to more reliable solutions. From a human factor, software technology,

- helps visualize problems so that they can be understood globally and intuitively and controlled effectively
- facilitates communication among problem solvers and creates a collaborative environment for dealing with tasks, documents, conditions, and events allowing for the recording of knowledge and experiences
- frees the problem-solving process from dependency on location, distance, or time
- provides effective tools for collecting and analyzing data and for data mining.

The specific impacts of software technology on business are elaborated in the following sections.

12.2.1 Exponential Growth in Capability

According to Moore's law, the density of digital chips doubles approximately every 18 months but cost remains constant, thus increasing computing power but not price. This in turn fuels software technology as software applications become increasingly powerful based on ever faster hardware platforms. No other problem-solving tools exist whose power expands so rapidly, yet remains so cheap. When the objective is to reduce business product development cycle time under the constraint of limited financial resources, computer technology allows solutions in less time and with lower cost. Due to this correlation with technology, the issue of the development of problem solving is coupled with technological forecasting for the computer industry. Next, the implications for business problem solving of the evolving power of computing will be considered.

12.2.2 Business Problem-Solving Optimization

As people solve problems, they rely on computer hardware and software to store and retrieve data, explore solution alternatives, use communication technology to interact with others, utilize perceived if – then rules to make decisions, and process data, knowledge, and techniques to implement solutions. Software technology can shorten this process, potentially translating it into a single application requiring only a single stage of inputs with solutions delivered rapidly.

Database management systems and information retrieval systems can serve as dynamic relational memories that not only store and retrieve data, but also link related components together. Memory and retrieval systems may be supplemented by the ability to recognize manual inputs using techniques such as optical character recognition or voice recognition technology. Expert and AI-based systems can harness the knowledge developed by experts, and Web-based applications can facilitate almost instantaneous communication, dramatically enhancing the ability to collaborate with widely distributed team members and other human resources. Web applications can also serve as a repository to store, retrieve, and search for data and information. The navigation power of the Web transfers the market power from producers and vendors to customers

and helps suppliers to provide better quality products with shorter turnaround. Software technology has enabled breakthrough transformations in businesses and provided benefits that have included:

- Simplification of business structures
- Removal of unnecessary processes
- Overall quality improvement
- Reduction in time to market
- Organizational flexibility
- Cost reduction
- Bringing the benefits of a more innovative business culture

12.2.3 The E-Business Revolution

Metcalfe's law observes that networks increase in value with each additional node (user) in proportion to the square of the number of users. This relationship follows because, with n nodes directly or indirectly interconnected, $n(n – 1)/2$ total possible interconnections are available. The telephone network is a classic instance of the effect of this kind of utility behavior. When the network is small, its overall value is relatively limited. As the network encompasses more users, its benefit grows disproportionately, with the individual benefit growing linearly in the number of users, n, and the total network benefit growing quadraticaly in n.

E-business illustrates the impact of networking power on industry. E-business led to the generation of value-chain partnerships, new ways of interacting with customers and new services. This e-transformation introduced the concept of a virtual organization to business. One consequence is the acceleration of the decision – making process. E-transformation removed or changed the character of business boundaries, including those between the inside and outside of a company, and opened companies to partnerships from unexpected sources, including new relationships with partners, providers, and even competitors. Moreover, e-business capabilities enabled an integrated back-end–front-end architecture that allows online sales and physical activities to support each other in an almost real-time manner.

Web-enabled business processes in the network economy include front-end functions that cover business-to-customer transactions and back-end transactions that define relationships with vendors and partners. This

includes inter-functional processes for internal data exchanges, viewing a business as having customers at both ends of its processes, and ensuring objectives are driven by customer satisfaction (Grover, Fiedler, & Tang 1994). Successful technological, Web-engineered processes triggered by the Internet have contributed to business the ability to slash inventories; customize products; bridge the communication gap between suppliers and individual customers; and even design personalized products that can be ordered online. All of these are part of the networked business process (Roberts 2000).

The general pattern of prior economic revolutions is recurring in the case of e-business: an enabling technology (Web engineering) has allowed the creation of a new (business) process that has sparked a global economic transformation (e-commerce). In commerce, such business processes can create an entirely new environment. Web-specific business processes transcend political, cultural, and social divisions to permit dynamic types of interaction between organizations and individuals when anyone anywhere can purchase or sell anything to anyone anywhere anytime via the Web.

Enabling Web solutions in businesses can reshape the entire business operation. The production process can be viewed from point of origin to point of delivery; emails generate inquiries and response turn-around accelerates (Roberts 2000). Therefore, efficient product management becomes a primary concern of the business process. Studies indicate that organizational strategy and sound management techniques result in quality products and profits (Elaina et al. 1995) and thus are integral to the business process. However, continuous improvement is only sustainable given endurance in the technology transformation, and Web-engineering business processes are currently among the decisive factors.

Research also indicates that increased competitiveness is the greatest anticipated benefit of e-commerce as it improves products and makes enterprises more effective and, thus, more competitive (Lederer, Mirchandani, & Sims 1996). With respect to user relationships, integrating business processes with the Internet leads to far greater transparency between customers and suppliers (Roberts 2000). This confirms that user satisfaction is the most widely used single measure of information technology success (Grover et al. 1994). The literature on e-business

suggests overall that an efficient business process in this environment can be achieved when processes are Web driven (or engineered): they are more competitive and more concerned with user relationships and satisfaction, and they require efficient product management.

Despite these revolutionary developments, the historic baseline for business remains the same as it has always been. In the final analysis, income statements and balance sheets remain the fundamental gauges or metrics of business performance. Mass manufacturing profoundly altered business processes, but the fundamental operations of business remained largely the same. Issues such as maintaining market share; ensuring adequate capitalization; sustaining profitability; controlling costs; and motivating workforces have always been primary challenges to managers. The same is true in the Web economy. Management strategies must therefore reconcile this global impact with the perennial need to keep their organizations growing and profitable.

12.2.4 Portability Power

One of the most notable characteristics of organizational problem solving is its frequent dependence on physical (as opposed to digital) resources: people, places, devices, connections, and work-flow documents. These extensively bind the problem-solving process to these resources. These bonds can restrict the ability of organizations to take advantage of opportunities that arise, for example, outside regular operating hours or beyond the physical location of the organization.

Information and software technology help transcend these boundaries by giving employees, decision-makers, and customers increased flexibility. Whether through LANs or wireless connections, one can be connected to the business environment regardless of location, time, or local technical infrastructure. Executive or expert systems that handle structured as well as routine connection problems provide backup for the communication and decision-making link. For example, online dynamic databases eliminate the need for live contact to check inventory or to process orders. Workflow application technology can support business processes and eliminate the need for physical paperwork through the use of smart digital archiving, thus reducing unnecessary organizational expense. These capabilities or opportunities can be further extended through portable devices such as laptops, PDAs, Internet-ready cell phones, optical scanners, etc.

Some conceptual and technological overlap exists between portability and the e-business transformation. However, portability focuses on expanding an organization's ability to work without physical limits, and e-business is related to extending external relationships with partners, vendors, and customers beyond traditional frameworks. E-business is Internet enabled portability, utilizing the Web and other technological capabilities in which information can be transported.

12.2.5 Connectivity Power

Software technology facilitates communication between devices in a multimedia fashion. A computer can be attached to a digital camcorder, TV, printer, scanner, external storage device, PDA, or another networked computer and to the Internet simultaneously. The architectural strategy of integrating these capabilities within a single platform can add more than mere entertainment or aesthetic value to business exchanges. It can lead to an environment in which the cycle time and costs of the business processes can be reduced via an all-in-one architecture. Multimedia data can be captured immediately, edited as required, stored on an electronic portable device, or sent to a vendor, customer, or business partner in almost real time.

Previously, such work required several departments, staff time, experience, and financial and technical resources. With the ability to represent and communicate multimedia information via a connected device with adequate software drivers installed, a well-equipped laptop can reproduce the functionality of an entire office or even an organization. Connectivity power provides unusual solutions to businesses such as manufacturing, engineering, medicine, and sports, as well as many other application domains in which demand for digital image processing, data mining, and feedback control is high.

Self Assessment Questions

1. According to Moore's law, the density of digital chips doubles approximately every _____ but cost remains constant, thus increasing computing power but not price.
2. Metcalfe's law observes that networks increase in value with each additional node (user) in proportion to the _____ of the number of users.
3. _____ illustrates the impact of networking power on industry.

