

**SOFTWARE
ENGINEERING
(DMCA201)
(MCA)**



ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

UNIT - I

Structure

1. INTRODUCTION TO SOFTWARE ENGINEERING AND GENERIC VIEW OF PROCESS

1.1 The Evolving Role of Software

1.2 Change Nature of Software

1.3 Software Myths

1.4 Layered Technology

1.5 Process Frame work

1.6 CMMI

1.7 Process Patterns

1.8 Process Assessment

1.9 Personal and Team Process Models

2. PROCESS MODELS

2.1 The Waterfall Model

2.2 Incremental Process Models

2.3 Evolutionary Process Models

2.4. The Unified Process

2.5. Functional Requirements

2.6. Non-functional Requirements

2.7. User requirements

2.8. System Requirements

2.9. Interface Specification

2.10 The software requirements Document

1. Introduction to Software Engineering and generic view of process

Objectives

After going through this unit, you should be able to:

- define software engineering;
- understand the evolution of software engineering;
- understand the software myths;
- learn about process frame work, patterns, and assessment;
- understand personal and team process models.
- learn about phases of software development life cycle models, and
- understand software development requirements.

1.1 The Evolving Role of Software:

The field of software engineering is related to the development of software. Large software needs systematic development unlike simple programs which can be developed in isolation and there may not be any systematic approach being followed.

In the last few decades, the computer industry has undergone revolutionary changes in hardware. That is, processor technology, memory technology, and integration of devices have changed very rapidly. As the software is required to maintain compatibility with hardware, the complexity of software also has changed much in the recent past. In 1970s, the programs were small, simple and executed on a simple uniprocessor system. The development of software for such systems was much easier. In the present situation, high speed multiprocessor systems are available and the software is required to be developed for the whole organisation. Naturally, the complexity of software has increased many folds. Thus, the need for the application of engineering techniques in their development is realized. The application of engineering approach to software development leads to the evolution of the area of Software Engineering. The IEEE glossary of software engineering terminology defines the Software Engineering as:

“(a) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, that is, the application of engineering to software. (b) The study of approaches in (a).”

There is a difference between programming and Software Engineering. Software Engineering includes activities like cost estimation, time estimation, designing, coding, documentation, maintenance, quality assurance, testing of software etc. whereas programming includes only the coding part. Thus, it can be said that programming activity is only a subset of software development activities. The above mentioned features are essential features of software.

Besides these essential features, additional features like reliability, future expansion, software reuse etc. are also considered. Reliability is of utmost importance in real time systems like flight control, medical applications etc.

Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is an information transformer producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

Software delivers the most important product of our time information. Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., Inter-net) 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 architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. Sophistication and complexity can produce dazzling results when a system succeeds, but they can also pose huge problems for those who must build complex systems.

Popular books published during the 1970s and 1980s provide useful historical insight into the changing perception of computers and software and their impact on our culture. Osborne characterized a "new industrial revolution." Toffler called the advent of microelectronics part of "the third wave of change" in human history, and Naisbitt predicted a transformation from an industrial society to an "information society." Feigenbaum and McCorduck suggested that information and knowledge (controlled by computers) would be the focal point for power in the twenty-first century, and Stoll argued that the "electronic community" created by networks and software was the key to knowledge interchange throughout the world.

As the 1990s began, Toffler described a "power shift" in which old power structures (governmental, educational, industrial, economic, and military) disintegrate as computers and software lead to a "democratization of knowledge." Yourdon worried that U.S. companies might lose their

competitive edge in software-related businesses and predicted “the decline and fall of the American programmer.”

Hammer and Champy argued that information technologies were to play a pivotal role in the “reengineering of the corporation.” During the mid-1990s, the pervasiveness of computers and software spawned a rash of books by “neo-Luddites” (e.g., *Resisting the Virtual Life*, edited by James Brook and Iain Boal and *The Future Does Not Compute* by Stephen Talbot). These authors demonized the computer, emphasizing legitimate concerns but ignoring the profound benefits that have already been realized.

During the later 1990s, Yourdon re-evaluated the prospects for the software professional and suggested the “the rise and resurrection” of the American programmer. As the Internet grew in importance, his change of heart proved to be correct. As the twentieth century closed, the focus shifted once more, this time to the impact of the Y2K “time bomb”.

Although the predictions of the Y2K doomsayers were incorrect, their popular writings drove home the pervasiveness of software in our lives. Today, “ubiquitous computing” has spawned a generation of information appliances that have broadband connectivity to the Web to provide “a blanket of connectedness over our homes, offices and motorways”. Software’s role continues to expand.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application. And yet, the same questions asked of the lone programmer are being asked when modern computer-based systems are built:

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all the errors before we give the software to customers?
- Why do we continue to have difficulty in measuring progress as software is being developed?

These, and many other questions, are a manifestation of the concern about software and the manner in which it is developed a concern that has led to the adoption of software engineering practice.

The latest trend in software engineering includes the concepts of software reliability, reusability, scalability etc. More and more importance is now given to the quality of the software product. Just as automobile companies try to develop good quality automobiles, software companies try to develop good

quality Software. The software creates the most valuable product of the present era, i.e., information.

The following Table summarises the evolution of software:

1960s	Infancy Machine Code
1970s Project Years	Higher Order Languages
1980s Project Years	Project Development
1990s Process and Production Era	Software Reuse

The problems arising in the development of software is termed as crisis. It includes the problems arising in the process of development of software rather than software functioning. Besides development, the problems may be present in the maintenance and handling of large volumes of software. Some of the common misunderstandings regarding software development are given below.

1. Correcting errors is easy. Though the changes in the software are possible, but, making changes in large software is extremely difficult task.
2. By proper development of software, it can function perfectly at first time. Though, theoretically, it seems correct, but practically software undergoes many development/coding/testing passes before becoming perfect for working.
3. Loose objective definition can be used at starting point. Once software is developed using loose objective, changing it for specific objectives may require complete change.
4. More manpower can be added to speed up the development. Software is developed by well coordinated teams. Any person joining it at a later stage may require extra efforts to understand the code.

Software Standards

Various terms related to software engineering are regularly standardized by organizations like IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), OMG (Object Management Group), CORBA (Common Object Request Broker Architecture).

IEEE regularly publishes software development standards.

OMG is international trade organisation ([http:// www.omg.org](http://www.omg.org)) and is one of the largest consortiums in the software industry. CORBA defines the standard capabilities that allow objects to interact with each other.

1.2. The changing Nature of Software:

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

A description of software might take the following form:

Software is (1) instructions (computer programs) that when executed provide desired function and performance,

(2) Data structures that enable the programs to adequately manipulate information, and

(3) Documents that describe the operation and use of the programs. There is no question that other, more complete definitions could be offered. But we need more than a formal definition.

Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and bread boarded prototype evolve into a physical product (chips, circuit boards, power supplies, etc.).

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

1. *Software is developed or engineered, it is not manufactured in the classical sense.*

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product" but the approaches are different.

Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. Software doesn't "wear out."

Figure 1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

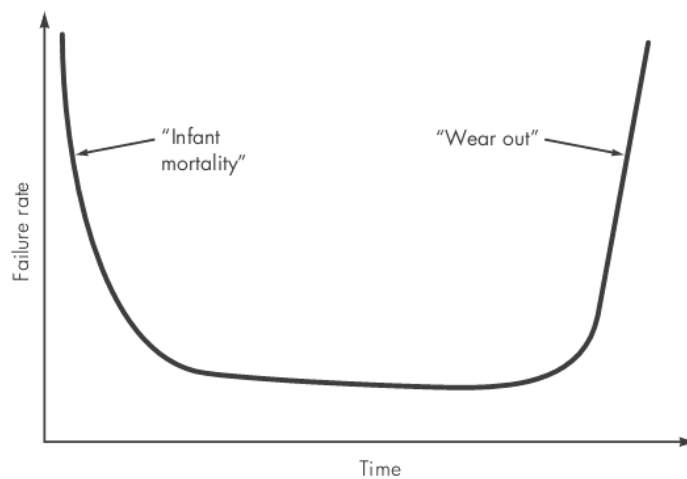


FIGURE 1: Failure curve for hardware

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross over-simplification of actual failure models for software.

However, the implication is clear software doesn't wear out. But it does deteriorate. This seeming contradiction can best be explained by considering the "actual curve" shown in Figure 2. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 2. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the

minimum failure rate level begins to rise the software is deteriorating due to change.

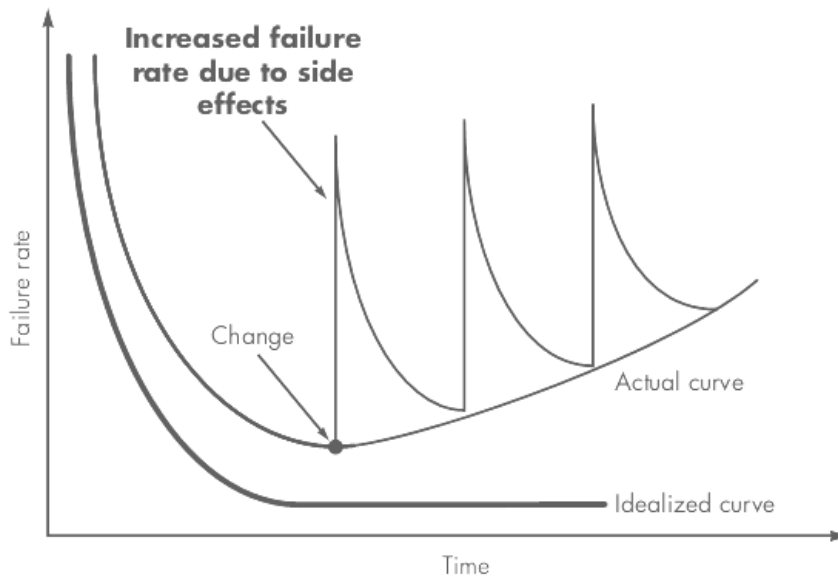


FIGURE 2. Idealized and actual failure curves for software

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3. Although the industry is moving toward component-based assembly, most software continues to be custom built.

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an IC or a chip) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a

natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well defined algorithms in an effective manner but had a limited domain of application. Today, we have extended our view of reuse to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

Software Applications:

Software may be applied in any situation for which a pre-specified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert system software and neural network software). Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information. For example, many business applications use highly structured input data (a database) and produce formatted "reports." Software that controls an automated machine (e.g., a numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.

Information determinacy refers to the predictability of the order and timing of information. An engineering analysis program accepts data that have a predefined order, executes the analysis algorithm(s) without interruption, and produces resultant data in report or graphical format. Such applications are determinate. A multiuser operating system, on the other hand, accepts inputs that have varied content and arbitrary timing, executes algorithms that can be interrupted by external conditions, and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate.

It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows, neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

System software: It is a collection of program written to service other programs. It is characterized by heavy interaction with computer hardware,

heavy wage by multiple users, concurrent operation that requires scheduling, resource sharing & 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. Elements of real-time software include a data gathering component, an analysis component, a control/output component and a monitoring component.

Business software: Business information processing is the largest single software application area. It encompass interactive-computing in addition to conventional data processing application.

Engineering & Scientific software: They have 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: It resides in ROM and is used to control products and systems for the consumer and industrial markets. It can perform very limited and esoteric functions or provide significant function and control capability.

Personal computer software: A few of hundreds of applications include word process spread sheets, computer graphics multimedia, entertainment, database management, personal & business financial applications external software, data base access.

Web-based software: It not only provides standalone features, computing functions, and content to the end user, but also is integrated with corporate databases and business applications.

Artificial Intelligence software: It makes use of non-ensure algorithms to solve complex problems that are not amenable to computation or straight forward analyses. Different applications are Robotics, Expert systems, Pattern recognition, artificial neural networks, theorem proving and game playing.

1.3. Software Myths

Def: Beliefs about software and the process used to build it. Myths have a number of attributes. The most knowledgeable software Engineering professionals recognize myths problems for managers and technical people.

Different categories of myths are:

Management myths

Managers with software responsibility like managers in most disciplines are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: We already have a book that's full of standards and procedures for building software won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: My people have state-of-the-art software development tools, after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

Myth: If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks: "adding people to a late software project makes it "In the absence of meaningful standards, a new industry like software comes to depend instead on folklore." later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an

outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing programs we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. Figure 3 illustrates the impact of change. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

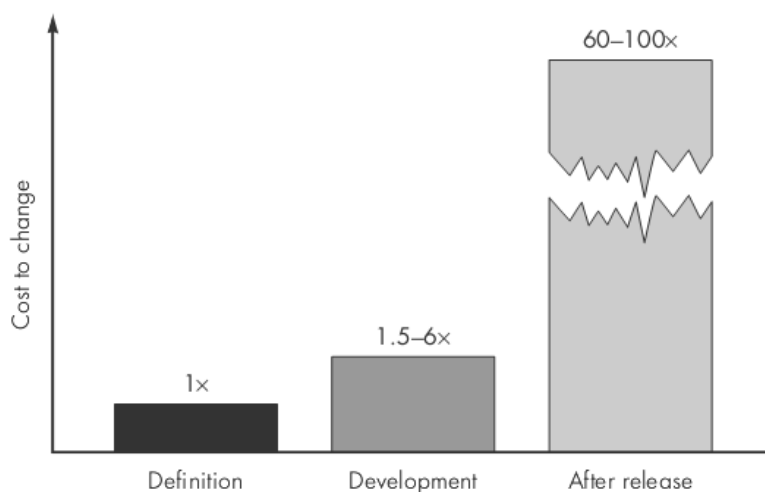


FIGURE 3. The impact of change

Practitioner's myths

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project the formal technical review. Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us creates voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.4. A Layered Technology

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

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

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

The IEEE has developed a more comprehensive definition when it states:

Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

According to Fritz Bauer, software engineering is the establishment & use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

Software engineering is a layered technology. It can be depicted from the fall.



Software engineering layers

Any engineering approach must rest on an organizational commitment to quality. Total quality management, six sigma and similar philosophies foster a continuous process improvement culture and this culture ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is Quality focus.

The foundation of software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software process defines a frame work that must be established for effective delivery of software engineering technology. It forms the basis for management control of software projects and establishes the context in which technical methods are

applied, work products are produced, milestones are established, quality is ensured and change is properly managed.

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

Software engineering tools provide automated a semi-automated support for the process and the methods. When tools are integrated so the information created by one tool can be uses by another, computer aided software engineering system is established for the support of software development.

1.5 Process Frame Work

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

In recent years, there has been a significant emphasis on “process maturity.” The Software Engineering Institute (SEI) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity. To determine an organization’s current state of process maturity, the SEI uses an assessment that results in a five point grading scheme. The grading scheme determines compliance with a capability maturity model (CMM) that defines key activities required at different levels of process maturity. The SEI approach provides a measure of the global effectiveness of a company’s software engineering practices and establishes five process maturity levels that are defined in the following manner:

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

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

Level 3: Defined. The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software.

This level includes all characteristics defined for level 2.

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

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

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

The SEI has associated key process areas (KPAs) with each of the maturity levels.

The KPAs describe those software engineering functions (e.g., software project planning, requirements management) that must be present to satisfy good practice at a particular level. Each KPA is described by identifying the following characteristics:

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

- **Methods for verifying implementation**—the manner in which proper practice for the KPA can be verified.

Eighteen KPAs (each described using these characteristics) are defined across the maturity model and mapped into different levels of process maturity. The following

KPAs should be achieved at each process maturity level:

Process maturity level 2

- Software configuration management
- Software quality assurance
- Software subcontract management
- Software project tracking and oversight
- Software project planning
- Requirements management

Process maturity level 3

- Peer reviews
- Intergroup coordination
- Software product engineering
- Integrated software management
- Training program
- Organization process definition
- Organization process focus

Process maturity level 4

- Software quality management
- Quantitative process management

Process maturity level 5

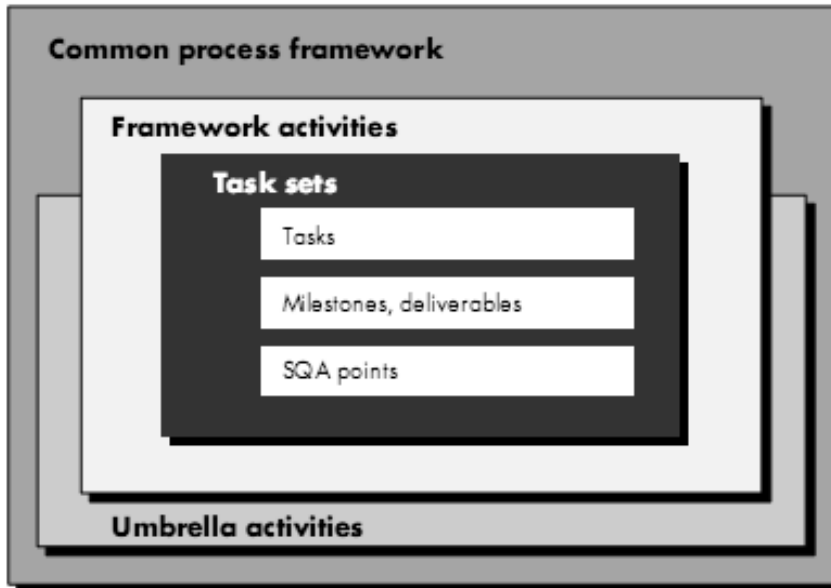
- Process change management

- Technology change management
- Defect prevention

It establishes the foundation for a complete software process by identifying a small number of frame works, activities that are applicable to all software projects, regardless of their size or complexity. It encompasses a set of umbrella activities, hat are applicable across the entire software process.

Each frame work activity is populated by a set of software engineering actions a collection of related tasks that produces a major software engineering work product. Each action is populated with individual work tasks that accomplish some part of the work implied by the action.

Software Process Frame Work:



The software process

Task Set: It defines the actual work to be done to accomplish the objectives of a software engineering action. The task set that best accommodates the needs in the project and the characteristics of the team is chosen. The typical umbrella activities in the generic view of software engineering category include the following:

- Software Project tracking and control
- Software Configuration Management
- Software Quality assurance
- Risk Management
- Formal Technical reviews
- Measurement

- Reusability Management
- Work product preparation & production

The generic process frame work basically consists of 5 frame work activities. They are:

Communication: This involves heavy communicate and collaboration with the customer and encompasses requirements gathering and other related activities.