12.3 Software Technology as a Limited Business Tool

Software technology enables business to solve problems more efficiently than otherwise. However, as with any tool, it has its limitations. Solving business problems involves many considerations that transcend hardware or software capabilities. Thus, software solutions can only become effective when they are placed in the context of a more general problem-solving strategy. Software solutions should be seen as essential tools in problem solving that are to be combined with other interdisciplinary tools and capabilities. This kind of interoperation can be achieved by integrating such tools with the software development process. Additionally, the software development process can also be used as a part of a larger problem-solving process that analyzes business problems and designs and generates working solutions with maximum business value. Some examples of this are discussed in the following sections.

12.3.1 People have different needs that change over time

Software technology is limited in its ability to recognize the application or cognitive stylistic differences of individuals or to adapt to the variety of individual needs and requirements. These differences among individuals have multiple causes and include:

- Use of different cognitive styles when approaching problem solving
- Variations in background, experience, levels and kinds of education, and, even more broadly, diversity in culture, values, attitudes, ethical standards, and religions
- Different goals, ambitions, and risk-management strategies
- Assorted levels of involvement and responsibilities in the business organization’s process.

A software system is designed once to work with the entire business environment all the time. However, organizational needs are not stable and can change for many reasons – even over short periods of time – due to changes in personnel, task requirements, educational or training level, or experience. Designing a software system that can adjust, customize, or personalize to such a diversity of needs and variety of cognitive styles in different organizations and dispersed locations is an immense challenge. It entails building a customizable software system and also necessitates a continuous development process to adapt to ongoing changes in the nature of the environment.

12.3.2 Most Users do not Understand Computer Languages

A software solution can only be considered relevant and effective after one has understood the actual user problems. The people who write the source code for computer applications use technical languages to express the solution and, in some cases, they do not thoroughly investigate whether their final product reflects what users asked for. The final product is expected to convert or transform the user's language and expectations in a way that realizes the system's requirements. Otherwise, the system will be a failure in terms of meeting its stated goals appropriately and will fail its validation and verification criteria.

In a utopian environment, end-users could become sufficiently knowledgeable in software development environments and languages so that they could write their software to ensure systems were designed with their own real needs in mind. Of course, by the very nature of the division of expertise, this could rarely happen and so the distance in functional intention between user languages and their translation into programming languages is often considerable. This creates a barrier between software solutions reaching their intended market and users and customers finding reliable solutions.

In many ways, the ideal scenario, in which one approached system design and development from a user point of view, was one of the driving rationales behind the original development of the software engineering discipline. Software engineering was intended as a problem-solving framework that could bridge the gap between user languages (requirements) and computer languages (the final product or source code). In software engineering, the user's linguistic formulation of a problem is first understood and then specified naturally, grammatically, diagrammatically, mathematically, or even automatically; then, it is translated into a preliminary software architecture that can be coded in a programming language. Thus, the underlying objective in software engineering is that the development solutions be truly reflective of user or customer needs.

12.3.3 Decisions and Problems – Complex and Ill Structured

The existence of a negative correlation between organizational complexity and the impact of technical change (Keen 1981) is disputed. More complex organizations have more ill-structured problems (Mitroff & Turoff 1963).

Consequently, their technical requirements in terms of information systems become harder to address. On the other hand, information technology may allow a complex organization to redesign its business processes so that it can manage complexity more effectively (Davenport & Stoddard 1994).

On balance, a negative correlation is likely in complex organizations for many reasons. First, the complexity of an organization increases the degree of ambiguity and equivocality in its operations (Daft & Lengel 1986). Many organizations will not invest resources sufficient to carry out an adequately representative analysis of a problem. Therefore, requirement specifications tend to become less accurate and concise. Implementing a system based on a poor systems analysis increases the likelihood of failure as well as the likelihood of a lack of compatibility with the organization's diverse or competing needs. A demand for careful analysis and feasibility studies to allow a thorough determination of requirements might bring another dimension of complexity to the original problem.

Second, technology faces more people-based resistance in complex organizations (Markus 1983). This can occur because a newly introduced system has not been well engineered according to accurate requirements in the first place, as well as because of the combination of social, psychological, and political factors found in complex organizations. One further factor complicating the effective delivery of computerized systems in large projects is the time that it takes to get key people involved.

Finally, there are obvious differences in the rate of growth for complex organizations and information technology. Although information technology advances rapidly, complex organizations are subject to greater inertia and thus may change relatively slowly. Subsequently, incorporating or synthesizing technical change into an organization becomes a real challenge for individuals and departments and is affected by factors such as adaptability, training, the ability to upgrade, and maintainability. For such reasons, one expects a negative correlation between organizational complexity and the impact of technical change in terms of applying software technology and achieving intended organizational outcomes.

12.3.4 Business View Software Technology as a Black Box for Creating Economic Value

Although software systems play a significant role in business organizations

in terms of business added value, the traditional focus of many organizations has been on their role in cost reduction because software automation can reduce error, minimize effort, and increase productivity. Innovative applications can enable organizations to achieve more than traditional software goals, including the ability to compete more effectively, maximize profitability, and solve complex business problems.

Business goals extend beyond direct financial benefits to include operational metrics involving customer satisfaction, internal processes, and an organization's innovation and improvement activities. Indeed, such operational measures drive future financial performance (Van Der Zee & De Jong 1999). Efficiency, quality, and market share and penetration are other important goals and measures of business vitality (Singleton, McLean, & Altman 1988) that can be dramatically improved by software systems. Moreover, research has shown that organizational performance can be maximized by clearly recognizing the interdependence between social and technological subsystems (Ryan & Harrison 2000). Software systems with Web capabilities can enhance business added value even more effectively through their ability to reach customers, affiliate with partners, and enrich information (Evans & Wurster 1999).

Although some small organizations use software systems only as one of the many tools to achieve financial goals, many organizations have become partially or totally dependent on software systems. Comprehensive software solutions are becoming the standard in many large organizations in which carefully thought out, unified software architectures are used to address business problems in levels of complexity that range from the operational to upper management and strategic levels.

When an organization decides to assess whether it should develop a software system, a feasibility study is usually carried out to compare costs to benefits. Based on evaluating the appropriate organizational criteria and financial metrics, managers can decide whether to move affirmatively towards selecting an information system from among various alternative options. Organizations look at software as a tool that can make their businesses better, their customers happier, and their shareholders wealthier. Three criteria used in recent research on assessing business value for IT-based systems are productivity, business profitability, and consumer surplus (Hitt & Brynjolfsson 1996 and 1996).

However, when a software system is being developed, the effective business value that it adds to the business performance of an organization tends to be neither explicitly addressed nor adequately quantified. In general, the focus in software development is generally on technical metrics intended to assure the quality of the software product, mainly in terms of its reliability characteristics. This is because software value is typically measured in terms of its intangible rather than tangible benefits on business. If a software system is reliable and robust, is tested, and can be maintained efficiently, it is assumed that it has a business value regardless of the resultant business outcomes. The overall business effect on value is rarely considered, nor is the distance between the potential value of a system and its realized value (Davern & Kauffman 2000).

Requirements validation is also an important metric when building software systems. However, the traditional forms of requirements focus on direct users' needs and overlook business value in terms of comprehensive and quantifiable measurements. Although project management and fiscally driven factors are part of the software engineering process, they are often not integrated well into the process. Moreover, a gap remains between the discipline of management information systems and the software development disciplines: MIS looks at solutions from a managerial perspective, but technical concerns are more influential for software development. The direct connection between software development and business performance is inadequate and is not well quantified or recognized as a core of measures: general measures and e-measures. The arrows in Figure 12.1 are bidirectional because they reflect the mutual influences between the initial two variables of this framework. Business goals should be triggered to guide an optimal software development process. Thus, this framework represents a view of the initial impact of business metrics on the development process.

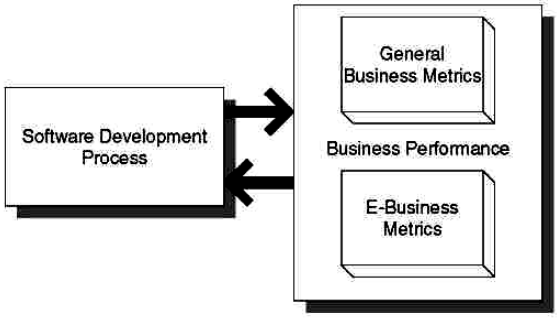


Fig. 12.1: Business View of Software Technology

The effect of the development process on business performance is also a key concern. Although many problem-solving strategies are used in software process modeling, the overall software process can be viewed in terms of certain basic elements or resources, such as activities, time, people, technology, and money. To reduce costs or increase benefits, one can think of combining activities, minimizing the cycle time, reducing the number of staff involved, maximizing profit, restructuring the composition of capital and finance, managing risk, or utilizing more technology. When the software process is reconsidered in these terms, business performance and metrics become the decisive driving force for building software process models.

Consequently, the software process has two related roles. The first role is internal: to assure software project payoff with better return on the information system investment, as discussed earlier. The second is external: the software process should make an actual difference in business performance. The first role has been addressed extensively in the software development and project management literature. However, few research efforts have been dedicated to the study of the external impact of the software process on business performance. In fact, these roles should always be combined because external impacts cannot be studied without considering internal impacts. Figure 12.2 depicts this dual approach.

This view represents the integration of the process and project themes and describes the evolution of software process models over the last several decades. Business value has always been embedded implicitly or explicitly in almost every progress in software process modeling. Minimization of time was behind the Rapid Application Development (RAD) and prototyping

models. Risk control and reduction were major issues behind spiral models. The efficient use of human resources lies behind the dynamic models. The impact of user involvement in software process models reflects the importance of customer influence. Achieving competitive advantage in software systems is a key business value related to users and customers. However, little empirical examination of the effect of the different problem solving strategies adopted in software process models takes place.

The interdependencies between the software process and business performance must be a key issue. The former is driven by the need for business value, and the latter in turn depends more than ever on software.

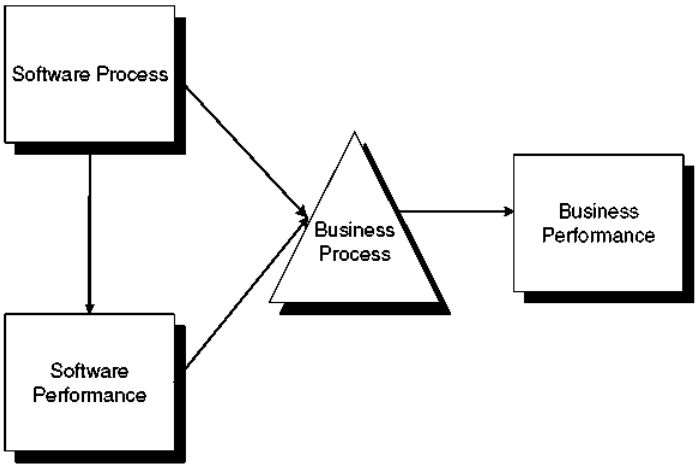


Fig. 12.2: Dual roles of the software process

This encompasses users, analysts, project managers, software engineers, customers, programmers, and other stakeholders. Computer systems are human inventions and do not function or interact without human input. Some manifestations of this dependency are:

- Software applications are produced by people and are based on people needs.
- Software applications that do not create value will not survive in the marketplace.
- Computers cannot elastically adjust to real situations (they work with pre-existing code and prescribed user inputs).
- Computers do not think in terms of expertise, they reflect if–then inputs or stored knowledge-based experiences.

- The main goal of software technology is to solve the problems of people.

This dependency on the human environment makes the automation that computers facilitate meaningless without human involvement and underscores the limits of computer systems. It also highlights the central role that people play in making software technology an effective tool for producing desired outcomes.

12.4 A View of Problem Solving and Software Engineering

Earlier sections presented a view of problem solving utilizing software technology and the impact of global problem-solving strategies on software-driven problem-solving strategies. They illustrated how global problem solving can apply a software-driven approach to enhance the efficiency of problem solving. The effectiveness of these approaches on business performance in terms of the business value created and software project optimization achieved was projected. Business value and project performance metrics were used to guide and reengineer the software-driven process modeling and the global problem-solving approaches.

This multidimensional, interactive, bidirectional view of global problem solving, software-driven problem solving, and business value is illustrated in the diagram in Figure 12.3. The software engineering literature has approached problem solving as a way of solving software problems. The view proposed here, as illustrated in this figure, uses an interdisciplinary approach to solving business problems in terms of software-driven technologies, tools, and capabilities. The objective is to create business value - a comprehensive and problem-solving approach. This view combines business, technology, and other relevant domains into an interdisciplinary framework for solving business problems.

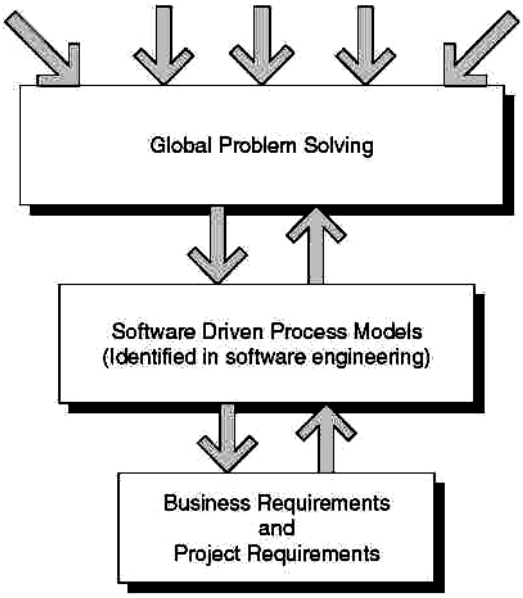


Fig. 12.3: A View of Problem Solving

Self Assessment Questions

4. Minimization of time was behind the _____ and prototyping models.
5. _____ and project performance metrics were used to guide and reengineer the software-driven process modeling and the global problem-solving approaches.

12.5 Summary

The unit discussed about what the computers can do to enable software technology to enhance the business model and also as to what the computers cannot do. The unit later discussed about the likely problems that business models might come across and how to optimally solve those issues with the help of software technology. Finally, an overview of the various problem solving methodologies were discussed.

12.6 Terminal Questions

1. Explain Software Technology in the context of Enabling Business Tool.
2. Explain Software Technology in the context of Limited Business Tool.
3. Give the two roles of the software process.

12.7 Answers

Self Assessment Questions

1. 18 months
2. Square
3. E-business
4. Rapid Application Development (RAD)
5. Business value

Terminal Questions

1. The application of software technology to problem solving exhibits characteristics that are fundamental to business organizations. Software technology allows for the acceleration of the problem-solving process by automating tasks and reducing the need for repetition. (Refer section 12.2)
2. Software technology enables business to solve problems more efficiently. However, as with any tool, it has its limitations. Solving business problems involves many considerations that transcend hardware or software capabilities. Thus, software solutions can only become effective when they are placed in the context of a more general problem-solving strategy. (Refer section 12.3)
3. The first role is internal: to assure software project payoff with better return on the information system investment, as discussed earlier. The second is external: the software process should make an actual difference in business performance. The first role has been addressed extensively in the software development and project management literature. However, few research efforts have been dedicated to the study of the external impact of the software process on business performance. (Refer section 12.4)

Unit 13 Diversification of Problem-Solving Strategies in Software Engineering

Structure:

13.1 Introduction

This unit examines factors that have promoted the diversification of software process models. The intention is to understand more clearly the problem-solving process in software engineering and to identify criteria that can be used to evaluate alternative software-driven problem-solving strategies for differing project requirements. A review of software process modeling is given first, followed by a discussion of process evaluation techniques. A taxonomy for categorizing process models, based on establishing decision criteria, is identified that can guide selecting the appropriate model from a set of alternatives on the basis of model characteristics and software project needs. These criteria can facilitate adaptability in the software process so that the process can be “altered or adapted to suit a set of special needs or purposes” (Basili & Rombach 1977).

The factors that have contributed to the diversification of software process models have often been related to the expansion in goals and capabilities in the software industry.