Planning: This establishes a plan for software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resourced that will be required, the work products to be produced, and a work, schedule.

Modeling: This encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.

Construction: This combines code generation and the testing that is required uncovering errors in the code.

Deployment: The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

1.6. The Capability Maturity Model Integration (CMMI)

The CMMI represents a process meta-model in 2 different ways:

The CMMI is a significant achievement in software engineering. It provides a comprehensive discussion of the activities and actions that should be present when an organization builds computer software.

a. Continuous Model: It describes a process in two dimensions. They are: Process area, which is formally assessed against specific goals and practices and rated acc to capability levels.

Capability levels, categorized into different categories which are:

Level Number	Level Name	Description
0	Incomplete	The process area is either not performed or does not achieve all goals and objectives defined by CMMI.

1	Performed	All of the specific goals of the process area have been satisfied. Work tasks required to produce defined work products are being conducted.
2	Managed	All level-1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy. All work tasks and work products are monitored, controlled, and reviewed, and the area is evaluated for adherence to the process description.
3	Defined	All level-2 criteria have been achieved. In addition, the process is tailored and conforms to guidelines and contributes work products, measures, and other process improvement information to the organizational process assets.
4	Quality Managed	All level-3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment.
5	Optimized	All capability level-4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

b. Staged Model:

It defines five maturity levels, each of which can be achieved if and only if the specific goals and practices associated with a set of process areas are achieved.

Level	Focus	Process Areas
Optimizing	Continuous process improvement	<ul style="list-style-type: none"> . Organizational Innovation & Deployment. . Causal Analysis & Resolution

Quantitatively Managed	Quantitative Management	<ul style="list-style-type: none"> . Organizational process performance . Quantitative project management
Defined	Process standardization	<ul style="list-style-type: none"> . Requirements development . Technical solution . Product integration . Verification . Validation . Organizational process focus . Organizational process definition . Organizational training . Integrated project management . Integrated supplier management . Risk management . Organizational environment for integration . Integrated teaming
Managed	Basic project management	<ul style="list-style-type: none"> . Requirements management . Project planning . Project monitoring & Control . Supplier agreement management . Measurement & analysis . Process and product quality assurance . Configuration management

1.7. Process patterns

The software process can be defined as collection of patterns that define a set of activities actions, work tasks, work products and related behaviors required to develop computer software. A process pattern provides us with a template a consistent method for describing an important characteristic of software process. By combining patterns, a software team can construct a process that best meets the needs of a project. Patterns can be defined at any level of abstraction which can describe either a process or frame active.

The following template describes a process pattern:

Pattern Name: The pattern is given a meaningful name that describes its function within the software process.

Intent: The objective of the pattern is described briefly in the intent.

Type: The pattern type is specified in type. These are of three types namely TASK PATTERNS (define a software engineering action or work task that is part of the process and relevant to successful software engineering practice), STAGE PATTERNS (define a frame work activity for the process and it incorporates multiple task patterns that are relevant to the stage), PHASE PATTERNS (define the sequence of frame work activities that occur with the process, even when the overall flow of activities is iterative in nature).

Initial context: The conditions under which the pattern applies are described in initial context. Prior to this, some information is required regarding

- What organizational or team related activities have already occurred?
- What is the entry state for the process?
- What software engineering information or project information already exists?

Problem: The problem to be solved by the pattern is described in the problem.

Solution:

The implementation of the patters is described in the solution. This section discusses how the initial state of the process is modified a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern us transformed as a consequence of successful execution of patters.

Resulting content: The conditions that will result one the pattern has been successfully implemented are described in resulting context section. Upon the completion of the pattern, some information is required regarding.

- What organizational or team-related activities must have accrued?
- What is the exit state for the processes?
- What software engineering information or project informal has been developed?

Related patter: A list of all process patterns that are directly related to this one are provided as a hierarchy or in some other diagrammatic form in related patterns section.

The specific instances in which the pattern is applicable are indicated in known uses/ examples section.

Advantages:

1. They provide an effective mechanism for describing any software process.
2. They enable a software engineering organization that begins at a high level of abstraction.
3. They can be reused for the definition of process variants, once they are developed.

1.8 Process Assessment:

The process itself should be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering. A number of different approaches have been proposed for software process assessment. They are:

- **SCAM PI (Standard CMMI Assessment Method for Process Improvement):** This standard provides a five-step model that incorporates initiating, diagnosing, establishing, acting and learning. This uses SEICMMI as the basis for assessment.
- **CBA IPI (CMM-Base) Appraisal for Internal Process Improvement):** This provides a diagnostic technique for assess the relative maturity of a software organization, using the SEICMMI as the basis for the assessment.
- **SPICE (ISO/IEC 15504):** This standard defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.
- **ISO 9001: 2000 for software:** This is a generic standard that applies to any organization that wants to improve the overall quality of the products, systems or services that it provides. ISO has developed this standard to define the requirements for a quality management system that will serve to produce higher quality products and there by improve customer satisfaction. The cycle adopted by this standard is “PLAN – DO – CHECK – ACT”.

1.9. Personal and Team process models

If a software model has been developed at a corporate or organizational level, it can be effective only if it is amenable to significant adaptation to meet the project team needs that is actually doing software engineering work. Each software engineer would create a process that fits best his/her needs besides meeting the broader needs of the team & organization, which may be called as PERSONAL SOFTWARE PROCESS (PSP). Alternatively, the team itself would create its own process besides meeting the mass own needs of

individuals & broader needs of organization, which may be termed as TEAM SOFTWARE PROCESS (TSP).

PSP: It emphasizes personal measurement of both the work product. In addition, it makes the practitioner responsible for project planning and empowers the practitioner to control the quality of all software work products that are developed. This model defines five frame work activities. They are:

Planning: This activity isolates requirements and based on these, develops both size and resource estimates besides a defect estimate. All metrics are recorded on work sheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design: External specifications for component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High-level design review: Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development: The component level design is defined and reviewed. Code is generated, reviewed, compiled and tested. Metrics are maintained for all important tasks and work results.

Post mortem: Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

Advantages:

- PSP represents a disciplined, metrics – based approach to software engineering.
- PSP introduction results in significant improvement of software engineering productivity & quality.
- PSP stresses the need for each software engineer to identify errors early & to understand types of errors that he is likely to make.

Disadvantages:

- PSP is intellectually challenging and demands a level of commitment that is not always possible to obtain.
- Training is relatively lengthy
- Training costs are high

- The required level of measurement is culturally difficult for many software people.

TSP: The goal of TSP is to build a “self-directed” project team that organized itself to produce high-quality software. A self directed team has a consistent understanding of its overall goals and objectives. It

- defines roles & responsibilities for each team member
- tracks quantitative project data
- identifies a team process that is appropriate for the project and a strategy for implementing the process.
- defines local standards that are applicable to the team’s software engineering work.
- Continually assesses risk and reacts to it.
- Tracks, manages and reports project status.

TSP defines the following frame work activities:

- Launch
- High-level Design
- Implementation
- Integration & Test
- Postmortem

TSP makes use of a wide variety of scripts, forms and standards that serve to guide team members in their work, in which scripts define specific process activities and other more detailed work functions that are part of team process.

Advantages:

- TSP allows the team to adapt to changing customer needs and lessons learned from previous activities, with launch activity.
- It recognizes that the best software teams are self-directed.
- It provides distinct and quantifiable benefits in productivity and quality.

Lesson end Questions

1. Indicate various problems related with software development.

2. What are various phases of software development?

3. Provide some examples of software. Are there any differences between software and hardware? Justify?

4. Find the distinct possible examples of Application, Languages, Packages and Software?

References:

1. Roger S. Pressman, *Software Engineering – A Practitioners Approach*, Sixth Edition, McGraw –Hill, 2005.

2. Sommerville I; *Software Engineering*(Perason) 6th Edition

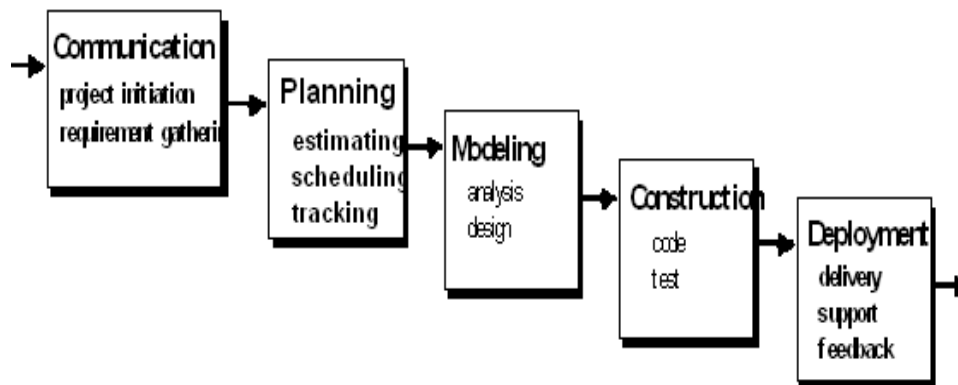
3. Brooks, F., *The Mythical Man-Month*, Addison-Wesley, 1975.

4. Glass, R.L., *Software Runaways*, Prentice-Hall, 1997.

2. Process models

2.1 The Waterfall Model

It is also termed as the classic life cycle which suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progress through planning, modeling, construction and deployment. It is the oldest paradigm for software engineering it basically consists of five levels:



Waterfall Model

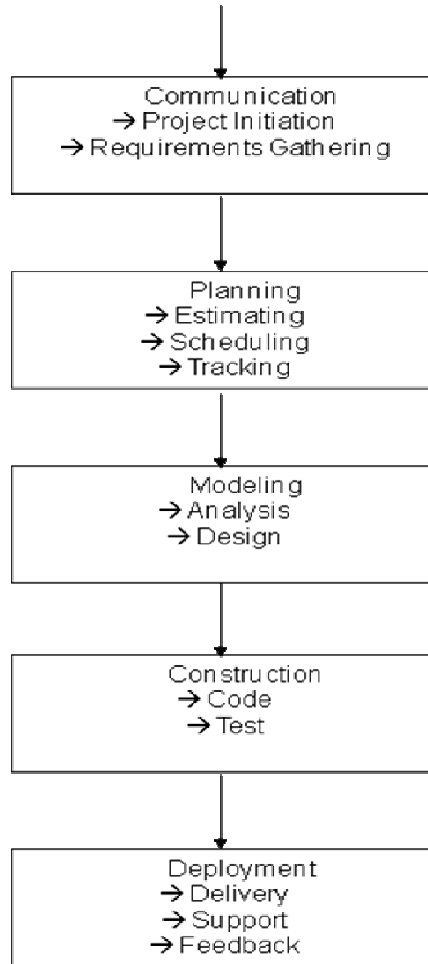
In this model, the phases are organized in a linear order. The model was originally proposed by Royce. In this model, a project begins with feasibility analysis upon successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins. The design starts after the requirements analysis is complete, and coding begins after the design is complete. Once the programming is completed, the code is integrated and testing is done upon, successful completion of testing, the system is installed.

Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and the beginning of the risk, some certification mechanism has to be employed as the end of each phase. This is usually done by some verification and validation (which ensures that the output of a phase is consistent with the input (output of previous phase) and overall requirements of the system).

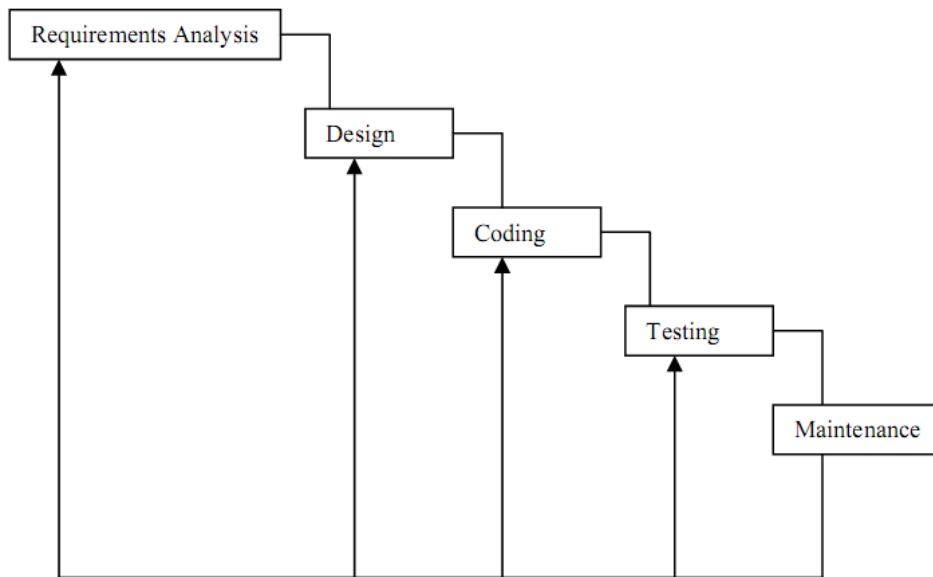
The outputs of all phases are often termed as “works products” and are usually in the form of documents. The following documents generally form a reasonable set that should be produced in each project.

- Requirements Document
- Project plan
- Design Document
- Test plan of Test Reports

- Final code
- Software Manuals



A slight modification of the waterfall model is a model with feedback. Once software is developed and is operational, then the feedback to various phases may be given.



Water fall model with feedback

Advantages:

1. It is simple.
2. It is conceptually straight forward and divides the large task of building a software system into series of phases, each phase dealing with a separate logical concern.
3. It is easy to administer in a contractual setup as each phase is completed and its work product produced, some amount of money is given by customer of dev. Organization.

Limitations:

1. It assumes that the requirements of a system can be frozen before the design begins. But, unchanging requirements is unrealistic for some projects.
2. Freezing the requirements usually requires choosing the hardware. If the hardware is selected early for a large project (which takes long time to get completed), the changing technologies that the final software uses becomes obsolete.
3. It follows the "BIG – BANG" approach – the entire software is delivered in one shot at the end i.e. It has the ALL or NOTHING value proposition.
4. It is a document – driver process that requires formal documents at the end of each phase.

2.2 Incremental Process Models

This is a process model that is designed to produce the software in increments in situation like.

- The overall scope of development effort precludes a purely linear process, even though the initial software requirements are well-defined.
- If there is a need to provide a limited of software functionality to users quickly and then refine and expand on that functional in later software releases.

These are of two kinds. There are:

The Incremental Model

It combined the elements of waterfall model applied in an iterative fashion. The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable “increments” of the software. When this model is used, the first increment is often a Core Product i.e. Basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use and/or Evaluation, a plan is developed for the next increment, which addresses the modification of the core product for its betterment. This process is repeated following the delivery of each increment, until the complete product in produced.

Advantages:

1. Incremental Model is iterative in nature.
2. It focuses on the delivery of an operational product with each increment.
3. It is useful when staffing is unavailable for a complete implementation.
4. Increments can be planned to manage technical risks.

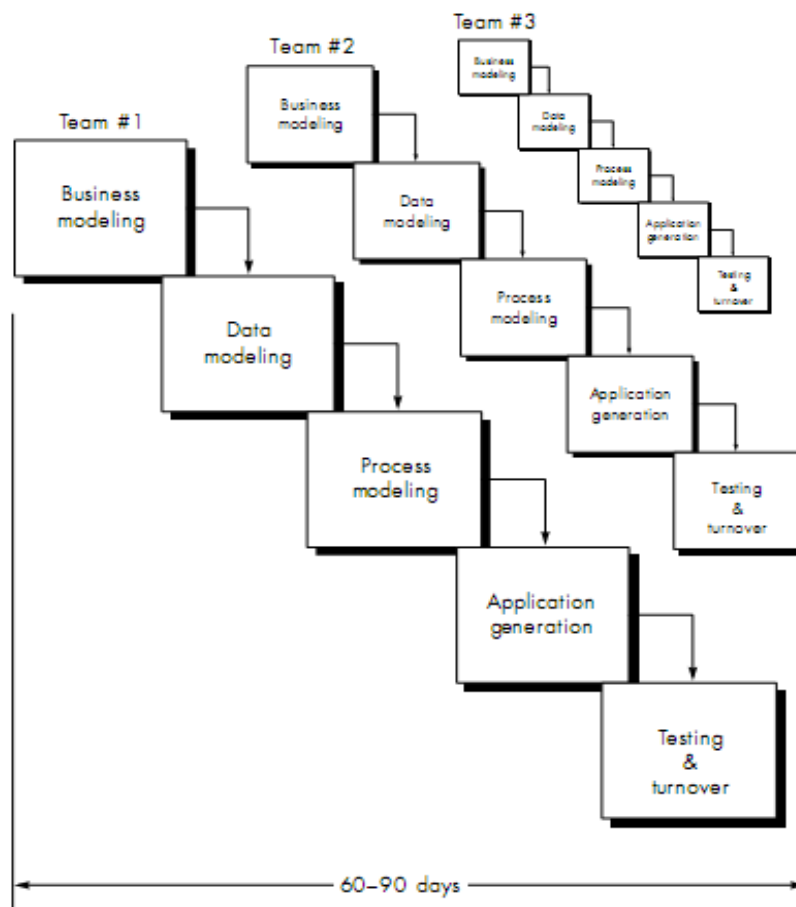
The RAD Model

RAD (Rapid Application Development) is an incremental software process model that emphasizes a short development cycle. It is a high – speed adaptation f waterfall model, in which rapid development in achieved by using a component based construction approach. If requirements are well understood and project scope in constrained, the RAD process enables a development team to create a “fully functional system” within a very short

period. RAD approach maps into the genetic frame work activities comprising of Communication, Planning, Modeling, Construction, and Deployment.