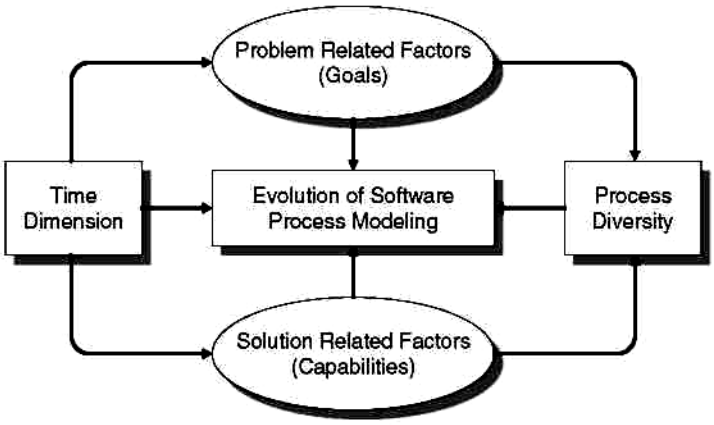


Fig. 13.1: Evolution of Software Process Modeling

Objectives:

After studying this unit, you should be able to:

- explain the diversification in software engineering
- discuss driving forces of diversity in development strategies
- describe the hidden values of differences
- explore the factors that affect interdisciplinary ignorance
- list out diversity in problem solver skills at the project management level

13.2 Understanding Diversification in Software Engineering

At the problem level, the roots of diversification include:

- Scope and complexity of problems
- Types of requirements and forms of problems
- Need to learn and apply new capabilities
- Challenges of continuous change
- Impact of the consumer economy and interdisciplinary effects
- Development of e-business applications
- Multiplicity of stakeholders, project team skills, background requirements, and business goals.

At the solution level, diversity has been driven by variations in:

- Project management approaches
- General standards
- Quality-assurance standards

- Hardware and software tools
- Networking tools
- Data mining and automation tools
- Nature, scope, and domain of applications
- Need for business-driven software engineering
- Secure software engineering
- “Killer” applications
- Mobile or wireless software engineering

13.2.1 Driving Forces of Diversity in Development Strategies

Diversity is a prevalent characteristic of the software process modeling literature. This reflects the evolution in software development in response to changes in business requirements, technological capabilities, methodologies, and developer experience. Process diversity also reflects the changing dimensions of project requirements, with process models maturing over time in their ability to address evolving project requirements. Diversification is also driven by the increasing importance of interdisciplinary views in modeling software processes. Figure 13.2 describes the combined effect of such temporal and interdisciplinary effects.

The temporal parameter is correlated with greater demands and changes that require ongoing adaptation and increased complexity. Time also introduces greater capabilities that afford better problem analysis and Technological capabilities seem to have the most influence on process modeling in terms of their impact on process automation, visualization, and degree of process control. Thus, although early process models were manual and sequential in structure, this changed with the introduction of fourth-generation techniques and languages. Process technology enabled the support of the rapid application development needed for iterative approaches with their greater emphasis on risk minimization and user satisfaction.

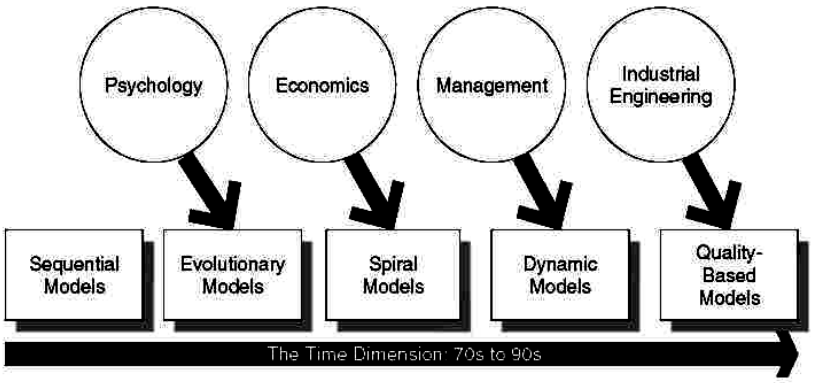


Fig. 13.2: Driving Forces of Diversity in Development Strategies

Time also increases the accumulated pool of experience in process modeling development. The movement from the traditional waterfall to the V-shaped model or from the conventional spiral to the win–win spiral model over the decades is examples of the effect of accumulated experience on process modeling structure and definition. This capability measure is also a function of problem-related factors, with increases in problem complexity and business requirements affecting the collective pool of experience and altering how problems were solved.

The type of methodology adopted also has considerable impact on process modeling evolution. For instance, an object-oriented methodology supports the architecture-centric approach in rational unified process models in terms of structure, automation, and visualization, as distinguished from process-oriented methodologies. Although these two methodologies exhibit generic conceptual similarities in the earlier phases of the process model, they become more differentiated as implementation-related factors are considered or techniques and representational constructs are utilized (Agarwal, De, & Sinha 1999). The SOFL model of Liu and colleagues (1997) presents an integrated approach that adopts structured methodologies in the requirements phases and object-oriented methodologies in the design and implementation phases. The adopted methodology can be driven by quality assurance and associated with the evaluation of software systems. Gradual improvement approaches such as TQM view problems differently than highly dynamic approaches such as BPR (business resource planning). For gradual improvement, SEI-CMM, the Kaizen approach, QIP, and the

BUTD approach have been introduced with significant effects on structuring and automating the development process (Bandinelli et al. 1995).

The software field originated with little attention paid to human factors. The importance of social context disciplines was only later appreciated, driven particularly by the increasingly widespread awareness of the high failure rate of software projects and its relation, at least in part, to social science-related factors. At that point, human factors began to be accommodated more seriously – for example, through the use of systems dynamics modeling and greater attention to cognitive effects and behavioral models. This reflected a more interdisciplinary understanding of software problem solving (Boehm, 1974).

Economic considerations were more systematically addressed by incorporating risk management in the prototyping, spiral, and other iterative process models. They were manifested in the development process with increased attention to feasibility assessment, cost estimation, risk assessment, productivity, and control. Industrial engineering and operations research are examples of other interdisciplinary influences affecting the evolution of process modeling. The application of quality-assurance standards to business processes is one example. Software modeling, in terms of development structure and process visualization, has also been affected by the increasing impact of customers on business. Thus, iterative structures substantially escalate user involvement and customer–developer communication becomes more effective with greater visualization. Thus, customer considerations have significantly affected process evolution. However, it is worth noting that working with small systems entails a different experience than working with large systems because modularization is not reliable without tailored approaches (DeRemer & Kron 1976). A schematic representation of drivers for the evolution of software process modeling is shown in Figure 13.3.

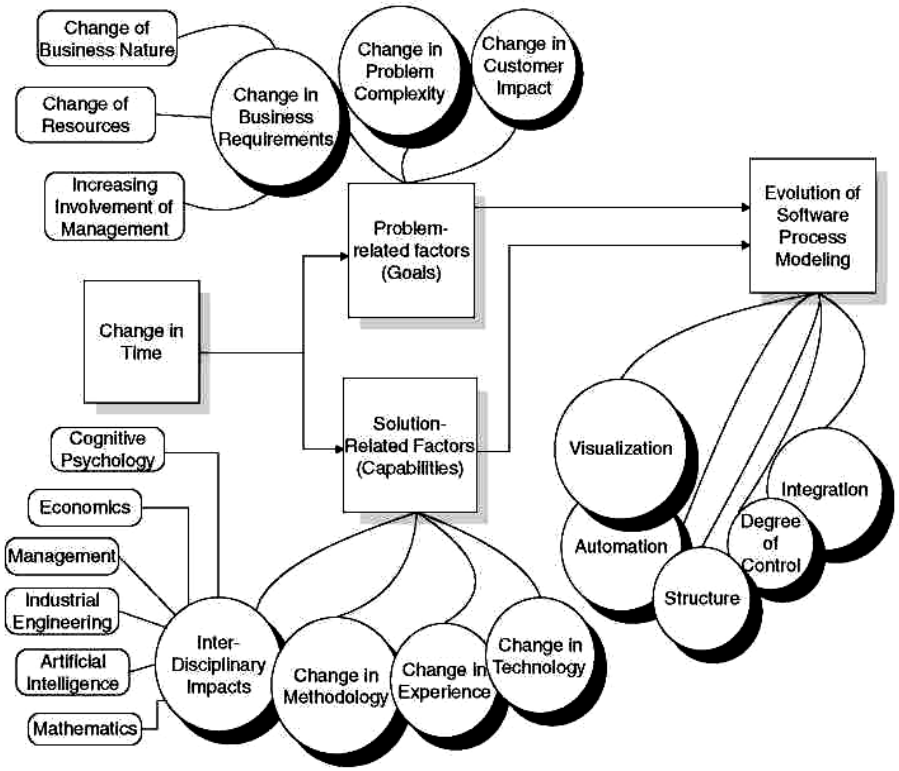


Fig. 13.3: Drivers for the evolution of software process model

Several implications are worth noting here. For one, it is clear that the arrow of time is critically correlated with advances in software process modeling. Indeed, most of the influential drivers in process modeling evolution are time dependent, although time is inadequate to explain all the variation. Time can be thought of as a necessary requirement for problem and solution-related drivers, acting as a trigger and a constraint. Although problem-related factors have been essential to precipitating changes, the availability of resources and capabilities (solution-related drivers) have had even greater impact on this evolution. This can be attributed to the impact of capabilities on problem-related factors. Thus, problem-and solution-related factors are not mutually exclusive, but depend on one other. The degree of automation, control, and integration and the extent to which changes in process structure take place can be used as measures of the evolution of software process modeling.

Another consideration has been the increasing degree of visualization provided for process models. Initial models, like the Waterfall, Evolutionary, and Spiral models, had a static view of the software development process, but later behavioral models explicitly portrayed the dynamic character of real-world software development processes. Indeed, with process improvement models and state-of-the-art advances in CASE tool technology, one is now able to monitor the development process in a multidimensional view, including full simulation of the dynamic behavior of the process. This advances the goal of efficiently controlling the software process. Figure 13.4 describes this evolution in visualization capability.

13.3 The Hidden Value of Differences

Paradoxically, diversity can be acquired through inheritance as well as by overriding the presuppositions that derive from inheritance. Cultural differences are examples of inherited characteristics that affect the degree of diversification in an environment. Scientific, social, political, psychological, philosophical, experiential, and other differences modulate acquired diversity through exposure to values, education, involvement, and interaction. Amid such diversity, commonly shared human needs play a unifying role. Conventional problem solving addresses problems by trying to eliminate the sources of contradiction; integrative problem-solving approaches try to capitalize on differences to obtain optimized solutions. Conventional problem solving eliminates or minimizes *the other* (the difference) in favor of specialization, cutting-edge problem solving incorporates (integrates) *the other* by inclusion.

Obviously, not every kind of difference can or should become a factor in a problem-solving strategy. Some differences may reflect contradictory facts or disputes about fundamentals, some of which may be irrelevant to the issue at hand. However, the idea of integrating differences rather than removing them is worthwhile if the legitimacy and relevance of the differences have been established. The key to distinguishing between negative differences (which ought to be excluded) and positive differences (which ought to be integrated) is to determine whether the differences are valuable and relevant. If they are, they should be utilized, not ignored or eliminated. Many modalities affect the status or interpretation of differences, for example:

- *The simultaneity factor.* Some differences can appear contradictory when they occur simultaneously, but are actually complementary when placed in sequential order on a timeline. For example, consider a false dichotomy such as analysis or design, process or architecture is more important in software development. Of course, when analysis and design are viewed as phases in a unified life cycle, each one is as important as the other. A business firm needs to diagnose a problem before providing architecture for its solution, and architecture needs to be tailored to a particular case. On the other hand, a good analysis is worthless if it is followed by a poor design.
- *The unique answer factor.* Differences can appear contradictory if only one element of a situation is taken as representative of the entire situation. This leaves no room for other contributing factors and no way to find relationships between diverse differences. For example, is a problem a technical or a business problem? Recognizing that a situation may arise from business as well as technical errors is totally different from understanding the issue from only a single perspective. Different elements can contribute to a complete picture and they may interact with or complement each other. Thus, a technical problem may affect business factors and business factors may create technical problems. The failure of a commercial Website to generate revenue may have been caused by inadequate technical support, which led to frustrated customers. A lack of appropriate budgeting may in turn have been responsible for the shortfall in technical support.

Self Assessment Questions

1 _____ is a prevalent characteristic of the software process modeling literature.

2 _____ also increases the accumulated pool of experience in process modeling development.

3 A lack of appropriate _____ may in turn have been responsible for the shortfall in technical support.

13.4 Integration – Not Differentiation

What is really needed in solving a problem is to find out whether the relevant differences or diversities can or should be made to work together? The purpose in integrating differences is not only to ensure resolution of

contradictory or conflicting factors. Indeed, diverse elements may not even be able to function independently of one another, and eliminating one element in favor of another may introduce other problems. To illustrate the integration of differences, consider another false dichotomy posed by the following question: “Which is more important: the process or the project?” This is a misguided alternative because it implies differentiation is the only choice and that integration is out of the question. In fact, no process exists without a project and no project can have a successful outcome without the guidance provided by a systematic problem-solving process. Thus, the project and the process must be integrated, combined, or synthesized – not differentiated in an exclusionary sense by sacrificing one element for the other.

In problem solving, it is tactically unwise to give priority to differentiation over integration because this tends to predispose developers to ignore or postpone examining the relationships among differences until they are compelled to do so by a roadblock in the solution effort. If differentiation is done first, a roadblock may occur after initial progress has been made in solving a problem when a difficulty related to some defect in the tentative solution is recognized. In this case, the process will be forced to backtrack, retracing its steps to determine what went wrong. By contrast, if one examines the potential benefit of integrating differences before selecting one of the apparent “alternatives,” the risk can be reduced. Thus, a *differentiation-first* approach is more likely to entail a costly restructuring of an entire effort in order to debug and correct a faulty process, but an *integration-first approach* may require only a preliminary inquiry and relatively primitive tests to evaluate the potential benefits of integration.

13.4.1 Investing in Diversification

Diversity is an organizational asset. It embodies the *hidden value* of differences: a value that is frequently underestimated, underutilized, or obscured in traditional approaches. Appreciating diversity is the only way in which one can successfully implement interdisciplinary thinking in software engineering. The purpose of investing in diversity is ultimately to exploit and incorporate the interdisciplinary knowledge that it represents into a unified problem-solving framework. Diversity investment leads to a wider understanding of the role of diversity in software engineering and bringing it to bear on issues identified during the problem-solving process. It also

implies identifying new, unrecognized, or underutilized areas of knowledge and exploring new aspects of problem definition.

One venue for doing this is by incorporating diverse requirements and capabilities into problem solving so that it is tailored to various kinds of business problems and project goals. For example, investment in diversity can be implemented by establishing training programs that prepare employees to think in an interdisciplinary way, to understand diversity, and to learn to incorporate diverse sources and types of knowledge to construct a broad-based approach to problem solving.

13.4.2 Factors that affect Interdisciplinary Ignorance

For present purposes, the term *ignorance* refers to a lack of data or the presence of inaccurate data in a circumstance in which such a lack hinders the proper understanding and definition of business and human problems. Ignorance in this sense includes lack of knowledge about available information as well as about adequate or effective tools. This results in a problem-solving process that may have unreliable or insufficient inputs. Understanding the sources and varieties of ignorance can help reduce the failure rate in problem-solving processes. Just as in the case of domain knowledge, domain or process ignorance is also an interdisciplinary phenomenon. Thus, overcoming this kind of ignorance requires an inter-disciplinary response. Although a thorough grasp of a problem area and the solution domain results in success, ignorance masks or obscures the real situation and thus broadens the distance between actual problems and their appropriate solutions. The many sources of ignorance include unreliable sources of information, partial knowledge, lack of communication, and inter-organizational ignorance.

1) Unreliable Sources of Information

This category includes inadequately accountable sources of information. Examples range from unconfirmed, inconsistent, suspicious, or doubtful resources to resources that are untrustworthy or lack qualification. Clearly, determining whether a resource is reliable requires examining the quality and credibility of the data and the data carrier. Even computerized systems can be based on incorrect formulas, programming bugs, and inaccurate entries. Interdisciplinary capabilities are needed to eliminate or disqualify unreliable resources and to rate or rank sources, which can be human,

digital, or hardcopy sources. For example, one can estimate the reliability of a human source by examining characteristics of subjects such as their skills, psychology, physiological criteria, etc. Technical testing may be required if data is delivered by electronic media. If a source involves specialized information, domain knowledge and expertise in the area may be needed to evaluate its reliability.

2) Partial Knowledge

This refers to aspects of an issue that have not been revealed (so-called in-breadth ignorance) or information about a specific aspect of an issue that is left incomplete (so-called in-depth ignorance). This type of ignorance may even be derived from a complacent or self-satisfied attitude – "what we do not know does not exist."