Communication works to understand the business problem and the information characteristics that the software must accommodate. Planning is essential because multiple software teams work in parallel on different system functions. Modeling encompasses three major phases business Modeling, Data Modeling and Process Modeling and establisher design representation that serve as the basis for RAD's construction activity. Construction emphasizes the use of pre-existing software components and the application of automatic code generation. Finally, Development establishes a basis (4) for subsequent iterations, if required.

The RAD model is illustrated in the below figure:



RAD model

Advantages:

- Each major function of the project can be addressed by a separate RAD team and their integrated to form a whole.

Drawbacks:

1. For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
2. If developers and customers are not committed to the rapid fire activities necessary to complete the system in an abbreviated time frame, RAD projects will fail.
3. If a system cannot be properly modularized, building the components necessary for RAD will be problematic.
4. If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components tuning the interfaces to system components, the RAD approach, may not work.
5. RAD may not be appropriate when technical risks are high.

2.3 Evolutionary Process Models

These models were developed to meet the competitive/business presence. It was mainly developed when the details of system extensions are to be defined even though set of system requirements are well understood. These models are iterative in nature. These are characterized in a manner that enables software engineers to develop increasingly more complete versions of software. These are mainly of three types. They are:

Prototyping Model

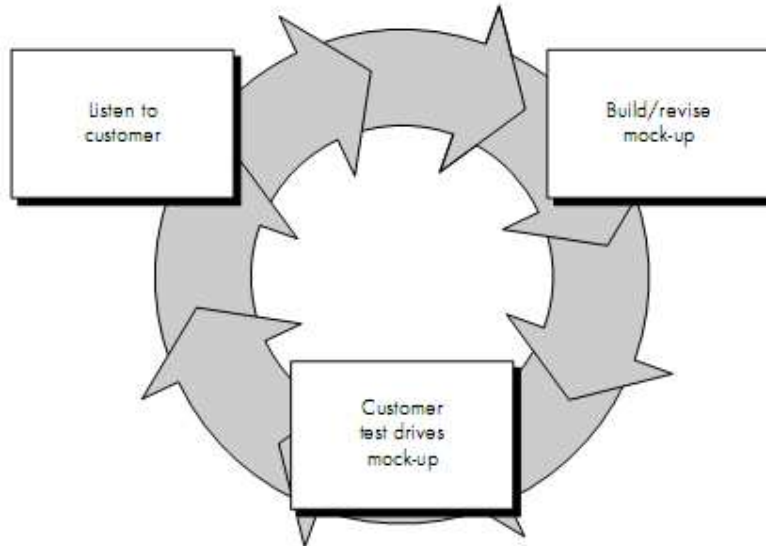
In this model, a throw away prototype is built to understand the requirements, instead of freezing the requirements before any design or coding can proceed. This prototype is developed based on currently known requirements.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements. A development process using throw away prototyping typically proceeds as follows:

1. The development of the prototype typically starts when the preliminary version of requirements specification document has been developed.
2. After the prototype has been developed, the end users and clients are given an opportunity to use it and play with it.
3. Based on their experience, they provide feedback to the developers regarding the prototype.
4. Based on the feedback, the prototype is modified to incorporate some of the suggested changes that can be done easily, and then users and the clients are again allowed to use the system.

5. This cycle repeats, until the feedback is given by the clients and users that the prototype meets the requirements of them completely.

6. Based on the feedback, the initial requirements are modified to produce the final requirements specification, which is then used to develop the production quality system.



The prototyping paradigm

7. It begins with communication. The software engineer and customer meet to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

8. A prototyping iteration is planned quickly and modeling occurs.

9. The Quick Design focuses on a representation of those aspects of software that'll be visible to the customer / end-user and this leads to the construction of a prototype.

10. The prototype is deployed and then evaluated by the customer/users.

11. Feedback is used to refine the requirements for the software.

12. Iteration occurs as the prototype is tuned to satisfy that needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Advantages:

1. The development approach is quick and dirty with the focus on only those features which are not properly understood.
2. As the prototype is to be thrown away, only minimal documentation needs to be produced during prototyping.
3. Another important cost-cutting measure is to reduce testing.
4. It is well suited for projects where requirements are hard to determine.
5. It is an excellent technique for reducing some types of risks associated with a project.

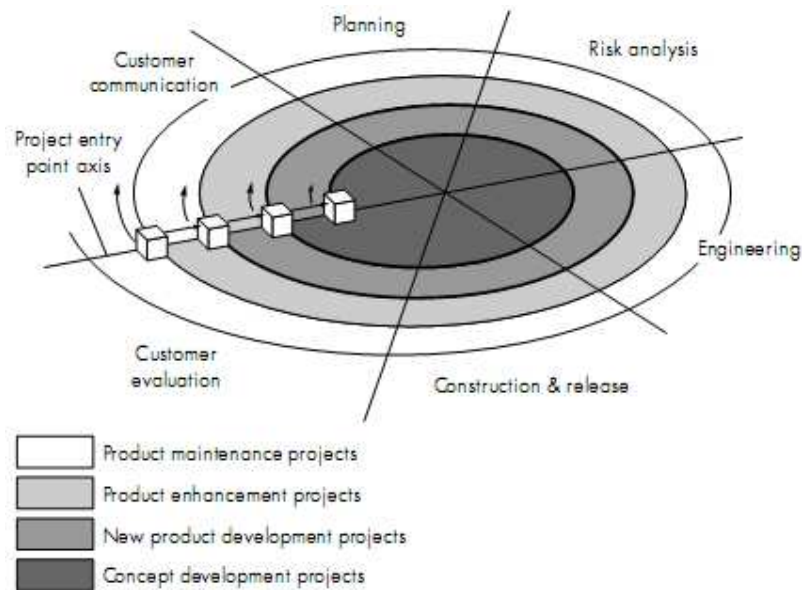
Disadvantages:

It can be problematic for two reasons.

- (a) The customer sees what appears to be a working version of the software, unaware of fact that the software quality or long-term maintainability is not considered to get it working and the prototype is held together. When informed that the product must be re-built so that high-levels of quality can be maintained, the customer demands that a “few fixes” be applied to make the prototype a working, product.
- (b) The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known. An inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than – ideal choice has now become an integral part of the system.

The Spiral Model

It is an evolutionary software process model that couples the iterative nature of prototyping, with the controlled and systematic aspects of waterfall model. It provides the potential for rapid development of increasingly more complete version of software. Using this model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.



A spiral model is divided into a set of frame work activities defined by the software engineering team, each of which represents one segment of the spiral path. As the process begins, the software team performs activities that are implied by a circuit around the spiral in a clock-wise direction, beginning at the centre. Risk is considered as each revolution is made. A combination of work products and conditions (ANCHOR-POINT-MILE STONES) that are attained along the path of the spiral are noted for each evolutionary process.

The first circuit around the spiral might result in the development of a product specification. Subsequent passes around the spiral might be used to develop a prototype and the progressively mentioned versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manages adjusts the planned number of iterations required to complete the software.

The first circuit around the spiral might represent a “CONCEPT DEVELOPMENT PROJECT” which starts at the core of the spiral and continues for multiple iterations until CONCEPT DEVELOPMENT is complete. If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a “NEW PRODUCT DEVELOPMENT PROJECT” begins. Whenever a change is initiated, the process starts at the appropriate entry point.

Advantages:

1. This model can be adapted to apply throughout the life of the computer software.
2. It is a realistic approach to the development of large-scale systems and software.

3. The developer and customer better understand and react to risks at each evolutionary level, because, software evolves as the process progresses.
4. This model uses PROTOTYPING as a risk reduction mechanism and also enables the developer to apply prototyping approach at any stage in the evolution of the product.
5. It maintains the systematic step-wise approach of waterfall model, in corporation it into an iterative frame work that more realistically reflects the real world.
6. It demands a direct consideration of technical risks at all stages of the project which enables the uses to reduce risks before they become problematic.

Disadvantages:

1. It may be difficult to convince customers that the evolutionary approach is controllable.
2. As risk assessment is demanded, if a major risk is not uncovered and managed, problems will undoubtedly occur.

The concurrent development model

It can be schematically represented as a series of frame work activities, software engineering actions and tasks and their associated states.

The above diagram provides a schematic representation of one software engineering task within the modeling activity for the concurrent process model. The activity modeling may be in any one of the states noted at any given time. Similarly other activities or tasks can be represented in an analogous manner. All activities exist concurrently, but reside in different states.

- The modeling activity exists in the “NONE” state, while critical communication was completed.
- It then makes a transition into the “UNDER DEVELOPMENT” state.
- It then moves into the “AWAITING CHANGE” state as requirement changes are indicated by the customer.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions or tasks.

For example During early stages of design, an inconsistency in the analysis model is uncovered, by which the event, “ANALYSIS MODEL CORRECTION”

is generated, triggering the analysis action from “DONE” state into the “AWAITING CHANGES” state.

Advantages:

1. It is applicable to all types of software development.
2. It provides an accurate picture of the current state of a project.
3. It defines a network of activities.
4. Events generated at one point in the process network trigger transitions among the states.

2.4. The Unified Process

- It is an attempt to draw on the best features and characteristics of conventional, software process models, but characterize them in a way that implements many of the best principles of agile software development.
- It recognizes the importance of customer communication and streamlined methods for describing the customer’s view of a system.
- it emphasizes the important role of software architecture and helps the architect focus on the right goals, such as understandability, reliance to future changes and re-use.
- It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

History: During 1990s, James Rumbaugh, Grady Booch and Ivar Jacobson worked on a “Unified Method” that combines the best features of each of their individual methods and adopt additional features proposed by other experts in object oriented field. The result was “UML” a Unified Modeling Language that contains a robust notation for the modeling and development of object oriented systems.

Advantages: UML provides the necessary technology to support object-oriented software engineering practice.

Drawback: UML doesn’t provide the process frame work to guide project teams in their application of technology.

To overcome the drawback of UML, all the 3 of them developed the “UNIFIED PROCESS” which is a frame work for object-oriented software engineering using UML. Different phases of up are depicted by the foll diagram, in relation to genuine activities:

Interruption: It encompasses both customer communication and planning activities. In this phase

- Business registration for software are identified,
- A rough architecture for system is proposed
- A plan for iterative incremental nature of the ensuring project is developed.

Elaboration: It encompasses the customer indications and modeling activities of the generic process model. This phase refines and expands the preliminary use-cases that were developed as part of interception phase and expands the architectural representation to include 5 different views (use case, analysis, design, implementation & deployment) of software plan is carefully reviewed at the culmination of this phase to ensure that the scope, risks and delivery dates remain reasonable. Modifications to the plan may be made at this time.

Construction: Using the architectural model as input, this phase develops or acquires software components that'll make each use-case operational for end-users. To accomplish this, analysis and design models of elaboration phase are completed to reflect the final version of software increment. All necessary and required features and functions of software increment are then implemented in source code and so, Unit tests are designed and executed for each, besides integration activities being conducted.

Transition: It encompasses the latter stages of generic construction activity and the first part of generic deployment activity. The software team creates the necessary support information that is required for the release. Software is given to end-users for beta testing and use feedback reports both defects & necessary changes.

Production: This phase coincides with the deployment activity of the generic process. During this phase, the on-going use of software monitored, support for the operating environment is provided and defect reports and requests for changes are submitted and evaluated.

2.5. Functional Requirements:

These are statements of services the system should provide, how the system should react to particular input and how the system should behave in particular situations. These may also explicitly state what the system should not do.

These depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. They describe the system function in detail, its inputs

and outputs, exceptions and so on. They define specific facilities to be provided by the system. In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that the requirements should not have contradictory definitions.

For example Function Requirements for a University library system called LIBSYS, used by students of faculty to order books & documents from other libraries.

- The user shall be able to search either all of the initial set of databases or select a subset from it.
- The system shall provide appropriate viewers for the user to read documents in the document store.
- Every order shall be allocated a unique identifies (ORDER – ID), which the user shall be able to copy to the account’s permanent storage area.

2.6. Non-functional Requirements

These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. They often apply to the system as a whole. They do not usually just apply to individual system features or services.

As the name suggests, these requirements are not directly concerned with the specific functions delivered by the system. They may select to emergent system properties such as salability, response lime and store occupancy. Alternatively, they may define constraints on the system such as the capabilities of the input/output devices and the data representations used in system interfaces.

Failing to meet a non-functional requirement can mean that the whole system is unusable. For example if an air-craft system doesn’t meet its reliability requirements, it will not be certified as “SAFE FOR OPERATION”. on functional requirements neither are nor just concerned with the software system to be developed. Some of them may constrain the process that should be used to develop the system.

Eg: Process requirements include.

- A specification of quality standards that should be used in the process.
- A specification that the design must be produced with a particular CASE toolset and

- A description of the process that should be followed Non-functional requirements arise through user needs
- because of budget constraints
- because of organizational policies.
- because of the need for interoperability with other software or hardware systems.
- because of external factors such as safety regulations or privacy legislation.

Non-functional requirements are broadly classified into three types. They are:

Product Requirements: These specify the product behavior. Performance requirements on how fast the system must execute and how much memory it requires.

- Reliability requirements that set out the acceptable failure rate
- Portability requirements
- Usability requirements

Organizational Requirements: These are delivered from policies and procedures in the customer's and developer's organization.

- Process standards that must be used
- Implementation requirements such as the programming language or design method used
- Delivery requirements that specify when the produced and its documentation are to be delivered.

External Requirements: These are all that are derived from factors external to the system and its development process.

- Inter operability requirements that define how the system interacts with systems in other organizations.
- Legislative requirements that must be followed to ensure that the system operates within the law.

Metrics for specifying non-functional requirements:

Property	Measure
----------	---------

Speed	Processed Transactions / second User / Event response time Screen refresh time
Size	K-bytes Number of RAM chips
Ease and Use	Training time Number of help framed
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target-dependent statements Number of largest systems

2.7. User requirements

The user requirements for a system should describe the functional and non-functional requirements so that they are understandable by system users without detailed technical knowledge. They should only specify the external behavior of the system and should avoid, as far as possible, system design characteristics. The user requirements should be written in simple language, with simple table forms and intuitive diagrams. While writing the user requirements, the following points must be considered, as various problems may arise.

Several different requirements must not be expressed together as a single requirement

Eg: LIBSYS shall provide a financial accounting system that maintains records of all payments made by users of the system. System managers may configure this system so that regular users may receive discounted rates.

Explanation:

The requirement includes both conceptual and detailed information. It expresses the concept that there should be an accounting system as an inherent part of LIBSYS. However, it also includes the detail that the accounting system should support discounts for regular LIBSYS users. This detail should be left to system req.

NOTE:

1. User requirements should be separated from more detailed system requirements.
2. User requirements that include too much information constrain the freedom of the system developer to provide innovative solutions to user problems and are difficult to understand. The user requirement should simply focus on the key-facilities to be provided.
3. A rationale is to be associated with each user requirement, to explain why the requirement has been included.

2.8. System Requirements

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. Ideally, they should describe the external behavior of the system and its operations constraints. They should not concern with how the system should be designed or implemented.

It is essential to write the user requirements in a language that non-specialists can understand. These system requirements can be written in more specialized notations as follows, which include stylized, structured natural language, graphical models of the requirements such as use-cases to formal mathematical specifications:

Notation	Description
Structured Natural Language	This approach depends on defining standard forms or templates to express the requirements specification.
Design Description Languages	This approach user a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system.

Graphical Notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. Use-case descriptions and sequence diagrams are commonly used now.
Mathematical Specifications	These are notations based on mathematical concepts such as finite state machines, or sets. These unambiguous specifications reduce the arguments between customer & contractor about system functionality.

2.9. Interface Specification:

If a new system and existing systems must work together, the interfaces of existing systems have to be precisely specified. These specifications should be defined early in the process and included in the requirements document. There are three types of interfaces that may have to be defined:

Procedural interfaces: Where existing programs or subsystems offer a range of services that are access by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces (API).

Data structures: Those are passed from one sub-system to another. Graphical data models are the best notations for this type of description.

Data Representations: That has been established for an existing sub system. These interfaces are most common in embedded, real-time system. 'ADA' language supports this level of specification.

2.10 The software requirements Document

Also called as Software Requirements Specification (SRS), is the official statements of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. The following figure illustrated how the possible users of the document use it:

Possible users	Usage
System customers	Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
Managers	Use the requirements document to plan a bid for the system and to plan the system development process
System engineers	Use the requirements to understand what system is to be developed
System test engineers	Use the requirements to develop validation tests for the system
System maintenance engineers	Use the requirements to understand the system and the relationships between its parts

- The requirement document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers and including information about possible system evolution.
- The level of detail to be included depends on the type of system that is being developed and the development process used.

The IEEE standard suggests the following structure for requirements documents:

1. INTRODUCTION

- 1.1. Purpose of the requirements document
- 1.2. Scope of the product
- 1.3. Definitions, acronyms of abbreviations
- 1.4. References
- 1.5. Overview of the remainder of the document

2. GENERAL DESCRIPTION

- 2.1. Product perspective
- 2.2. Product functions
- 2.3. User characteristics
- 2.4. General constraints
- 2.5. Assumptions and dependencies

3. SPECIFIC REQUIREMENTS:

- These cover functional requirements
- Non functional requirements
- Interface requirements

The requirements may document external interfaces, describe system functionality & performance, specify logical d/b requirements, design constraints, emergent system properties and quality characteristics.

4. APPENDICES**5. INDEX**

The following figure illustrates a possible organization for a requirements document that is based on IEEE standard:

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe its functions and explain how it'll work with other systems. It should describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. Assumptions should not be made about the experience or reader's expertise.
User requirements definition	The services provided for the user and non-functional system requirements should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers. Product & process standards which must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture showing the distribution of functions across system modules.

	Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional of non-functional requirements in more detail. If necessary, further detail may also be added to non-functional requirements. Eg: Interfaces to other systems may be defined.
System Models	This should sat out one or more system models showing the relationships b/w system components and system fits environment. These might be object models, data flow models, semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	These should provide detailed, specific information which is relates to the application which is being developed. Examples of appendices that may be included are hardware and d/b descriptions.
Index	Several indexed to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, etc.

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 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.

Software engineering covers the entire range of activities used to develop software. The activities include requirements analysis, program development using some recognized approach like structured programming, testing techniques, quality assurance, management and implementation and maintenance. Further, software engineering expects to address problems which are encountered during software development.

Prescriptive software process models have been applied for many years in an effort to bring order and structure to software development. Each of these conventional models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modeling, construction, and deployment,

Lesson end Questions

1. Indicate various problems related with software development.
2. Give a comparative analysis of various types of software process models.
3. What are various phases of software development life cycle?
4. What are different levels of capability maturity model?
5. Describe the law of conservation of organizational stability.
6. Provide three examples of software projects that would be amenable to the incremental model

UNIT – II

Structure

1. SOFTWARE ENGINEERING PRACTICE

- 1.1 Introduction
- 1.2 Software Engineering Practice
- 1.3 Communication Practices
- 1.4 Planning Practices
- 1.5 Modeling Practice
- 1.6 Construction Practice
- 1.7 Deployment

2. SYSTEM ENGINEERING

- 2.1 Introduction
- 2.2 Computer-based systems
- 2.3 The system engineering hierarchy
- 2.4 Business process engineering: an overview
- 2.5 Product engineering: an overview
- 2.6 Requirements engineering

3. REQUIREMENT ENGINEERING

- 3.1 Introduction
- 3.2 Requirements engineering tasks
- 3.3 Initiating the Requirements Engineering Process
- 3.4 Eliciting Requirements
- 3.5 Building Analysis Model
- 3.6 Software Prototyping and Specification
- 3.7 Validation requirements

Objectives

After going through this unit, you should be able to:

- Discuss about software engineering practice;
- Discuss principles of communication practices;
- Discuss principles planning practices and modeling practice;
- Discuss about construction practice and deployment.
- define system engineering hierarchy;
- understand the Business process engineering: an overview;
- understand the Product engineering: an overview;
- discuss about Requirements engineering.
- discuss about Requirements engineering tasks;
- learn about Initiating the Requirements Engineering Process

- understand the Eliciting Requirements;
- understand the Building Analysis Model;
- understand Software Prototyping and Specification.

1. Software Engineering Practice

1.1 Introduction

A dark image of software engineering practice to be sure, but upon reflection many of the readers of this book will be able to relate to it. People who create computer software practice the art or craft or discipline' that is software engineering. But what is software engineering "practice"? In a generic sense, practice is a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis. Practice allows managers to manage software projects and software engineers to build computer programs. Practice populates a software process model with the necessary technical and management how to have to get the job done. Practice transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success.

1.2 Software Engineering Practice

Practice is a broad array of concepts, principles, methods, and tools that you must consider as software is planned and developed.

It represents the details the technical considerations and how to be that are below the surface of the software process the things that you'll need to actually build high-quality computer software.

The Essence of Practice

This section lists the generic framework (communication, planning, modeling, construction, and deployment) and umbrella (tracking, risk management, reviews, measurement, configuration management, reusability management, work product creation, and product) activities found in all software process models.

George Polya, in a book written in 1945, describes the essence of software engineering practice ...

1. *Understand the problem* (communication and analysis).
 - *Who are the stakeholders?*
 - *What are the unknowns? "Data, functions, features to solve the problem?"*
 - *Can the problem be compartmentalized? "Smaller that may be easier to understand?"*

- *Can the problem be represented graphically? Can an analysis model be created?*
2. *Plan a solution (modeling and software design).*
 - *Have you seen a similar problem before?*
 - *Has a similar problem been solved? If so, is the solution reusable?*
 - *Can sub-problems be defined?*
 - *Can you represent a solution in a manner that leads to effective implementation?*
 3. *Carry out the plan (code generation).*
 - *Does the solution conform to the plan?*
 - *Is each component part of the solution probably correct?*
 4. *Examine the result for accuracy (testing and quality assurance).*
 - *Is it possible to test each component part of the solution?*
 - *Does the solution produce results that conform to the data, functions, features, and behavior that are required?*

Core Principles

The Reason It All Exists: Provide value to the customer and the user. If you can't provide value, then don't do it.

KISS—Keep It Simple, Stupid! *All design should be as simple as possible, but no simpler.* This facilitates having a more easily understood and easily maintained system.

Maintain the product and project “vision.” *A clear vision is essential to the success of a SW project.*

What you produce, others will consume. *Always specify, design, and implement knowing someone else have to understand what you are doing.*

Be open to the Future. *Never design yourself into a corner.* Always ask “what if,” and prepare yourself for all possible answers by creating systems that solve the general problem, not just the specific one.

Plan Ahead for Reuse. *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*

Think! Placing clear, complete thought before action almost always produces better results.

1.3 Communication Practices

Before customer requirements can be analyzed, modeled, or specified they must be gathered through a *communication* (also called *requirement elicitation*) activity.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that confront a S/W engineer.

In this context, the following are communication **principles** and concepts that apply to customer communication:

Listen: focus on the speaker's words, rather than formulating your response to those words. Be a polite listener.

Prepare before you communicate: Spend the time to understand the problem before you meet with others "research".

Someone should facilitate the communication activity. Have a leader "moderator" to keep the conversation moving in a productive direction.

Face-to-face communication is best.

Take notes and document decisions.

Collaborate with the customer. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Stay focused, modularize your discussion. The facilitator should keep the conversation modular; leaving one topic only after it has been resolved.

Draw pictures when things are unclear.

(a)Once you agree to something, move on; (b) if you can't agree to something, move one; (c) if a feature or function is unclear and can't be clarified at the moment, move on.

Negotiation is not a contest or a game. It works best when both parties win.

1.1 Planning Practices

The *planning* activity encompasses a set of management and technical practices that enable the S/W team to define a road map as it travels toward its strategic goal and tactical objectives.

Regardless of the rigor with which planning is conducted, the following principles always apply:

Understand the project scope. Scope provides the S/W team with a destination.

Involve the customer (and other stakeholders) in the planning activity. The customer defines priorities and establishes project constraints. S/W engineers must often negotiate order of delivery, timelines, and other related issues.

Recognize that planning is iterative. A plan must be adjusted to accommodate changes.

Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.

Consider risk as you define the plan.

Be realistic. Even the best S/W engineers make mistakes.

Adjust granularity as you plan. A fine granularity plan provides significant work task detail that is planned over relatively short time increments. A coarse granularity plan provides broader work tasks that are planned over longer time periods.

Define how quality will be achieved.

Define how you'll accommodate changes. "Can the customer request a change at any time?"

Track what you've planned and make adjustments as required.

Barry Boehm states: "You need an organizing principle that scales down to provide simple plans for simple projects."

Boehm suggests an approach that addresses project objectives, milestones, and schedules, responsibilities, management and technical approaches, and required resources.

Boehm calls it ***W⁵HH principle***, after a series of questions that lead to a definition of key project characteristics and the resultant project plan.

Why is the system being developed? Does the business purpose justify the expenditure of people, time and money?

What will be done? Identify the functionality to be built.

When will it be accomplished? Establish a workflow and timeline for key project tasks and identify milestones required by the customer.

Who is responsible for a function? Define members' roles and responsibilities.

Where are they located (organizationally)? Customers also have responsibilities.

How will the job be done technically and managerially? Once a scope is defined, a technical strategy must be defined.

How much of each resource is needed? The answer is derived by developing estimates based on answers to earlier questions.

1.5 Modeling Practices

The process of developing analysis and design models is described in this section. The emphasis is on describing how to gather the information needed to build reasonable models, but no specific modeling notations are presented in this chapter. UML and other modeling notations are described in detail later in the text.

In S/W Eng. work, two models are created: analysis models and design models.

Analysis models represent the customer requirements by depicting the S/W in three different domains: the information domain, the functional domain, and the behavioral domain.

Design models represent characteristics of the S/W that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Analysis Modeling Principles

The information domain of a problem must be represented and understood. The *information domain* encompasses the data that flow into the system (end-users, other systems, or external devices), the data that flow out of the system and the data stores that collect and organize persistent data objects.

Represent software functions. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Represent software behavior. The behavior of the S/W is driven by the interaction with the external environment.

The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered fashion (or hierarchical).

The analysis task should move from essential information toward implementation detail. Analysis begins by describing the problem from the end-user perspective. The “essence” of the problem is described without any consideration of how a solution will be implemented.

Design Modeling Principles

The software design model is the equivalent of an architect’s plans for a house.

Set of principles used:

Design must be traceable to the analysis model. The analysis model describes the information domain of the problem, user visible functions, system behavior, and a set of analysis classes that package business objects with the methods that service them.

The design model translates this information into architecture: a set of subsystems that implement major functions, and a set of component-level designs that are the realization of analysis class.

Always consider architecture. S/W architecture is the skeleton of the system to be built. Only after the architecture is built should the component-level issues should be considered.

Focus on the design of data as it is as important as a design. Data design is an essential element of architectural design.

Interfaces (both user and internal) must be designed. A well designed interface makes integration easier and assists the tester in validating component functions.

User interface design should be tuned to the needs of the end-user. “Ease of use.”

Component-level design should exhibit functional independence. The functionality that is delivered by a component should be *cohesive*- that is, it should focus on one and only one function.

Components should be loosely coupled to one another and to the external environment. *Coupling* is achieved in many ways – via a component interface, by messaging through global data. Coupling should be kept as low as is reasonable. As the level of coupling increases, error

propagation also increases and the overall maintainability of the system decreases.

Design representation (models) should be easily understood.

The design model should be developed iteratively. With each iteration, the designer should strive for greater simplicity.

1.6 Construction Practices

In this text “construction” is defined as being composed of both coding and testing. The purpose of testing is to uncover defects. Exhaustive testing is not possible so processing a few test cases successfully does not guarantee that you have bug free program. Unit testing of components and integration testing will be discussed in greater later in the text along with software quality assurance activities.

Although testing has received increased attention over the past decade, it is the weakest part of software engineering practice for most organizations.

Coding Principles and Concepts

Preparation Principles: Before writing one line of code, be sure of:

1. Understand the problem you are trying to solve.
2. Understand the basic design principles.
3. Pick a programming language that meets the needs of the S/W to be built and the environment in which it will operate.
4. Select a programming environment that provides tool that will make your work easier.
5. Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles: As you begin writing code, be sure you

1. Constrain your algorithm by following structured programming practice.
2. Select the proper data structure.
3. Understand the software architecture.
4. Keep conditional logic as simple as possible.
5. Create easily tested nested loops.
6. Write code that is self-documenting.
7. Create a visual layout.

Validation Principles: After you've completed your first coding pass, be sure you

1. Conduct a code walkthrough.
2. Perform unit test and correct errors.
3. Refactor the code.

Testing Principles

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

1.7 Deployment Practices

Customer Expectations for the software must be managed. "Don't promise more than you can deliver."

A complete delivery package should be assembled and tested.

A support regime must be established before the software is delivered.

Appropriate instructional materials must be provided to end-users.

Buggy software should be fixed first, delivered later.

2. System engineering

2.1 Introduction

Software engineering occurs as a consequence of a process called system engineering. Instead of concentrating solely on software, system engineering focuses on a variety of elements, analyzing, designing, and organizing those elements into a system that can be a product, a service, or a technology for the transformation of information or control.

The system engineering process is called business process engineering when the context of the engineering work focuses on a business enterprise. When a product (in this context, a product includes everything from a wireless telephone to an air traffic control system) is to be built, the process is called product engineering.

Both business process engineering and product engineering attempt to bring order to the development of computer-based systems. Although each is applied in a different application domain, both strive to put software into context. That is, both business process engineering and product engineering work to allocate a role for computer software and, at the same time, to establish the links that tie software to other elements of a computer-based system.

In this lesson, we focus on the management issues and the process-specific activities that enable a software organization to ensure that it does the right things at the right time in the right way.

2.2 Computer-based systems

The word system is possibly the most overused and abused term in the technical lexicon. We speak of political systems and educational systems, of avionics systems and manufacturing systems, of banking systems and subway systems. The word tells us little. We use the adjective describing system to understand the context in which the word is used. Webster's Dictionary defined system in the following way:

1. A set or arrangement of things so related as to form a unity or organic whole;
2. A set of facts, principles, rules, etc., classified and arranged in an orderly form so as to show a logical plan linking the various parts;

3. A method or plan of classification or arrangement;
4. An established way of doing something; method; procedure . . .

Five additional definitions are provided in the dictionary, yet no precise synonym is suggested. System is a special word.

Borrowing from Webster's definition, we define a computer-based system as A set or arrangement of elements that are organized to accomplish some pre-defined goal by processing information. The goal may be to support some business function or to develop a product that can be sold to generate business revenue. To accomplish the goal, a computer-based system makes use of a variety of system elements:

Software: Computer programs, data structures, and related documentation that serve to effect the logical method, procedure, or control that is required.

Hardware: Electronic devices that provide computing capability, the interconnectivity devices (e.g., network switches, telecommunications devices) that enable the flow of data, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function.

People: Users and operators of hardware and software.

Database: A large, organized collection of information that is accessed via software.

Documentation: Descriptive information (e.g., hardcopy manuals, on-line help files, Web sites) that portrays the use and/or operation of the system.

Procedures: The steps that define the specific use of each system element or the procedural context in which the system resides.

The elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product; a robot transforms a command file containing specific instructions into a set of control signals that cause some specific physical action. Creating an information system to assist the marketing department and control software to support the robot both require system engineering.

One complicating characteristic of computer-based systems is that the elements constituting one system may also represent one macro element of a still larger system. The macro element is a computer-based system that is one part of a larger computer-based system. As an example, we consider a "factory automation system" that is essentially a hierarchy of systems. At the lowest level of the hierarchy we have a numerical control machine, robots, and data entry devices. Each is a computer based system in its own right. The elements of the numerical control machine include electronic and

electromechanical hardware (e.g., processor and memory, motors, sensors), software (for communications, machine control, and interpolation), people (the machine operator), a database (the stored NC program), documentation, and procedures. A similar decomposition could be applied to the robot and data entry device. Each is a computer-based system.

At the next level in the hierarchy, a manufacturing cell is defined. The manufacturing cell is a computer-based system that may have elements of its own (e.g., computers, mechanical fixtures) and also integrates the macro elements that we have called numerical control machine, robot, and data entry device.

To summarize, the manufacturing cell and its macro elements each are composed of system elements with the generic labels: software, hardware, people, database, procedures, and documentation. In some cases, macro elements may share a generic element. For example, the robot and the NC machine both might be managed by a single operator (the people element). In other cases, generic elements are exclusive to one system.

The role of the system engineer is to define the elements for a specific computer based system in the context of the overall hierarchy of systems (macro elements). In the sections that follow, we examine the tasks that constitute computer system engineering.

2.3 The system engineering hierarchy

Regardless of its domain of focus, system engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy illustrated in Figure 2.1.

The system engineering process usually begins with a “world view.” That is, the entire business or product domain is examined to ensure that the proper business or technology context can be established. The world view is refined to focus more fully on specific domain of interest. Within a specific domain, the need for targeted system elements (e.g., data, software, hardware, and people) is analyzed. Finally, the analysis, design, and construction of a targeted system element is initiated. At the top of the hierarchy, very broad contexts are established and, at the bottom, detailed technical activities, performed by the relevant engineering discipline (e.g., hardware or software engineering), are conducted.

Stated in a slightly more formal manner, the world view (WV) is composed of a set of domains (D_i), which can each be a system or system of systems in its own right.

$$WV = \{D_1, D_2, D_3, \dots, D_n\}$$

Each domain is composed of specific elements (E_j) each of which serves some role in accomplishing the objective and goals of the domain or component:

$$D_i = \{E_1, E_2, E_3 \dots E_m\}$$

Finally, each element is implemented by specifying the technical components (C_k) that achieve the necessary function for an element:

$$E_j = \{C_1, C_2, C_3 \dots C_k\}$$

In the software context, a component could be a computer program, a reusable program component, a module, a class or object, or even a programming language statement.

It is important to note that the system engineer narrows the focus of work as he or she moves downward in the hierarchy just described. However, the world view portrays a clear definition of overall functionality that will enable the engineer to understand the domain, and ultimately the system or product, in the proper context.

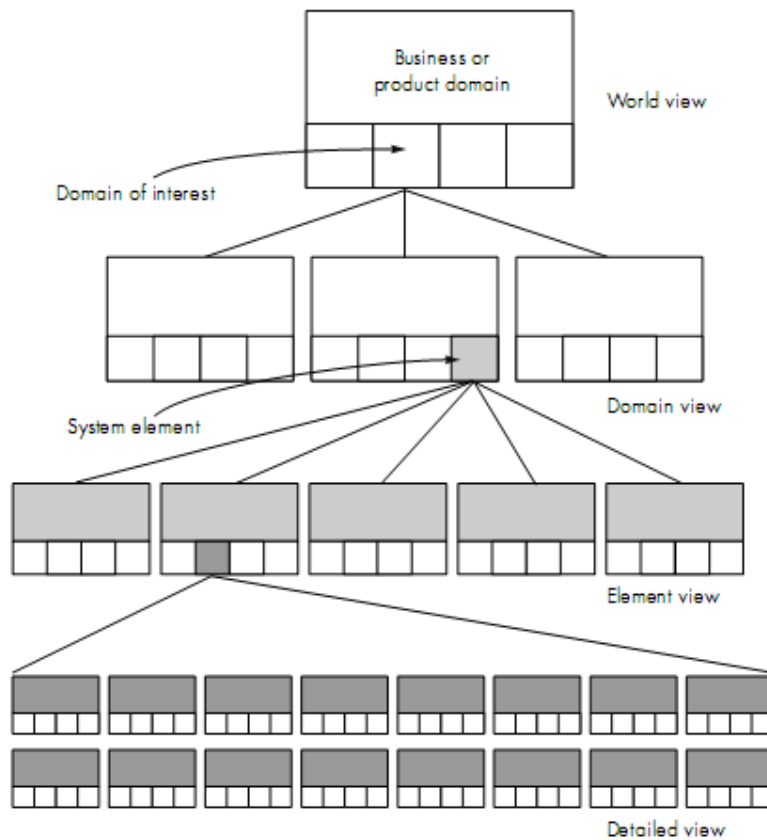


FIGURE 2.1 the system engineering hierarchy

2.3.1 System Modeling

System engineering is a modeling process. Whether the focus is on the world view or the detailed view, the engineer creates models that

- Define the processes that serve the needs of the view under consideration.
- Represent the behavior of the processes and the assumptions on which the behavior is based.
- Explicitly define both exogenous and endogenous input³ to the model.
- Represent all linkages (including output) that will enable the engineer to better understand the view.