In-breadth ignorance assumes that information can be gathered using only one or two paths of knowledge, with other aspects of the problem not even considered for relevancy. Failure to recognize all the dimensions of an issue can result in solving the wrong problem and thus leaving the real problem unsolved. For example, although the infamous Y2K problem was at one level a technical problem, it had in fact many managerial aspects. For example, solving the technical dimension of Y2K was arguably easier than finding sufficient staff capable of reviewing systems for relevant bugs. In this situation, because of the intense demand for qualified staff, managing the available human resources became a real challenge. The "technical" problem was indeed interdisciplinary, like most business problems.

In-depth ignorance may recognize the relevant aspects of an issue but not study them thoroughly enough to understand them effectively. For example, when considering the e-business readiness of a certain organization, a company may be deemed well prepared in terms of Web presence, design, and infrastructure, but may have overlooked the need to train and prepare its staff for the demands of e-business. Staff training is a key ingredient of e-business readiness – at least as critical as technical skills, written policies, or strategies. E-business needs to begin with solid technical preparation, but in the long run it requires sufficient staff support, involvement, and understanding. In-depth coverage means that each dimension or component of an issue is studied and analyzed fully.

3) Lack of Communication

Lack of communication is a major source of ignorance. Communication narrows the distance between the various elements of the problem in question. Lack of communication originates in factors such as failure to contact the stakeholders in a business problem, not using effective communication techniques, or not being able to carry out an efficient communication process. The effects of a lack of communication can be summarized as follows:

- **Ignorance of lack of sources***.* Communication is the primary method for acquiring data from existing or prospective sources. Lack of communication reduces or omits sources of information.
- **Extra contextual ignorance.** Communication can ease tension between conflicting parties and improve common understanding. This is beneficial when gathering reliable data. Furthermore, the more that data resides outside an organizational context, the more difficult it is to obtain. Communication encourages an amicable and mutually accessible environment in which differences can be viewed as sources of data and knowledge. This also creates opportunities for transferring and exchanging data.
- **Ignorance of lack of communication channels.** Without appropriate communication channels, it is often difficult to deliver timely or on-time data. Late data delivery can make the problem-solving process less effective. This is especially important in achieving competitive advantage and responding to urgent situations.
- **Differentiation ignorance.** The current trend in business is to learn from competitors and to seek partnerships to achieve common goals. It is known that integrative approaches facilitate more effective problem solving roles in terms of gathering reliable data, compared to non-integrative, differentiating approaches. Communication is the cornerstone for facilitating any integrative process.

4) Inter-organizational Ignorance

The value of knowledge stems from its usability and adaptability, not from its mere existence. To be valuable, information or data must add value to an organization and to its problem-solving processes. Otherwise, it is tantamount to a form of double ignorance in which people do not know what

they know but assume that they do (or, they do not know that they do not know). This can make knowledge expensive if one is in possession of unused data, or make an organization a victim of knowledge utilization delays that result from a lack of awareness or ignorance of ignorance. Knowledge-based ignorance can hide weakness behind apparent strength and business sickness behind an apparently healthy organization. This source of ignorance has many manifestations and degrees and even low levels can be damaging and costly.

Consider, for example, the sales transactions that a department store conducts with its customers on a daily basis. If this accumulated daily data is only stored until the end of the year and then used solely for purposes related to taxes and inventory, the opportunity to apply such critical information may have been permanently lost. For example, applied in a timely fashion, the daily data could have been utilized for a variety of purposes – including tracking inventory in order to avoid going below a repurchase point. If data is not processed on time for such tracking purposes, business sales can suffer because of out-of-stock occurrences on key saleable items, possibly resulting in a loss of strategic clients, alliances, or business partners. Ignorance at the inventory level can block a business from operating, partially or totally in a very short time. Therefore, even though this type of ignorance is associated with a low level of the structured business process, lack of use has a potential for major impact and so represents a serious risk.

Studying customer behavior in a manner that measures customer requirements on an accurate, predictive basis is another example of the applicability of such low-level data. Without analyzing daily sales data statistically, it may be impossible to cluster customers, products, or sales points so that the store can prosper and maintain its competitive advantage. Ignorance at the customer satisfaction level may not preclude a business from continuing operation, but it may put such a business at a competitive disadvantage. The level of *risk of ignorance* in this situation may be moderate, but the long-term effects may be critical. This situation belongs to the branch-level management class of business processes.

Conducting ongoing cost-benefit analysis to measure financial performance and to control share profit is also an important issue. Absence of knowledge

critical to supporting decision-making processes may prevent an organization from effectively supporting strategic management decisions. Such knowledge is of strategic value and can only be derived from daily transactional data. A lack of knowledge of what is happening on a given day may be minimal in terms of risk. However, in the long term, this may mask critical risk factors lurking behind the scene that can lead to business failure.

Although many levels of ignorance are linked simply to lack of data, information, or knowledge, some ignorance can be attributed to vague, surface, or unused knowledge. Examples include:

- *Unprocessed data.* Data that is not transformed into useful information in the right form, at the right time, and provided to the right people represents unprocessed data. Unprocessed data makes what we know less effective, but still expensive. Many organizations are excellent at gathering data, but fail to relate it to their problems because they do not convert it to other, more meaningful forms of information or knowledge.
- *Unused data.* When data is not used to solve problems, it amounts to an absence of data. Unused data, regardless of its level of transformation or meaningfulness, merely represents an added cost created by careless business behavior. If this is related to data that has not been processed, it is a waste of time and money. If it is related to processed data known to be useful, then retaining this data without further examination or resolution is a problem and contributes to wasted time and resources. If data is unused due to lack of managerial commitment and despite the established value of the data, this transcends mere normal ignorance and rises to the level of culpable ignorance. Deliberate or culpable ignorance represents a type of business malfeasance.
- *Untailored data.* Utilizing data effectively requires an accurate problem definition, just as medication makes no sense without a proper prior diagnosis. Thus, understanding the problem and the solution domain is as important as knowledge of the data.
- *Vague data.* Data may be too low quality to be considered for processing. This is a case of ignorance of the data that one has. Such data may be uncertain, unconfirmed, unclear, or undefined, or need proper translation or adequate clarification. If the data is processed

despite its errors or uncertainties, unreliable outcomes and inefficiencies in decision-making result.

- *Politically based ignorance.* Organizational politics can play a destructive role in obtaining reliable data. If nonscientific, non-rational, or biased motivations are behind the selection of data, this may preclude obtaining critical data. Such politically selected data cannot be considered representative. The excluded data may contain contradicting facts, clarifying statistics, or a more complete picture. Intentional and biased ignorance of this type affects the problem-solving process negatively. There must be a legitimate and objective reason, not based on political or economic interest, to justify exclusion of data. Biases are another kind of filter blocking accurate data acquisition. They represent a kind of color-blindness in viewing facts in which the interpretation of the data depends on whether it supports a position held in advance. This attitude inhibits seeing other viewpoints merely because they are the viewpoints of others.
- *Technically based ignorance.* This refers to the lack of reliable tools that enable us to see, understand, and interpret phenomena correctly. Ignorance is strongly tied to such lack of tool support. One cannot be expected to make sense of data without reliable tools. When tools are unavailable, one should anticipate that the data may not be processed at all, may not be processed on time, may not be processed accurately, may be lost or destroyed due to lack of storage tools; may be used only at lower levels of management, or may not be recognized as enabling decision-making processes.
- *Statistically based ignorance.* This refers to a failure to establish the right relationship between things in an interconnected environment. Ignorance is often not so much a failure to collect data, as a failure to explore the data or see how data is interconnected. For example, a change in organizational effectiveness that occurs in parallel with a newly adopted style of management may not be coincidental. Viewing data as isolated bits of information without making the effort to observe its correlations or interrelationships is a type of ignorance. The effectiveness of the problem-solving process strongly depends on the ability to observe, discover, or predict relationships between variables in an organizational context.

- *Illusion-based ignorance.* Data is not always transparent. It may mask deception, illusion, imagination, or tricks that create false impressions. Major national corporations have gone belly-up as the result of this kind of ignorance. In order to distinguish facts from illusions, caution must be exercised when viewing available data. Figure 13.4 illustrates factors that influence interdisciplinary ignorance.

13.5 Diversity in Problem Solver Skills at the Project Management Level

Little empirical evidence is available about the skills required for a project manager to be successful or how the training or experience of managers affects the success of the projects that they supervise. However, there does seem to be a consensus on the kinds of skills required at a very high level, with technical competence a "given" for project managers. Naturally, however, technical competence alone is not enough to be a successful project manager.