To construct a system model, the engineer should consider a number of restraining factors:

1. Assumptions that reduce the number of possible permutations and variations, thus enabling a model to reflect the problem in a reasonable manner. As an example, consider a three-dimensional rendering product used by the entertainment industry to create realistic animation. One domain of the product enables the representation of 3D human forms. Input to this domain encompasses the ability to specify movement from a live human actor, from video, or by the creation of graphical models. The system engineer makes certain assumptions about the range of allowable human movement (e.g., legs cannot be wrapped around the torso) so that the range of inputs and processing can be limited.

2. Simplifications that enable the model to be created in a timely manner. To illustrate, consider an office products company that sells and services a broad range of copiers, faxes, and related equipment. The system engineer is modeling the needs of the service organization and is working to understand the flow of information that spawns a service order. Although a service order can be derived from many origins, the engineer categorizes only two sources: internal demand and external request. This enables a simplified partitioning of input that is required to generate the service order.

3. Limitations that help to bound the system. For example, an aircraft avionics system is being modeled for a next generation aircraft. Since the aircraft will be a two-engine design, the monitoring domain for propulsion will be modeled to accommodate a maximum of two engines and associated redundant systems.

4. Constraints that will guide the manner in which the model is created and the approach taken when the model is implemented. For example, the technology infrastructure for the three-dimensional rendering system described previously is a single G4-based processor. The computational complexity of problems must be constrained to fit within the processing bounds imposed by the processor.

5. Preferences that indicate the preferred architecture for all data, functions, and technology. The preferred solution sometimes comes into conflict with other restraining factors. Yet, customer satisfaction is often predicated on the degree to which the preferred approach is realized.

The resultant system model (at any view) may call for a completely automated solution, a semi-automated solution, or a non-automated approach. In fact, it is often possible to characterize models of each type that serve as alternative solutions to the problem at hand. In essence, the system engineer simply modifies the relative influence of different system elements (people, hardware, software) to derive models of each type.

2.3.2 System Simulation

In the late 1960s, R. M. Graham made a distressing comment about the way we build computer-based systems: "We build systems like the Wright brothers built airplanes build the whole thing, push it off a cliff, let it crash, and start over again." In fact, for at least one class of system the reactive system we continue to do this today.

Many computer-based systems interact with the real world in a reactive fashion. That is, real-world events are monitored by the hardware and software that form the computer-based system, and based on these events; the system imposes control on the machines, processes, and even people who cause the events to occur. Real-time and embedded systems often fall into the reactive systems category.

Unfortunately, the developers of reactive systems sometimes struggle to make them perform properly. Until recently, it has been difficult to predict the performance, efficiency, and behavior of such systems prior to building them. In a very real sense, the construction of many real-time systems was an adventure in "flying." Surprises (most of them unpleasant) were not discovered until the system was built and "pushed off a cliff." If the system crashed due to incorrect function, inappropriate behavior, or poor performance, we picked up the pieces and started over again.

Many systems in the reactive category control machines and/or processes (e.g., commercial aircraft or petroleum refineries) that must operate with an extremely high degree of reliability. If the system fails, significant economic or human loss could occur. For this reason, the approach described by Graham is both painful and dangerous.

Today, software tools for system modeling and simulation are being used to help to eliminate surprises when reactive, computer-based systems are built. These tools are applied during the system engineering process, while the role of hardware and software, databases and people is being specified. Modeling and simulation tools enable a system engineer to "test drive" a specification of the system.

2.4 Business process engineering: an overview

The goal of business process engineering (BPE) is to define architectures that will enable a business to use information effectively. Michael Guttman describes the challenge when he states:

However, the price for this change is largely borne by the IT [information technology] organizations that must support this polyglot configuration. Today, each IT organization must become, in effect, its own systems integrator and architect. It must design, implement, and support its own unique configuration of heterogeneous computing resources, distributed logically and geographically throughout the enterprise, and connected by an appropriate enterprise-wide networking scheme.

Moreover, this configuration can be expected to change continuously, but unevenly, across the enterprise, due to changes in business requirements and in computing technology. These diverse and incremental changes must be coordinated across a distributed environment consisting of hardware and software supplied by dozens, if not hundreds, of vendors.

And, of course, we expect these changes to be seamlessly incorporated without disrupting normal operations and to scale gracefully as those operations expand.

When taking a world view of a company's information technology needs, there is little doubt that system engineering is required. Not only is the specification of the appropriate computing architecture required, but the software architecture that populates the "unique configuration of heterogeneous computing resources" must be developed. Business process engineering is one approach for creating an overall plan for implementing the computing architecture.

Three different architectures must be analyzed and designed within the context of business objectives and goals:

- Data architecture
- Applications architecture
- Technology infrastructure

The data architecture provides a framework for the information needs of a business or business function. The individual building blocks of the architecture are the data objects that are used by the business. A data object contains a set of attributes that define some aspect, quality, characteristic, or descriptor of the data that are being described. For example, an information engineer might define the data object customer. To more fully describe customer, the following attributes are defined:

Object: Customer

Attributes:

- name
- company name
- job classification and purchase authority
- business address and contact information
- product interest(s)
- past purchase(s)
- date of last contact
- status of contact

Once a set of data objects is defined, their relationships are identified. A relationship indicates how objects are connected to one another. As an example, consider the objects: customer, and product A. The two objects can be connected by the relationship purchases; that is, a customer purchases product A or product A is purchased by a customer. The data objects (there may be hundreds or even thousands for a major business activity) flow between business functions, are organized within a database, and are transformed to provide information that serves the needs of the business.

The application architecture encompasses those elements of a system that transform objects within the data architecture for some business purpose. In the context of this book, we consider the application architecture to be the system of programs (software) that performs this transformation. However, in a broader context, the application architecture might incorporate the role of people (who are information transformers and users) and business procedures that have not been automated.

The technology infrastructure provides the foundation for the data and application architectures. The infrastructure encompasses the hardware and software that are used to support the application and data. This includes computers, operating systems, networks, telecommunication links, storage technologies, and the architecture (e.g., client/server) that has been designed to implement these technologies. To model the system architectures described earlier, a hierarchy of business process engineering activities is defined. Referring to Figure 2.2, the world view is achieved through information strategy planning (ISP). ISP views the entire business as an entity and isolates the domains of the business (e.g., engineering, manufacturing, marketing, finance, sales) that are important to the overall enterprise. ISP defines the data objects that are visible at the enterprise level, their relationships, and how they flow between the business domains.

The domain view is addressed with a BPE activity called business area analysis (BAA). Hares describe BAA in the following manner:

BAA is concerned with identifying in detail data (in the form of entity [data object] types) and function requirements (in the form of processes) of

selected business areas [domains] identified during ISP and ascertaining their interactions (in the form of matrices). It is only concerned with specifying what is required in a business area.

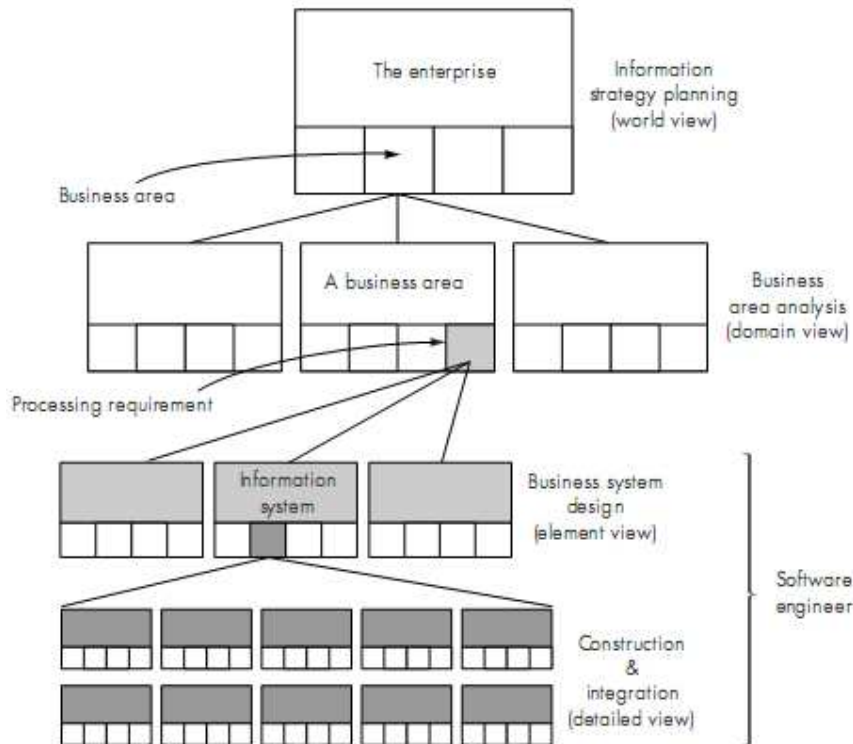


FIGURE 2.2 the business process engineering hierarchy

As the system engineer begins BAA, the focus narrows to a specific business domain. BAA views the business area as an entity and isolates the business functions and procedures that enable the business area to meet its objectives and goals. BAA, like ISP, defines data objects, their relationships, and how data flow. But at this level, these characteristics are all bounded by the business area being analyzed. The outcome of BAA is to isolate areas of opportunity in which information systems may support the business area.

Once an information system has been isolated for further development, BPE makes a transition into software engineering. By invoking a business system design (BSD) step, the basic requirements of a specific information system are modeled and these requirements are translated into data architecture, applications architecture, and technology infrastructure.

The final BPE step construction and integration focuses on implementation detail. The architecture and infrastructure are implemented by constructing an appropriate database and internal data structures, by building applications using software components, and by selecting appropriate elements of a technology infrastructure to support the design created during BSD. Each of

these system components must then be integrated to form a complete information system or application. The integration activity also places the new information system into the business area context, performing all user training and logistics support to achieve a smooth transition.

2.5 Product engineering: an overview

The goal of product engineering is to translate the customer's desire for a set of defined capabilities into a working product. To achieve this goal, product engineering like business process engineering must derive architecture and infrastructure. The architecture encompasses four distinct system components: software, hardware, data (and databases), and people. A support infrastructure is established and includes the technology required to tie the components together and the information (e.g., documents, CD-ROM, video) that is used to support the components.

Referring to Figure 2.3, the world view is achieved through requirements engineering. The overall requirements of the product are elicited from the customer. These requirements encompass information and control needs, product function and behavior, overall product performance, design and interfacing constraints, and other special needs. Once these requirements are known, the job of requirements engineering is to allocate function and behavior to each of the four components noted earlier.

Once allocation has occurred, system component engineering commences. System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering. Each of these engineering disciplines takes a domain - specific view, but it is important to note that the engineering disciplines must establish and maintain active communication with one another. Part of the role of requirements engineering is to establish the interfacing mechanisms that will enable this to happen.

The element view for product engineering is the engineering discipline itself applied to the allocated component. For software engineering, this means analysis and design modeling activities (covered in detail in later chapters) and construction and integration activities that encompass code generation, testing, and support steps. The analysis step models allocated requirements into representations of data, function, and behavior. Design maps the analysis model into data, architectural, interface, and software component-level designs.

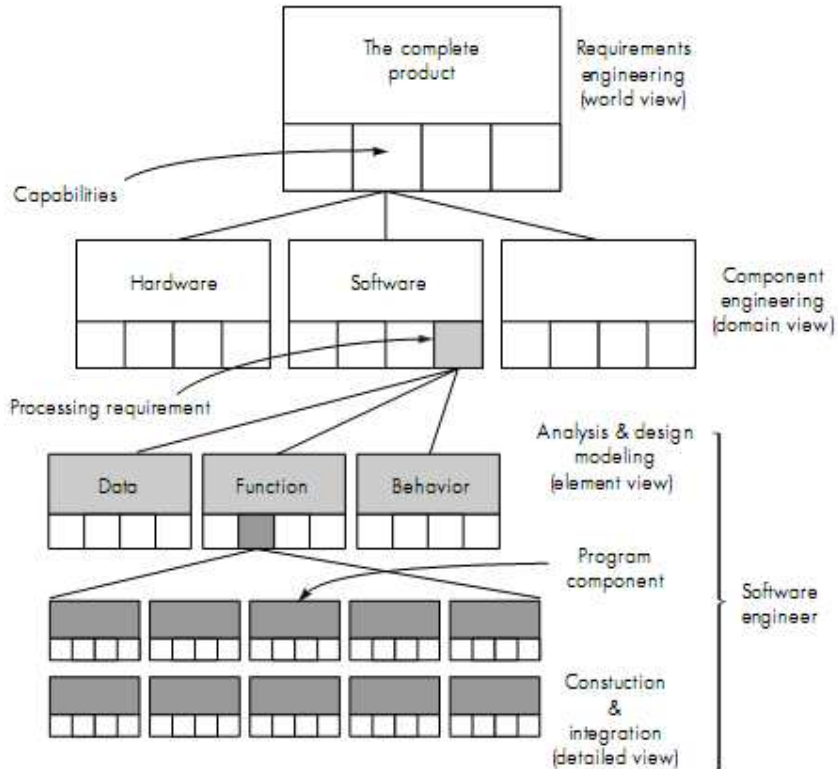


FIGURE 2.3 The product engineering hierarchy

2.6 Requirements engineering

The outcome of the system engineering process is the specification of a computer based system or product at the different levels described generically in Figure 2.1. But the challenge facing system engineers (and software engineers) is profound: How can we ensure that we have specified a system that properly meets the customer's needs and satisfies the customer's expectations? There is no foolproof answer to this difficult question, but a solid requirements engineering process is the best solution we currently have.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. The requirements engineering process can be described in five distinct steps:

- Requirements elicitation
- Requirements analysis and negotiation
- Requirements specification
- System modeling

- Requirements validation
- Requirements management

2.6.1 Requirements Elicitation

It certainly seems simple enough ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple it's very hard. Christel and Kang identify a number of problems that help us understand why requirements elicitation is difficult:

- Problems of scope. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- Problems of understanding. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
- Problems of volatility. The requirements change over time.

To help overcome these problems, system engineers must approach the requirements gathering activity in an organized manner. Sommerville and Sawyer suggest a set of detailed guidelines for requirements elicitation, which are summarized in the following steps:

- Assess the business and technical feasibility for the proposed system.
- Identify the people who will help specify requirements and understand their organizational bias.
- Define the technical environment (e.g., computing architecture, operating system, telecommunications needs) into which the system or product will be placed.
- Identify "domain constraints" (i.e., characteristics of the business environment specific to the application domain) that limit the functionality or performance of the system or product to be built.
- Define one or more requirements elicitation methods (e.g., interviews, focus groups, team meetings).

- Solicit participation from many people so that requirements are defined from different points of view; be sure to identify the rationale for each requirement that is recorded.
- Identify ambiguous requirements as candidates for prototyping.
- Create usage scenarios to help customers/users better identify key requirements.

The work products produced as a consequence of the requirements elicitation activity will vary depending on the size of the system or product to be built. For most systems, the work products include

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in the requirements elicitation activity.
- A description of the system's technical environment.
- A list of requirements (preferably organized by function) and the domain constraints those apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in the requirements elicitation.

2.6.2 Requirements Analysis and Negotiation

Once requirements have been gathered, the work products noted earlier form the basis for requirements analysis. Analysis categorizes requirements and organizes them into related subsets; explores each requirement in relationship to others; examines requirements for consistency, omissions, and ambiguity; and ranks requirements based on the needs of customers/users. As the requirements analysis activity commences, the following questions are asked and answered:

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?

- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?

It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It also is relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs".

The system engineer must reconcile these conflicts through a process of negotiation. Customers, users and stakeholders are asked to rank requirements and then discuss conflicts in priority. Risks associated with each requirement are identified and analyzed (see Chapter 6 for details). Rough estimates of development effort are made and used to assess the impact of each requirement on project cost and delivery time. Using an iterative approach, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

2.6.3 Requirements Specification

In the context of computer-based systems (and software), the term specification means different things to different people. A specification can be a written document, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. Some suggest that a "standard template" should be developed and used for a system specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

The System Specification is the final work product produced by the system and requirements engineer. It serves as the foundation for hardware engineering, software engineering, database engineering, and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification

bounds each allocated system element. The System Specification also describes the information (data and control) that is input to and output from the system.

2.6.4 System Modeling

Assume for a moment that you have been asked to specify all requirements for the construction of a gourmet kitchen. You know the dimensions of the room, the location of doors and windows, and the available wall space. You could specify all cabinets and appliances and even indicate where they are to reside in the kitchen.

Would this be a useful specification? The answer is obvious. In order to fully specify what is to be built, you would need a meaningful model of the kitchen, that is, a blueprint or three-dimensional rendering that shows the position of the cabinets and appliances and their relationship to one another. From the model, it would be relatively easy to assess the efficiency of work flow (a requirement for all kitchens), the aesthetic “look” of the room (a personal, but very important requirement).

We build system models for much the same reason that we would develop a blueprint or 3D rendering for the kitchen. It is important to evaluate the system’s components in relationship to one another, to determine how requirements fit into this picture, and to assess the “aesthetics” of the system as it has been conceived.

2.6.5 Requirements Validation

The work products produced as a consequence of requirements engineering (a system specification and related information) are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the formal technical review. The review team includes system engineers, customers, users, and other stakeholders who examine the system specification⁵ looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements. Although the requirements validation review can be conducted in any manner that results in the discovery of requirements errors, it is useful to examine each requirement against a set of checklist questions. The following questions represent a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified. Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the system specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with system performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Checklist questions like these help ensure that the validation team has done everything possible to conduct a thorough review of each requirement.

2.6.6 Requirements Management

In the preceding chapter, we noted that requirements for computer-based systems change and that the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds.

Like SCM, requirements management begins with identification. Each requirement is assigned a unique identifier that might take the form

<requirement type><requirement>

where requirement type takes on values such as F = functional requirement, D = data requirement, B = behavioral requirement, I = interface requirement, and P = output requirement. Hence, a requirement identified as F09 indicates a functional requirement assigned requirement number 9.

Once requirements have been identified, traceability tables are developed. Shown schematically in Figure 2.4, each traceability table relates identified requirements to one or more aspects of the system or its environment. Among many possible traceability tables are the following:

Features traceability table: Shows how requirements relate to important customer observable system/product features.

Source traceability table: Identifies the source of each requirement.

Dependency traceability table: Indicates how requirements are related to one another.

Subsystem traceability table: Categorizes requirements by the subsystem(s) that they govern.

Interface traceability table: Shows how requirements relate to both internal and external system interfaces.

Requirement	Specific aspect of the system or its environment						
	A01	A02	A03	A04	A05		Aii
R01			✓		✓		
R02	✓		✓				
R03	✓			✓			✓
R04		✓			✓		
R05	✓	✓		✓			✓
Rnn	✓		✓				

FIGURE 2.4 Generic traceability table

In many cases, these traceability tables are maintained as part of a requirements database so that they may be quickly searched to understand how a change in one requirement will affect different aspects of the system to be built.

2.7 System Modeling

Every computer-based system can be modeled as information transforms using an input-processing-output template. Hatley and Pirbhai have extended this view to include two additional system features user interface processing and maintenance and self-test processing. Although these additional features are not present for every computer-based system, they are very common,

and their specification makes any system model more robust. Using a representation of input, processing, output, user interface processing, and self-test processing, a system engineer can create a model of system components that sets a foundation for later steps in each of the engineering disciplines.

To develop the system model, a system model template is used. The system engineer allocates system elements to each of five processing regions within the template:

- (1) user interface,
- (2) input,
- (3) system function and control,
- (4) output, and
- (5) maintenance and self-test.

The format of the architecture template is shown in Figure 2.5. Like nearly all modeling techniques used in system and software engineering, the system model template enables the analyst to create a hierarchy of detail. A system context diagram (SCD) resides at the top level of the hierarchy. The context diagram "establishes the information boundary between the system being implemented and the environment in which the system is to operate". That is, the SCD defines all external producers of information used by the system, all external consumers of information created by the system, and all entities that communicate through the interface or perform maintenance and self-test.

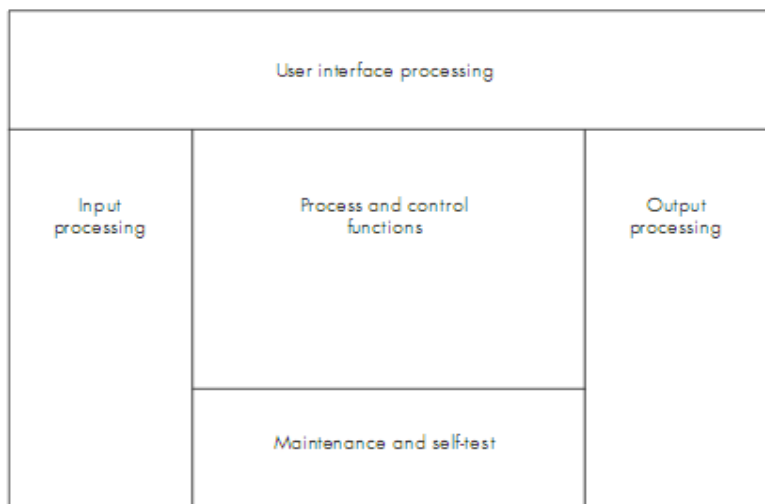


FIGURE 2.5 System model template

To illustrate the use of the SCD, consider the conveyor line sorting system. The system engineer is presented with the following (some what nebulous) statement of objectives for CLSS:

CLSS must be developed such that boxes moving along a conveyor line will be identified and sorted into one of six bins at the end of the line. The boxes will pass by a sorting station where they will be identified. Based on an identification number printed on the side of the box (an equivalent bar code is provided), the boxes will be shunted into the appropriate bins. Boxes pass in random order and are evenly spaced. The line is moving slowly.

For this example, CLSS is extended and makes use of a personal computer at the sorting station site. The PC executes all CLSS software, interacts with the bar code reader to read part numbers on each box, interacts with the conveyor line monitoring equipment to acquire conveyor line speed, stores all part numbers sorted, interacts with a sorting station operator to produce a variety of reports and diagnostics, sends control signals to the shunting hardware to sort the boxes, and communicates with a central factory automation mainframe. The SCD for CLSS (extended) is shown in Figure 2.6.

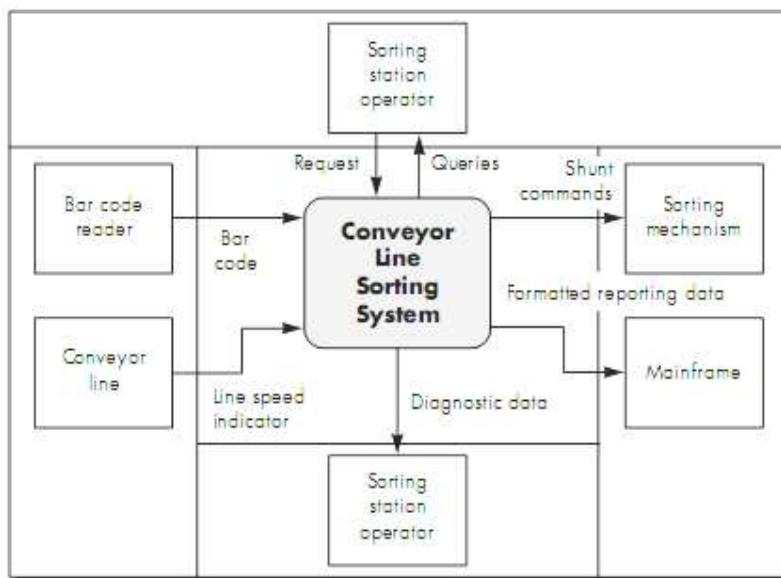


FIGURE 2.6 System context diagram for CLSS

Each box shown in Figure 2.6 represents an external entity that is, a producer or consumer of system information. For example, the bar code reader produces information that is input to the CLSS system. The symbol for the entire system (or, at lower levels, major subsystems) is a rectangle with rounded corners. Hence, CLSS is represented in the processing and control region at the center of the SCD. The labeled arrows shown in the SCD represent information (data and control) as it moves from the external environment into the CLSS system. The external entity bar code reader produces input information that is labeled bar code. In essence, the SCD places any system into the context of its external environment.

The system engineer refines the system context diagram by considering the shaded rectangle in Figure 2.6 in more detail. The major subsystems that enable the conveyor line sorting system to function within the context defined by the SCD are identified. Referring to Figure 2.7, the major subsystems are defined in a system flow diagram (SFD) that is derived from the SCD. Information flow across the regions of the SCD is used to guide the system engineer in developing the SFD a more detailed "schematic" for CLSS. The system flow diagram shows major subsystems and important lines of information (data and control) flow. In addition, the system template partitions the subsystem processing into each of the five regions discussed earlier. At this stage, each of the subsystems can contain one or more system elements (e.g., hardware, software, people) as allocated by the system engineer.

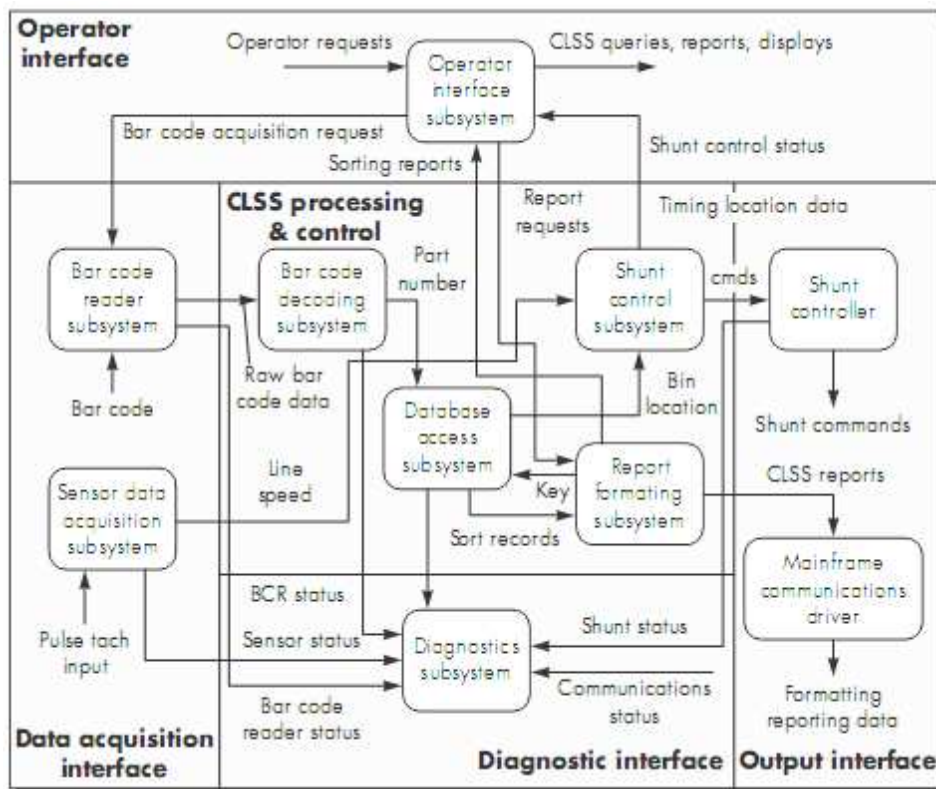


FIGURE 2.7 System flow diagram for CLSS

The initial system flow diagram becomes the top node of a hierarchy of SFDs. Each rounded rectangle in the original SFD can be expanded into another architecture template dedicated solely to it. This process is illustrated schematically in Figure 2.8. Each of the SFDs for the system can be used as a starting point for subsequent engineering steps for the subsystem that has been described.

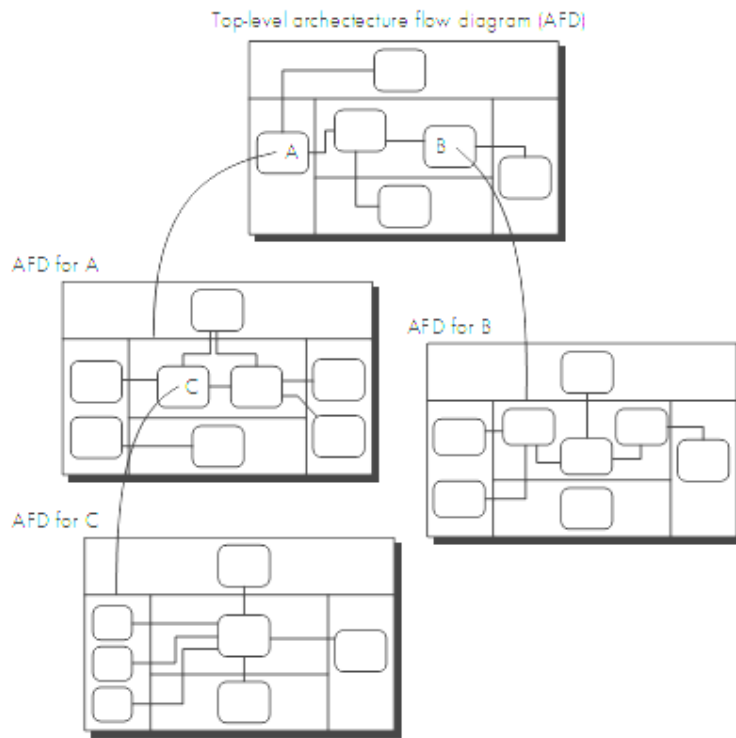


FIGURE 2.8 Building an SFD hierarchy

3. Requirement Engineering

3.1 Introduction

In the design of software, the first step is to decide about the objectives of software. This is the most difficult aspect of software design. These objectives, which the software is supposed to fulfill, are called requirements.

The IEEE definition of requirement is:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system component, to satisfy a contract formally imposed document.
3. A documented representation of a condition or capability as in (1) or (2).

Thus, requirements specify “what the system is supposed to do?” These requirements are taken from the user. Defining the requirements is most elementary & most difficult part of system design, because, at this level, sometimes, the user himself is not clear about it. Many software projects have failed due to certain requirements specification issues. Thus, overall quality of software product is dependent on this aspect. Identifying, defining, and analyzing the requirements is known as requirements analysis. Requirements analysis includes the following activities:

1. Identification of end user’s need.
2. Preparation of a corresponding document called SRS (Software Requirements Specification).
3. Analysis and validation of the requirements document to ensure consistency, completeness and feasibility.
4. Identification of further requirements during the analysis of mentioned requirements.

3.2 Requirements engineering tasks

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design (Figure 3.1). Requirements engineering activities result in the specification of software’s operational characteristics (function, data, and behavior), indicate software’s interface with other system elements, and establish constraints that software must meet. Requirements analysis allows the software engineer (sometimes called analyst in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software. Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs. Finally, the requirements specification provides the developer and the

customer with the means to assess quality once software is built. Software requirements analysis may be divided into five areas of effort:

- (1) Problem recognition,
- (2) Evaluation and synthesis,
- (3) Modeling,
- (4) Specification, and
- (5) Review.

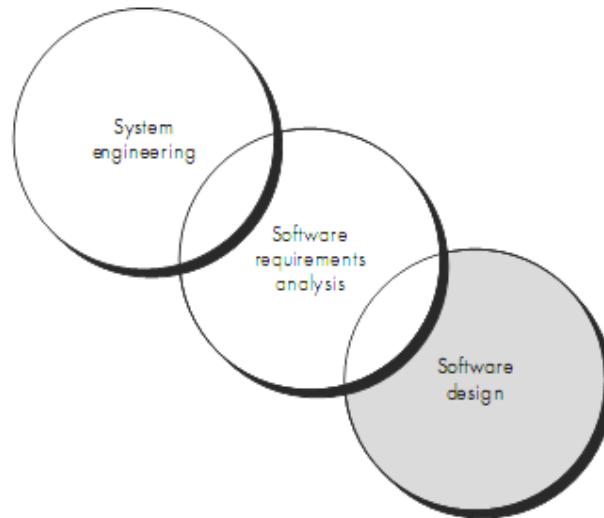


FIGURE 3.1 Analysis as a bridge between system engineering and software design

Initially, the analyst studies the System Specification (if one exists) and the Software Project Plan. It is important to understand software in a system context and to review the software scope that was used to generate planning estimates. Next, communication for analysis must be established so that problem recognition is ensured. The goal is recognition of the basic problem elements as perceived by the customer/users. Problem evaluation and solution synthesis is the next major area of effort for analysis. The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behavior in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints. Each of these tasks serves to describe the problem so that an overall approach or solution may be synthesized.

For example, an inventory control system is required for a major supplier of auto parts. The analyst finds that problems with the current manual system include (1) inability to obtain the status of a component rapidly, (2) two- or three-day turnaround to update a card file, (3) multiple reorders to the same vendor because there is no way to associate vendors with components, and so forth. Once problems have been identified, the analyst determines what

information is to be produced by the new system and what data will be provided to the system. For instance, the customer desires a daily report that indicates what parts have been taken from inventory and how many similar parts remain. The customer indicates that inventory clerks will log the identification number of each part as it leaves the inventory area.

Upon evaluating current problems and desired information (input and output), the analyst begins to synthesize one or more solutions. To begin, the data objects, processing functions, and behavior of the system are defined in detail. Once this information has been established, basic architectures for implementation are considered. A client/server approach would seem to be appropriate, but does the software to support this architecture fall within the scope outlined in the Software Plan? A database management system would seem to be required, but is the user/customer's need for associativity justified? The process of evaluation and synthesis continues until both analyst and customer feels confident that software can be adequately specified for subsequent development steps.

Throughout evaluation and solution synthesis, the analyst's primary focus is on "what," not "how." What data does the system produce and consume, what functions must the system perform, what behaviors does the system exhibit. During the evaluation and solution synthesis activity, the analyst creates models of the system in an effort to better understand data and control flow, functional processing, operational behavior, and information content. The model serves as a foundation for software design and as the basis for the creation of specifications for the software.

The customer may be unsure of precisely what is required. The developer may be unsure that a specific approach will properly accomplish function and performance. For these, and many other reasons, an alternative approach to requirements analysis, called prototyping, may be conducted.

3.3 Initiating the Requirements Engineering Process

Due to the complexity involved in software development, the engineering approach is being used in software design. Use of engineering approach in the area of requirements analysis evolved the field of Requirements Engineering.

3.3.1 Requirements Engineering

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

The output of requirements of engineering process is Requirements Definition Description (RDD). Requirements engineering may be defined in the context of Software Engineering. It divides the Requirements Engineering into two categories. First is the requirements definition and the second is requirements management. Requirements definition consists of the following processes:

1. Requirements gathering.
2. Requirements analysis and modeling.
3. Creation of RDD and SRS.
4. Review and validation of SRS as well as obtaining confirmation from user.

Requirements management consists of the following processes:

1. Identifying controls and tracking requirements.
2. Checking complete implementation of RDD.
3. Manage changes in requirements which are identified later.

3.3.2 Types of Requirements

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

1. Those that should be absolutely met.
2. Those that is highly desirable but not necessary.
3. Those that are possible but could be eliminated.