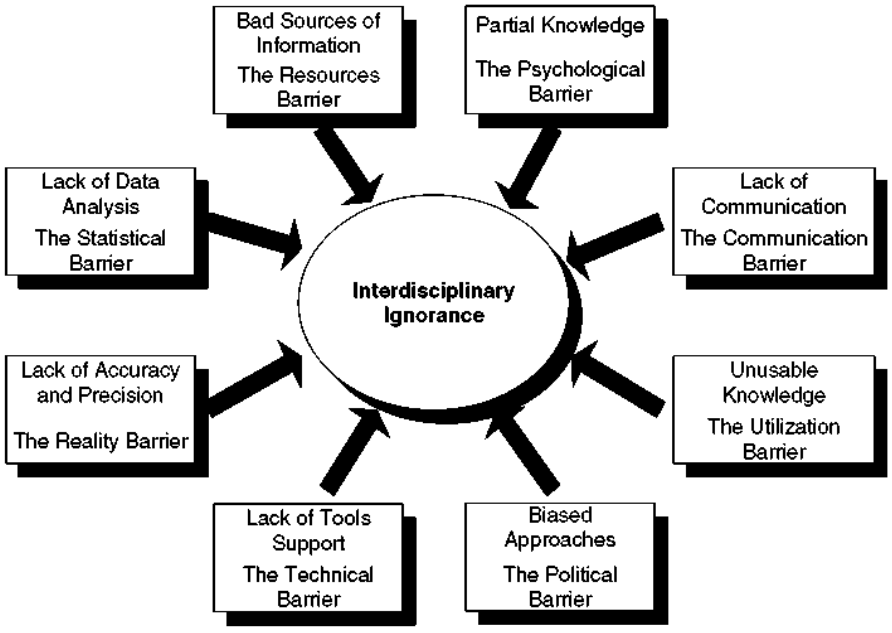


Fig. 13.4: Diversity in Problem Solver Skills at the Project Management Level

In addition, it is recognized that project managers also need various soft skills, including managerial and leadership abilities, relationship and

interpersonal skills, and negotiation and sales skills (Moore 1996; Pressman 1996, 1997; Tatnall & Shackleton 1996; Bach 1997; Phillips 1999; Haggerty). This includes the different parts of a whole, different views of an object or issue, different dimensions of a structure, and different steps of a process or procedure. These types of differences can be used to add value and can be reproduced, reused, or instantiated to further enhance added value.

Table 13.1: Views of Diversity

Type of Difference Deployment	*Representation*	*Explanation*
Difference activation	X – Y = Z	Diversity seen as potential problem
Difference neutralization	X + Y = X + Y	Diversity tolerated but not capitalized on
Difference optimization	X + Y = Z	Diversity properly deployed to create value

- *Interactive differences* (diversity of relationships) refers to elements that add value because of how they interact with one another.
- *Differences of degree, level, or version* (diversity of inheritance) refers to elements that are apparently different, but really only reflect differences of degree or level, or represent versions under a more comprehensive super-class.

In the case of software engineering, the different components of software engineering knowledge (theory, methods, techniques) and the different software engineering practices that tailor knowledge to specific requirements can add value to the problem-solving process (output or TO-BE). Furthermore, diversified resources can also add value to software engineering knowledge and to best practices (input or AS-IS). Thus, diversification can affect the AS-IS and the TO-BE levels of problem engineering. Existing differences can be brought together at the AS-IS level. For instance, diversified findings with regard to an existing system can be integrated in a complementary or interactive fashion. New interdisciplinary resources can be incorporated to enhance the proper definition of existing business problems at the TO-BE level.

Self Assessment Questions

4 The term _____ refers to a lack of data or the presence of inaccurate data.

5 Viewing ___ as isolated bits of information without making the effort to observe its correlations or interrelationships is a type of ignorance.

13.6 Summary

The unit mainly deals with the diversification of software engineering in problem solving approaches. The factors affecting the development methodologies in terms of diversification and ignorance were then discussed. The problem solving skills necessary at the project management level were discussed towards the end of the unit.

13.7 Terminal Questions

1 What are the factors affecting the development methodologies in terms of diversification?

2 Give the importance of integration over differentiation in development strategies.

3 List out the skills required by a successful project manager.

13.8 Answers

Self Assessment Questions

1. Diversity
2. Time
3. Budgeting
4. Ignorance
5. Data

Terminal Questions

1. Diversity is a prevalent characteristic of the software process modeling literature. This reflects the evolution in software development in response to changes in business requirements, technological capabilities, methodologies, and developer experience. (Refer section 13.2.1)

2. What is really needed in solving a problem is to find out whether the relevant differences or diversities can or should be made to work together? The purpose in integrating differences is not only to ensure

resolution of contradictory or conflicting factors. Indeed, diverse elements may not even be able to function independently of one another, and eliminating one element in favor of another may introduce other problems. (Refer section 13.4)

3. The project managers also need various soft skills, including managerial and leadership abilities, relationship and interpersonal skills, and negotiation and sales skills together. This includes the different parts of a whole, different views of an object or issue, different dimensions of a structure, and different steps of a process or procedure. These types of differences can be used to add value and can be reproduced, reused, or instantiated to further enhance added value. (Refer section 13.5)

Unit 14 Case Study

Structure

14.1 Introduction

Business schools have been using case studies for years to develop a student's analytical abilities, but they are rarely seen in software engineering courses. Case studies can also be used to develop the analytical abilities of software engineering students. Furthermore, case studies can help bridge the gap between the experienced software engineer and the inexperience student who has difficulty applying what he or she has learned in the classroom. A carefully designed case study focuses the students on specific software development problems.

Objectives

After studying this case study and attempting the questions given at the end, the student should be in a position to:

- explain the intricacies of the analysis of the situation
- describe the right approach for analysis
- give the correct rationale behind choosing a specific solution

ACME Financial Incorporated (AF Inc.) is an investment banking company that provides an investment banking company that provides an on-line service that allows their clients to access account and market information. ACME Financial Inc. recently acquired several small and medium sized companies through-out the country, each with their own financial and accounting systems. Almost all of the companies have developed their own application software for their analyst' use in their daily jobs, but only a few provided on-line account service. The CIO wants to consolidate the financial and accounting information into a corporate information system that can support decision support applications fir corporate management. Naturally,

since the computer hardware is different for different companies, the CIO expects to upgrade the hardware to accommodate the new Information Technology (IT) system. The CIO will select the best analytical software as the standard software used by all company analysis. Each local site will be expected to provide an on-line service for their account information. Finally, ACME Financial has developed special data mining software that gives them a competitive advantage. AF Inc. offers customers investment advice based on the information derived by the data mining software. Each account manager receives the information and then provides tailored recommendations to each customer based on their portfolio.

14.2 System Requirements

The following list of system requirements reflects the system's relative priorities:

1. **Availability:** The CIO's number one priority is high AF Inc. markets their reliability and feels that most clients choose them for their dependability. The CIO wants to maximize the system's availability. To achieve high availability, if a regional office cannot provide support then a customer must always have access to the on-line service through a different office.
2. **Data Integrity:** The requirement for data integrity varied within the system. The most important data are customer's transactions. It is essential that a customer's transaction is never lost and the system must guarantee that each transaction is completed. In contrast, data lost from the high data rate inputs, such as Reuter's and the NYSE, are easily recovered in subsequent broadcasts so it is not critical if some data are lost during a broadcast.
3. **Performance:** Financial markets are highly volatile; time sensitivity of data is measured in minutes. Million can be lost if information broadcast throughout the network.
4. **Security:** The CIO is concerned about the security of the data mining software and the information produced by the data mining software. The Chief Executive Officer thinks the data mining information software provides a competitive advantage for the company. If an unauthorized user had access to the information they could steal the data mining applications or steal the information produced by the data mining

software. In either case, the perpetrator could make the same investment recommendations as AF Inc. account managers. Therefore, if competitors had access to the information the results could be financially devastating to the company. The CIO is concerned that a competitor could pose as a customer and hack into the highly sensitive information through his on-line service account.