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

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

3.3.3 Software Requirements Specification (SRS)

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

- External Interfaces of the system: They identify the information which is to flow 'from and to' to the system.
- Functional and non-functional requirements of the system. They stand for the finding of run time requirements.
- Design constraints:

The SRS outline is given below:

1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions, acronyms, and abbreviations
 - 1.4 References
 - 1.5 Overview
2. Overall description
 - 2.1 Product perspective
 - 2.2 Product functions
 - 2.3 User characteristics
 - 2.4 Constraints
 - 2.5 Assumptions and dependencies

3. Specific requirements

- 3.1 External Interfaces
- 3.2 Functional requirements
- 3.3 Performance requirements
- 3.4 Logical Database requirements
- 3.5 Design Constraints
- 3.6 Software system attributes
- 3.7 Organizing the specific requirements
- 3.8 Additional Comments

4. Supporting information

- 4.1 Table of contents and index
- 4.2 Appendixes

3.3.4 Problems in SRS

There are various features that make requirements analysis difficult. These are discussed below:

1. Complete requirements are difficult to uncover. In recent trends in engineering, the processes are automated and it is practically impossible to understand the complete set of requirements during the commencement of the project itself.
2. Requirements are continuously generated. Defining the complete set of requirements in the starting is difficult. When the system is put under run, the new requirements are obtained and need to be added to the system. But, the project schedules are seldom adjusted to reflect these modifications. Otherwise, the development of software will never commence.
3. The general trends among software developer shows that they have over dependence on CASE tools. Though these tools are good helping agents,

over reliance on these Requirements Engineering Tools may create false requirements. Thus, the requirements corresponding to real system should be understood and only a realistic dependence on tools should be made.

4. The software projects are generally given tight project schedules. Pressure is created from customer side to hurriedly complete the project. This normally cuts down the time of requirements analysis phase, which frequently lead to disaster(s).

5. Requirements Engineering is communication intensive. Users and developers have different vocabularies, professional backgrounds and psychology. User writes specifications in natural language and developer usually demands precise and well-specified requirement.

6. In present time, the software development is market driven having high commercial aspect. The software developed should be a general purpose one to satisfy anonymous customer, and then, it is customized to suit a particular application.

7. The resources may not be enough to build software that fulfils all the customer's requirements. It is left to the customer to priorities the requirements and develop software fulfilling important requirements.

3.3.5 Requirements Gathering Tools

The requirement gathering is an art. The person who gathers requirements should have knowledge of what and when to gather information and by what resources. The requirements are gathered regarding organisation, which include information regarding its policies, objectives, and organisation structure, regarding user staff. It includes the information regarding job function and their personal details, regarding the functions of the organisation including information about work flow, work schedules and working procedure.

The following four tools are primarily used for information gathering:

1. Record review: A review of recorded documents of the organisation is performed. Procedures, manuals, forms and books are reviewed to see format and functions of present system. The search time in this technique is more.

2. On site observation: In case of real life systems, the actual site visit is performed to get a close look of system. It helps the analyst to detect the problems of existing system.

3. Interview: A personal interaction with staff is performed to identify their requirements. It requires experience of arranging the interview, setting the stage, avoiding arguments and evaluating the outcome.

4. Questionnaire: It is an effective tool which requires less effort and produces a written document about requirements. It examines a large number of respondents simultaneously and gets customized answers. It gives person sufficient time to answer the queries and give correct answers.

3.4 Eliciting Requirements

Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help. Communication has begun. But, as we have already noted, the road from communication to understanding is often full of potholes.

3.4.1 Initiating the Process

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

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

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

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

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

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

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

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

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

3.4.2 Facilitated Application Specification Techniques

Too often, customers and software engineers have an unconscious "us and them" mind-set. Rather than working as a team to identify and refine requirements, each constituency defines its own "territory" and communicates through series of memos, formal position papers, documents, and question and answer sessions. History has shown that this approach doesn't work very well. Misunderstandings abound, important information is omitted, and a successful working relationship is never established.

It is with these problems in mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called facilitated application specification techniques (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements. FAST has been used predominantly by the information systems community, but the technique offers potential for improved communication in applications of all kinds.

Many different approaches to FAST have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- A meeting is conducted at a neutral site and attended by both software engineers and customers.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

To better understand the flow of events as they occur in a typical FAST meeting, we present a brief scenario that outlines the sequence of events that lead up to the meeting, occur during the meeting, and follow the meeting. Initial meetings between the developer and customer occur and basic questions and answers help to establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customer write a one- or two-page "product request." A meeting place, time, and date for FAST are selected and a facilitator is chosen. Attendees from both the development and customer/user organizations are invited to attend. The product request is distributed to all attendees before the meeting date.

While reviewing the request in the days before the meeting, each FAST attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

As an example, assume that a FAST team working for a consumer products company has been provided with the following product description: Our research indicates that the market for home security systems is growing at a rate of 40 percent per year. We would like to enter this market by building a microprocessor-based home security system that would protect against and/or recognize a variety of undesirable "situations" such as illegal entry,

fire, flooding, and others. The product, tentatively called Safe Home, will use appropriate sensors to detect each situation, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

In reality, considerably more information would be provided at this stage. But even with additional information, ambiguity would be present, omissions would likely exist, and errors might occur. For now, the preceding "product description" will suffice. The FAST team is composed of representatives from marketing, software and hardware engineering, and manufacturing. An outside facilitator is to be used.

Each person on the FAST team develops the lists described previously. Objects described for Safe Home might include smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a control panel, a display, telephone numbers, a telephone call, and so on. The list of services might include setting the alarm, monitoring the sensors, dialing the phone, programming the control panel, reading the display (note that services act on objects). In a similar fashion, each FAST attendee will develop lists of constraints (e.g., the system must have a manufactured cost of less than \$80, must be user-friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, an event priority scheme should be implemented).

As the FAST meeting begins, the first topic of discussion is the need and justification for the new product everyone should agree that the product is justified. Once agreement has been established, each participant presents his or her lists for discussion. The lists can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive backed sheets, or written on a wall board. Alternatively, the lists may have been posted on an electronic bulletin board or posed in a chat room environment for review prior to the meeting. Ideally, each list entry should be capable of being manipulated separately so that lists can be combined, entries can be deleted and additions can be made. At this stage, critique and debate are strictly prohibited.

After individual lists are presented in one topic area, a combined list is created by the group. The combined list eliminates redundant entries, adds any new ideas that come up during the discussion, but does not delete anything. After combined lists for all topic areas have been created, discussion coordinated by the facilitator ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product/system to be developed. The objective is to develop a consensus list in each topic area (objects, services, constraints, and performance). The lists are then set aside for later action.

Once the consensus lists have been completed, the team is divided into smaller subteams; each works to develop mini-specifications for one or more entries on each of the lists. Each mini-specification is an elaboration of the word or phrase contained on a list. For example, the mini-specification for the Safe Home object control panel might be

- mounted on wall
- size approximately 9 X 5 inches
- contains standard 12-key pad and special keys
- contains LCD display of the form shown in sketch [not presented here]
- all customer interaction occurs through keys
- used to enable and disable the system
- software provides interaction guidance, echoes, and the like
- connected to all sensors

Each subteam then presents each of its mini-specs to all FAST attendees for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An issues list is maintained so that these ideas will be acted on later.

After the mini-specs are completed, each FAST attendee makes a list of validation criteria for the product/system and presents his or her list to the team. A consensus list of validation criteria is then created. Finally, one or more participants (or outsiders) are assigned the task of writing the complete draft specification using all inputs from the FAST meeting.

FAST is not a panacea for the problems encountered in early requirements elicitation. But the team approach provides the benefits of many points of view, instantaneous discussion and refinement, and is a concrete step toward the development of a specification.

3.4.3 Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. Originally developed in Japan and first used at the Kobe Shipyard of Mitsubishi Heavy Industries, Ltd., in the early 1970s, QFD “concentrates on maximizing customer satisfaction from the software engineering process” To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements:

Normal requirements: The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be

requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements: These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements: These features go beyond the customer's expectations and prove to be very satisfying when present. For example, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.

In actuality, QFD spans the entire engineering process [AKA90]. However, many QFD concepts are applicable to the requirements elicitation activity. We present an overview of only these concepts (adapted for computer software) in the paragraphs that follow.

In meetings with the customer, function deployment is used to determine the value of each function that is required for the system. Information deployment identifies both the data objects and events that the system must consume and produce. These are tied to the functions. Finally, task deployment examines the behavior of the system or product within the context of its environment. Value analysis is conducted to determine the relative priority of requirements determined during each of the three deployments.

QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements called the customer voice table that is reviewed with the customer. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements.

3.4.4 Use-Cases

As requirements are gathered as part of informal meetings, FAST, or QFD, the software engineer (analyst) can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use-cases, provide a description of how the system will be used.

To create a use-case, the analyst must first identify the different types of people (or devices) that use the system or product. These actors actually represent roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself.

It is important to note that an actor and a user are not the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system. Primary actors interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work. Once actors have been identified, use-cases can be developed. The use-case describes the manner in which an actor interacts with the system. Jacobson suggests a number of questions that should be answered by the use-case:

- What main tasks or functions are performed by the actor?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

In general, a use-case is simply a written narrative that describes the role of an actor as interaction with the system occurs.

Each use-case provides an unambiguous scenario of interaction between an actor and the software. It can also be used to specify timing requirements or other constraints for the scenario. For example, in the use-case just noted, requirements indicate that activation occurs 30 seconds after the stay or away key is hit. This information can be appended to the use-case.

Use-cases describe scenarios that will be perceived differently by different actors. Wyder suggests that quality function deployment can be used to develop a weighted priority value for each use-case. To accomplish this, use-cases are evaluated from the point of view of all actors defined for the system. A priority value is assigned to each use-case (e.g., a value from 1 to 10) by each of the actors. An average priority is then computed, indicating the perceived importance of each of the use-cases. When an iterative

process model is used for software engineering, the priorities can influence which system functionality is delivered first.

3.5 Building Analysis Model

Before the actual system design commences, the system architecture is modeled. In this section, we discuss various modeling techniques.

3.5.1 Elementary Modeling Techniques

A model showing bare minimum requirements is called Essential Model. It has two components.

1. Environmental model: It indicates environment in which system exists. Any big or small system is a sub-system of a larger system. For example, if software is developed for a college, then college will be part of University. If it is developed for University, the University will be part of national educational system. Thus, when the model of the system is made these external interfaces are defined. These interfaces reflect system's relationship with external universe (called environment). The environment of a college system is shown in Figure 3.2.

In environmental model, the interfaces should clearly indicate the inflow and outflow of information from the system.

The tools of environment model are:

- (i) Statement of purpose: It indicates the basic objectives of system.
- (ii) Event list: It describes the different events of system and indicates functionality of the system.
- (iii) Context diagram: It indicates the environment of various sub-systems.

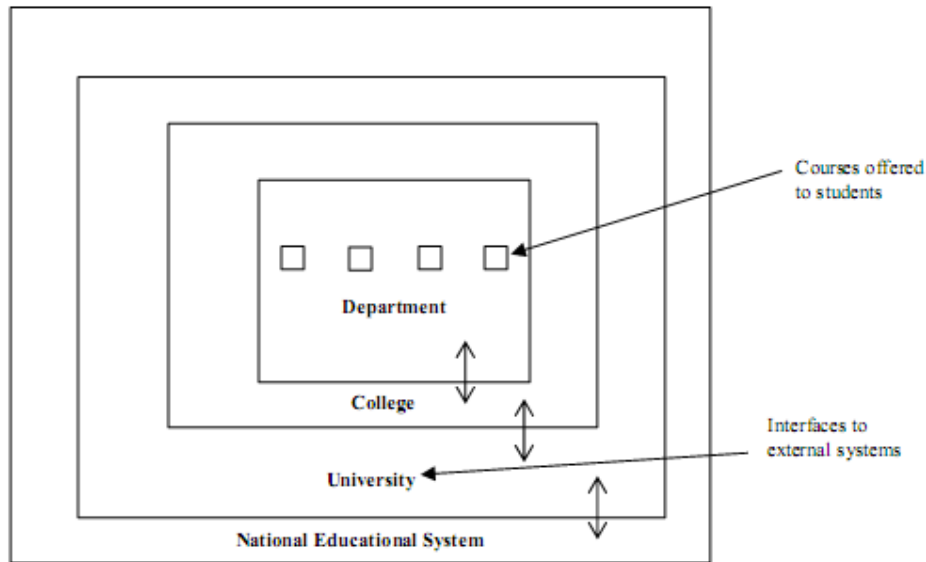


FIGURE 3.2 Environmental model of educational system

2. Behavioral Model: It describes operational behavior of the system. In this model, various operations of the system are represented in pictorial form. The tools used to make this model are: Data Flow Diagrams (DFD), E-R diagrams, Data Dictionary & Process Specification. These are discussed in later sections. Hence, behavioral model defines:

Data of proposed system:

- (i) The internal functioning of proposed system,
- (ii) Inter-relationship between various data.

In traditional approach of modeling, the analysts collect great deal of relatively unstructured data through data gathering tools and organize the data through system flow charts which support future development of system and simplify communication with the user. But, flow chart technique develops physical rather than logical system.

In structured approach of modeling the standard techniques of DFD, E-R diagrams etc. are used to develop system specification in a formal format. It develops a system logical model.

3.5.2 Data Flow Diagrams (DFD)

It is a graphical representation of flow of data through a system. It pictures a system as a network of functional processes. The basis of DFD is a data flow graph, which pictorially represents transformation on data as shown in Figure 3.3.

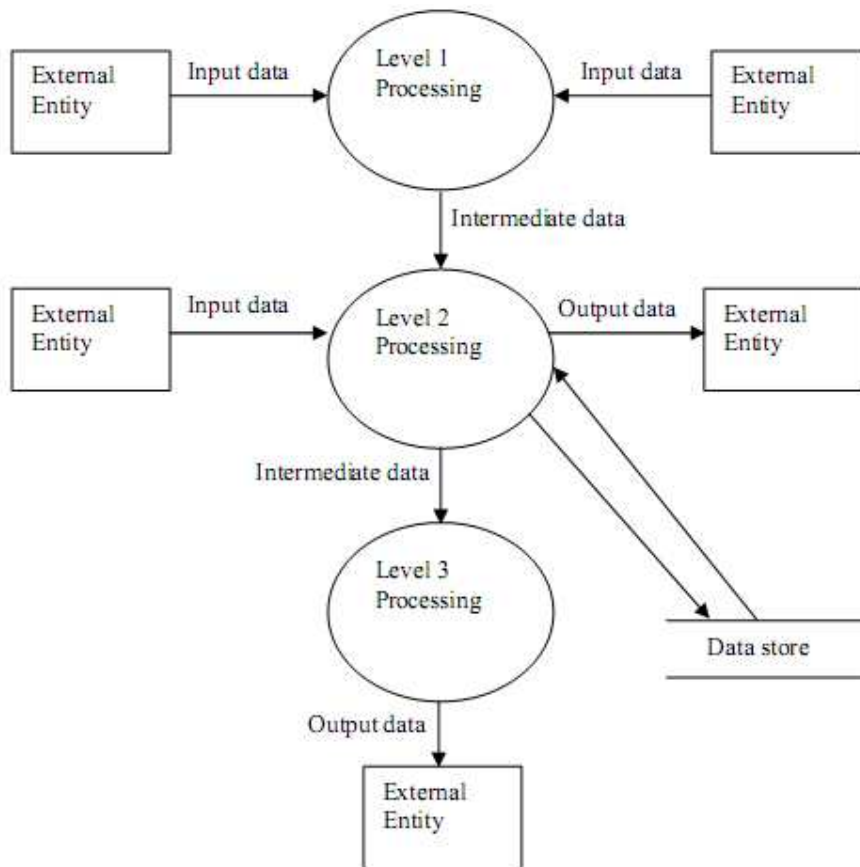


FIGURE 3.3 Data flow diagram

In this diagram, the external entities provide input data for the processing. During the processing, some intermediate data is generated. After final processing, the final output data is generated. The data store is the repository of data.

The structured approach of system design requires extensive modeling of the system. Thus, instead of making a complete model exhibiting the functionality of system, the DFD's are created in a layered manner. At the first layer, the DFD is made at block level and in lower layers, the details are shown. Thus, level "0" DFD makes a fundamental system (Figure 3.4).



FIGURE 3.4 Layer 1 depiction of process A

DFD's can represent the system at any level of abstraction. DFD of "0" level views entire software element as a single bubble with indication of only input

and output data. Thus, "0" level DFD is also called as Context diagram. Its symbols are shown in Figure 3.5.





Symbol	Name	Description
	Data Flow	Represents the connectivity between various processes
	Process	Performs some processing of input data
	External Entity	Defines source or destination of system data. The entity which receives or supplies information.
	Data Store	Repository of data

FIGURE 3.5 Symbols of a data flow diagram

3.5.3 Rules for making DFD

The following factors should be considered while making DFDs:

1. Keep a note of all the processes and external entities. Give unique names to them. Identify the manner in which they interact with each other.
2. Do numbering of processes.
3. Avoid complex DFDs (if possible).
4. The DFD should be internally consistent.
5. Every process should have minimum of one input and one output.

The data store should contain all the data elements that flow as input and output.

To understand the system functionality, a system model is developed. The developed model is used to analyze the system. The following four factors are of prime concern for system modeling:

1. The system modeling is undertaken with some simplifying assumptions about the system. Though these assumptions limit the quality of system model, it reduces the system complexity and makes understanding easier. Still, the model considers all important, critical and material factors. These assumptions are made regarding all aspects like processes, behaviors, values of inputs etc.
2. The minute details of various components are ignored and a simplified model is developed. For example, there may be various types of data present in the system. The type of data having minute differences is clubbed into single category, thus reducing overall number of data types.

3. The constraints of the system are identified. Some of them may be critical. They are considered in modeling whereas others may be ignored. The constraints may be because of external factors, like processing speed, storage capacity, network features or operating system used.