5. **Growth:** The CIO envisions an incremental migration process to install the new system due to the magnitude of the change. Also, he expects that AF Inc. will continue to grow and acquire more companies. The CIO wants to be able to develop more application software as new customer services are added. The CIO also wants to add more near-real time information sources to the system.
6. **Backup and Recovery:** The CIO understands that the system will encounter problems from time to time. A key factor in determining the system's success is how quickly the system can recover from a failure. Backup and recovery must be smooth and non-disruptive. One way to ensure that the system can easily recover from a system crash is to make sure the data is duplicated elsewhere on the system. The corporate database is the primary backup for each of the regional offices.
7. Each local office (Northeast, Northwest, Southeast, and Southwest) has accesses a regional information hub. Local offices use client software to access the local application server. These application servers access the local database for almost all of the information needed on a daily basis. For access to information needed less frequently the application software should access the central database at corporate headquarters. Each regional database has only the subset of information that is relevant for its area, whereas the corporate headquarters maintains all of the information from each region as well as data that is unique to corporate applications, such as additional accounting and company financial information. The corporate office is also responsible for the data mining software and information. Each of the regional databases is connected with high capacity links to the corporate database. Finally, the corporate office receives information from Reuter's, NYSE, MASDAQ and other financial markets. The information flow fluctuates daily from 30 40 KBps to 4 5 MBps. Twenty-five percent of the information is

immediately broadcast to the regional offices to support the on-line account service. All the information is filtered and stored in the database.

14.3 Architectural Alternatives

Alternative I: The Database Management System. This alternative takes advantage of the extended functionality provided by the popular relational database management companies, such as Oracle and Sybase. All information is delivered into the system where it is immediately stored into one of the databases. The relational database management software is responsible for the distribution of information throughout the system. Clients communicate with the databases through Standard Query Language (SQL). Corporate and regional databases are kept synchronized using features supplied by the RDBMS software. Transactions are guaranteed by using special Transaction Processing Software. The vendor supplied RDBMS software is responsible for back-up and recovery of all the databases. Data security is handled at the row level within each database. This means that clients can only receive records for which their user has permission. Existing application software may have to be modifies to use SQL.

Alternative II: Common Object Request Broker Architecture (CORBA). This solution depends on CORBA to tie together the clients and databases. CORBA is responsible for distributing data across the system. The RDBMS software is still responsible for the backup and recovery, but the databases are kept synchronized using CORBA as the primary transport mechanism for the data. Clients, application serves, and databases communicate to each other through CORBAs transport mechanism. Existing application software would be wrapped in IDL to communicate with other applications. Special near-real time handling application software would send the information to each of the regional offices where it would be directed to clients that subscribe to the information.

Alternative III: Message and Queuing (M & Q). The message queuing design uses commercial M & Q software combined with a transaction processing product to ensure customers' transactions are completed. Dec Message Queue and MQ Series are some of the leading products for messaging and queuing software. Clients communicate to other entities using messages. Messages are deposited in queues and the message and

queuing middleware is responsible for message distribution to the appropriate clients. The software applications will be modifies to send and receive from queues.

14.4 Terminal Questions

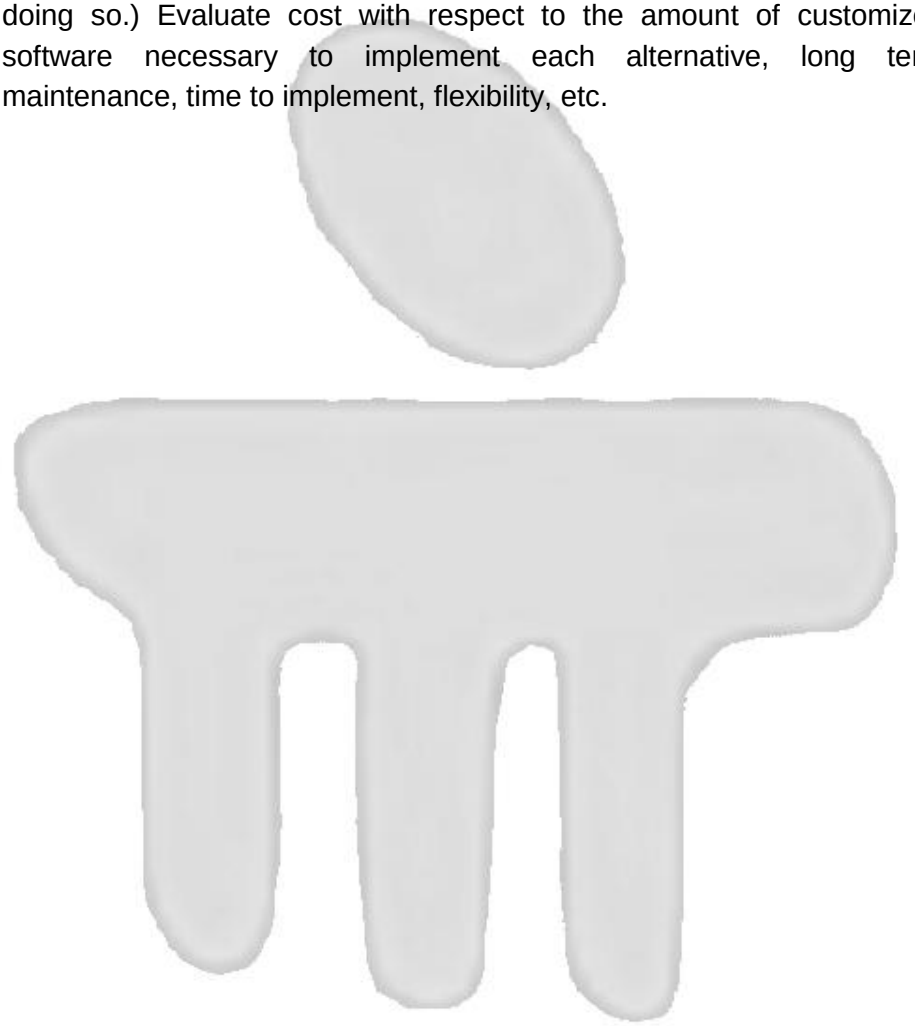
1. Describe in more detail the architecture of each alternative. Some services are automatically provided when a product is purchased, others must be developed to satisfy the system requirements. You should describe what services are automatically provided by some of the products, which services would need to be developed, and how services should be distributed across the network. (30 points)
2. Evaluate each of the alternatives against the system requirements. Discuss the advantages and disadvantages of each of the alternatives. Assume that the hardware will support all solutions. In your analysis which alternative provides easiest maintenance and which alternative requires the least modification to the current system. (30 points)
3. Prioritize each alternative or suggest a different solution if you think it superior to the 3 presented alternative. (20 points)

14.5 Answers

Suggestions on how to proceed (For questions 1 to 3 above)

1. There is not enough information to make an informed decision about each of the alternatives. As a team, allot a percentage of your time to discover which products offer what type of services. You do not have enough time to do a complete market survey so pick 2-3 products.
2. If you depend only on marketing information you may find that the alternatives are equivalent. So you might want to go beyond the market literature in doing your research for this assignment.
3. As you do your analysis pay particular attention to some of the following kinds of issues:
4. a. How well does the architecture support the basic system functionality requirements?
   b. How much run time performance overhead does the architecture impose?
   c. How well will specific products handle the high volume of data?
   d. How well each architecture handles occasional peak loads?
   e. How easy is it to customize the system to new requirements?

5. In your analysis, do not consider the actual product cost. (It may be impossible to get actual product costs anyway, so do not waste time doing so.) Evaluate cost with respect to the amount of customized software necessary to implement each alternative, long term maintenance, time to implement, flexibility, etc.



Acknowledgements, References and Suggested Readings:

1. Roger Pressman, “Software Engineering', McGraw Hill, Fifth Edition
2. Pankaj Jalote, "An Integrated Approach To Software Engineering", Narosa
3. W. S. Jawadekar, "Software Engineering”, TMH.
4. R. Mall, "Fundamentals of Software Engineering", Prentice Hall of India
5. Behferooz & F. J. Hudson, "Software Engineering Fundamentals", Oxford University Press
6. S. L. Pfleeger, "Software Engineering Theory and Practice", Pearson Education
7. James Peter, "Software Engineering An Engineering Approach”, John Wiley
8. Ian Sommerville, "Software Engineering', Pearson Education
9. Software Engineering – 1, D. Bjorner, Springer, 2005-06.
10. Software Engineering – 2, D. Bjorner, Springer, 2005-06.
11. Software Engineering – 3, D. Bjorner, Springer, 2005-06.
12. An Integrated approach to Software Engineering – 3rd Edition, Pankaj Jalote, Springer.
13. Roger Pressman, “Software Engineering', McGraw Hill, Fifth Edition.
14. S. L. Pfleeger, "Software Engineering Theory and Practice", Pearson Education.
15. James Peter, "Software Engineering An Engineering Approach”, John Wiley.