4. The customers mentioned preferences about technology, tools, interfaces, design, architecture etc. are taken care of.

Example: The 0th and 1st levels of DFD of Production Management System are shown in Figure 3.6 (a) and (b)

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

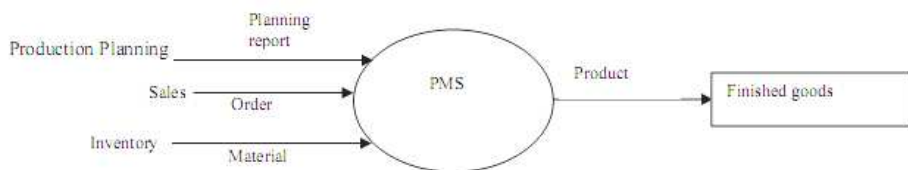


FIGURE 3.6 (a) : Level 0 DFD of PMS

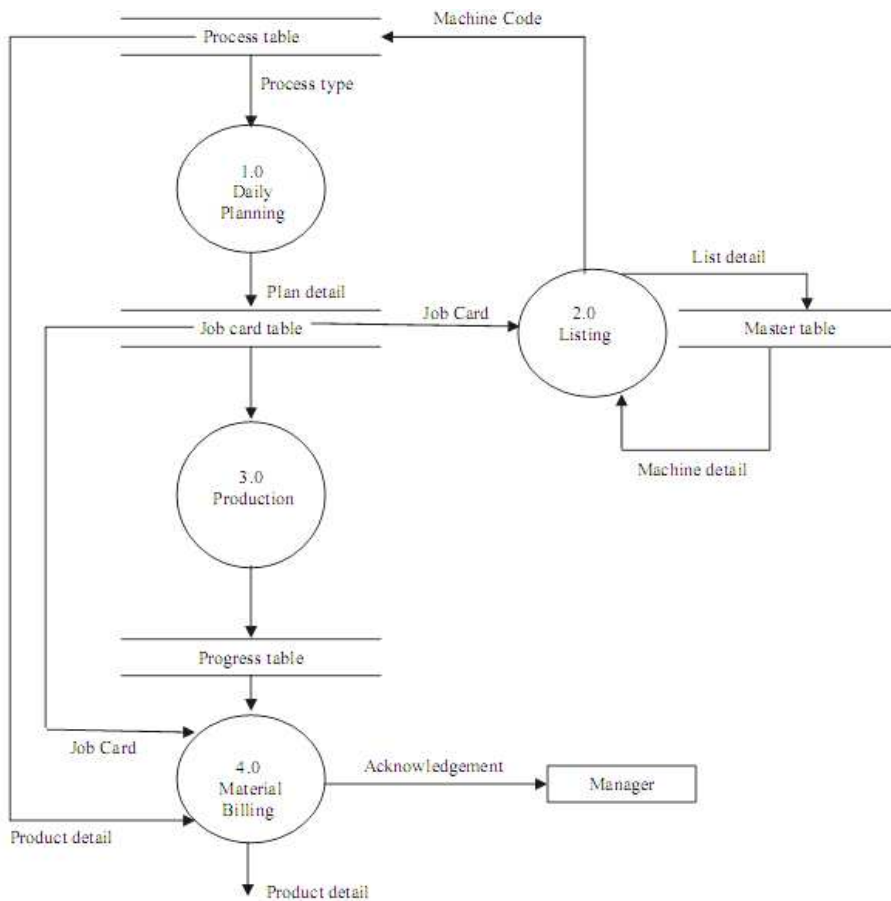


Figure 3.6 (b): Level 1 DFD of PMS

3.5.4 Data Dictionary

This is another tool of requirement analysis which reduces complexity of DFD. A data dictionary is a catalog of all elements of a system. DFD depicts flow of data whereas data dictionary gives details of that information like attribute, type of attribute, size, names of related data items, range of values, data structure definitions etc. The name specifies the name of attribute whose value is collected. For example, fee deposit may be named as FD and course opted may be named as CO.

Related data items captures details of related attributes. Range of values store total possible values of that data. Data structure definition captures the physical structure of data items.

Some of the symbols used in data dictionary are given below:

X = [a/b] x consists of either data element a or b
 X = a x consists of an optional data element a
 X = a+b x consists of data element a & b.
 X = y{a}z x consists of some occurrences of data elements a

which are between y and z.

| Separator
 ** Comments
 @ Identifier
 () Options

Example: The data dictionary of payroll may include the following fields:

PAYROLL = [Payroll Detail
 = @ employee name + employee + id number + employee
 address + Basic Salary + additional allowances
 Employee name = * name of employee *
 Employee id number = * Unique identification no. given to every employee*
 Basic Salary = * Basic pay of employee *
 Additional allowances = * the other perks on Basic pay *
 Employee name = First name + Last name
 Employee address = Address 1 + Address 2 + Address 3

3.5.5 E-R Diagram

Entity-relationship (E-R) diagram is detailed logical representation of data for an organisation. It is data oriented model of a system whereas DFD is a process oriented model. The ER diagram represents data at rest while DFD tracks the motion of data. ERD does not provide any information regarding functionality of data. It has three main components data entities, their relationships and their associated attributes.

Entity: It is most elementary thing of an organisation about which data is to be maintained. Every entity has unique identity. It is represented by rectangular box with the name of entity written inside (generally in capital letters).

Relationship: Entities are connected to each other by relationships. It indicates how two entities are associated. A diamond notation with name of relationship represents as written inside. Entity types that participate in relationship are called degree of relationship. It can be one to one (or unary), one to many or many to many.

Cardinality & Optionally: The cardinality represents the relationship between two entities. Consider the one too many relationships between two entities class and student. Here, cardinality of a relationship is the number of instances of entity student that can be associated with each instance of entity class. This is shown in Figure 3.7.

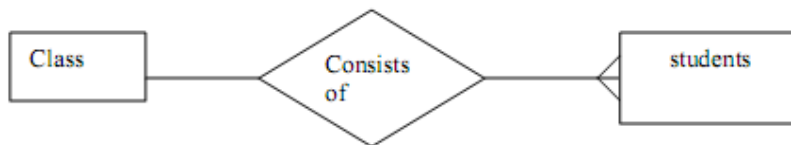


FIGURE 3.7 One to Many cardinality relationship

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

In a situation where there can be no instance of second entity, then, it is called as optional relationship. For example' if a college does not offer a particular course then it will be termed as optional with respect to relationship 'offers'. This relationship is shown in Figure 3.8.



FIGURE 3.8 Minimum cardinality relationship

When minimum cardinality of a relationship is one, then second entity is called as mandatory participant in the relationship. The maximum cardinality is the maximum number of instances of second entity. Thus, the modified ER diagram is shown in Figure 3.9.



FIGURE 3.9 Modified ER diagram representing cardinalities

The relationship cardinalities are shown in Figure 3.10.

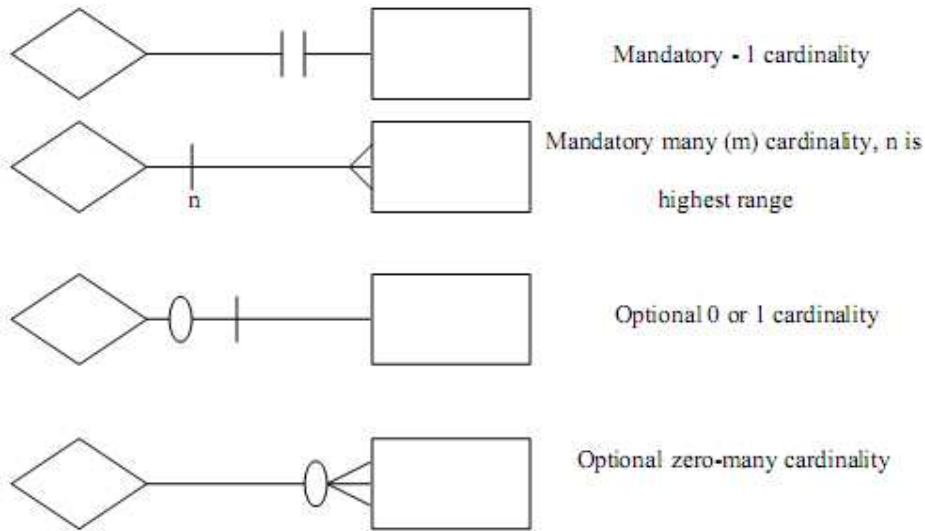


FIGURE 3.10 Relationship cardinalities

Attributes: Attribute is a property or characteristic of an entity that is of interest to the organisation. It is represented by oval shaped box with name of attribute written inside it. For example, the student entity has attributes as shown in Figure 3.11

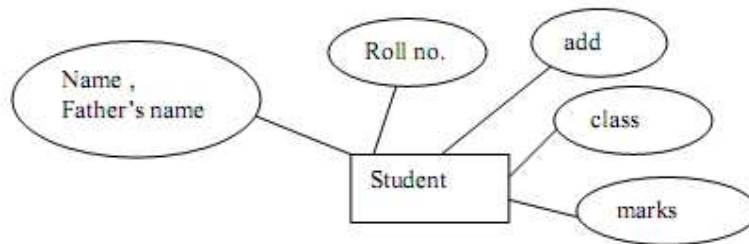


FIGURE 3.13 attributes of entity student

3.5.6 Structured Requirements Definition

Structured Requirements definition is an approach to perform the study about the complete system and its various sub-systems, the external inputs and outputs, functionality of complete system and its various sub-systems. The following are various steps of it:

Step 1: Make a user level data flow diagram. This step is meant for gathering requirements with the interaction of employee. In this process, the requirements engineer interviews each individual of organisation in order to learn what each individual gets as input and produces as output. With this gathered data, create separate data flow diagram for every user.

Step 2: Create combined user level data flow diagram. Create an integrated data flow diagram by merging old data flow diagrams. Remove the inconsistencies if encountered, in this merging process. Finally, an integrated consistent data flow diagram is generated.

Step 3: Generate application level data flow diagram. Perform data analysis at system's level to define external inputs and outputs.

Step 4: Define various functionalities. In this step, the functionalities of various sub-systems and the complete system are defined.

3.6 Software Prototyping and Specification

Prototyping is a process that enables developer to create a small model of software. The IEEE standard defines prototype as a preliminary form or instance of a system that serves as a model for later stages for the final complete version of the system.

A prototype may be categorized as follows:

1. A paper prototype, which is a model depicting human machine interaction in a form that makes the user understand how such interaction, will occur.
2. A working prototype implementing a subset of complete features.
3. An existing program that performs all of the desired functions but additional features are added for improvement.

Prototype is developed so that customers, users and developers can learn more about the problem. Thus, prototype serves as a mechanism for identifying software requirements. It allows the user to explore or criticize the proposed system before developing a full scale system.

3.6.1 Types of Prototype

Broadly, the prototypes are developed using the following two techniques:

Throw away prototype: In this technique, the prototype is discarded once its purpose is fulfilled and the final system is built from scratch. The prototype is built quickly to enable the user to rapidly interact with a working system. As the prototype has to be ultimately discarded, the attention on its speed, implementation aspects, maintainability and fault tolerance is not paid. In requirements defining phase, a less refined set of requirements are hurriedly

defined and throw away prototype is constructed to determine the feasibility of requirement, validate the utility of function, uncover missing requirements, and establish utility of user interface. The duration of prototype building should be as less as possible because its advantage exists only if results from its use are available in timely fashion.

Evolutionary Prototype: In this, the prototype is constructed to learn about the software problems and their solutions in successive steps. The prototype is initially developed to satisfy few requirements. Then, gradually, the requirements are added in the same prototype leading to the better understanding of software system. The prototype once developed is used again and again. This process is repeated till all requirements are embedded in this and the complete system is evolved.

According to SOMM the benefits of developing prototype are listed below:

1. Communication gap between software developer and clients may be identified.
2. Missing user requirements may be unearthed.
3. Ambiguous user requirements may be depicted.
4. A small working system is quickly built to demonstrate the feasibility and usefulness of the application to management.

The sequence of prototyping is shown in following Figure 3.14

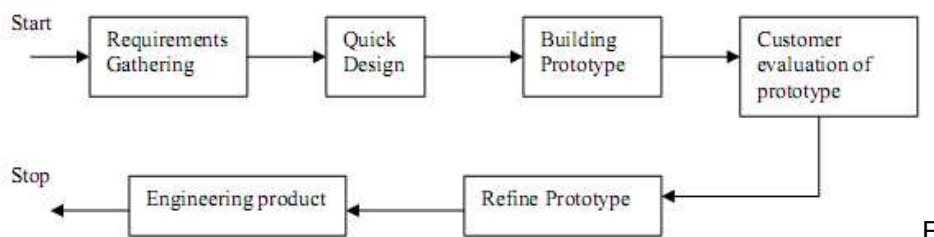


FIGURE 3.14: Sequence of prototyping.

3.6.2 Problems of Prototyping

In some organizations, the prototyping is not as successful as anticipated. A common problem with this approach is that people expect much from insufficient effort. As the requirements are loosely defined, the prototype sometimes gives misleading results about the working of software. Prototyping can have execution inefficiencies and this may be questioned as negative aspect of prototyping. The approach of providing early feedback to user may create the impression on user and user may carry some negative biasing for the completely developed software also.

3.6.3 Advantages of Prototyping

The advantages of prototyping outperform the problems of prototyping. Thus, overall, it is a beneficial approach to develop the prototype. The end user

cannot demand fulfilling of incomplete and ambiguous software needs from the developer.

One additional difficulty in adopting this approach is the large investment that exists in software system maintenance. It requires additional planning about the re-engineering of software. Because, it may be possible that by the time the prototype is build and tested, the technology of software development is changed, hence requiring a complete re-engineering of the product.

3.7 Validation requirements

Measurement is fundamental to any engineering discipline and software engineering is no exception. Software metric is a quantitative measure derived from the attribute of software development life cycle. It behaves as software measures.

A software measure is a mapping from a set of objects in the software engineering world into a set of mathematical constructs such as numbers or vectors of numbers.

Using the software metrics, software engineer measures software processes, and the requirements for that process. The software measures are done according to the following parameters:

- The objective of software and problems associated with current activities,
- The cost of software required for relevant planning relative to future projects,
- Testability and maintainability of various processes and products,
- Quality of software attributes like reliability, portability and maintainability,
- Utility of software product,
- User friendliness of a product.

Various characteristics of software measures identified by Basili are given below:

- Objects of measurement: They indicate the products and processes to be measured.
- Source of measurement: It indicates who will measure the software. For example, software designer, software tester and software managers.
- Property of measurement: It indicates the attribute to be measured like cost of software, reliability, maintainability, size and portability.
- Context of measurement: It indicates the environments in which context the software measurements are applied.

Common software measures

There are significant numbers of software measures. The following are a few common software measures:

Size: It indicates the magnitude of software system. It is most commonly used software measure. It is indicative measure of memory requirement, maintenance effort, and development time.

LOC: It represents the number of lines of code (LOC). It is indicative measure of size oriented software measure. There is some standardization on the methodology of counting of lines. In this, the blank lines and comments are excluded. The multiple statements present in a single line are considered as a single LOC. The lines containing program header and declarations are counted.

Summary

Software engineering practice encompasses concepts, principles, methods, and tools that software engineers apply throughout the software process. Every software engineering project is different, yet a set of generic principles and tasks apply to each process framework activity regardless of the project or the product.

System engineering demands intense communication between the customer and the system engineer. This is achieved through a set of activities that are called requirements engineering elicitation, analysis and negotiation, specification, modeling, validation, and management.

After requirements have been isolated, a system model is produced and representations of each major subsystem can be developed. The system engineering task culminates with the creation of a System Specification a document that forms the foundation for all engineering work that follows.

Due to the complexity involved in software development, the engineering approach is being used in software design. Use of engineering approach in the area of requirements analysis evolved the field of Requirements Engineering. Before the actual system design commences, the system architecture is modeled. Entity-relationship (E-R) diagram is detailed logical representation of data for an organisation. It is data-oriented model of a system whereas DFD is a process oriented model. The ER diagram represents data at rest while DFD tracks the motion of data. ERD does not provide any information regarding functionality of data. It has three main components data entities, their relationships and their associated attributes. Structured Requirements definition is an approach to perform the study about the complete system and its various subsystems, the external inputs and outputs, functionality of complete system and its various subsystems. Prototyping is a process that enables developer to create a small model of software. The IEEE standard defines prototype as a preliminary form or instance of a system that serves as a model for later stages for the final complete version of the system. Measurement is fundamental to any engineering discipline and software engineering is no exception. Software

metric is a quantitative measure derived from the attribute of software development life cycle.

Questions

1. What three “domains” are considered during analysis modeling?
2. Why is feedback important to the software team?
3. What is the advantage of DFD over ER diagram?
4. What are the significance specifying functional requirements in SRS document?
5. What is the purpose of using software metrics?
6. What does “feasibility analysis” imply when it is discussed within the context of the inception function?
7. Why do we say that the analysis model represents a snapshot of a system in time?

UNIT – III

Structure

1. BUILDING ANALYSIS MODEL

- 1.1 Introduction
- 1.2 The Elements of the Analysis Model
- 1.3 Data Modeling
- 1.4 Flow Oriented Modeling
- 1.5 Creating Behavioral model

2. DESIGN ENGINEERING

- 2.1 Introduction
- 2.2 The Process
- 2.3 Software Design and Software Engineering
- 2.4 The Design Process
- 2.5 Design Principles
- 2.6 Design Concepts
- 2.7 Effective Modular Design
- 2.8 Design Heuristics for Effective Modularity
- 2.9 The Design Model
- 2.10 Design Documentation

Objectives

After going through this unit, you should be able to:

- discuss about elements of the analysis model ;
- understand the data modeling;
- understand the flow oriented modeling.
- define process;
- discuss distinguish between software design and software engineering;
- discuss design principles and design concepts;
- discuss design model

1. Building Analysis Model

1.1 Introduction

At a technical level, software engineering begins with a series of modeling tasks that lead to a complete specification of requirements and a comprehensive design representation for the software to be built. The analysis model, actually a set of models, is the first technical representation of a system. Over the years many methods have been proposed for analysis modeling. However, two now dominate. The first, structured analysis, is a classical modeling method and is described in this lesson.

1.2 The Elements of the Analysis Model

The analysis model must achieve three primary objectives:

- (1) to describe what the customer requires,
- (2) to establish a basis for the creation of a software design, and
- (3) to define a set of requirements that can be validated once the software is built.

To accomplish these objectives, the analysis model derived during structured analysis takes the form illustrated in Figure 1.1. At the core of the model lies the data dictionary a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the core. The entity relation diagram (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.

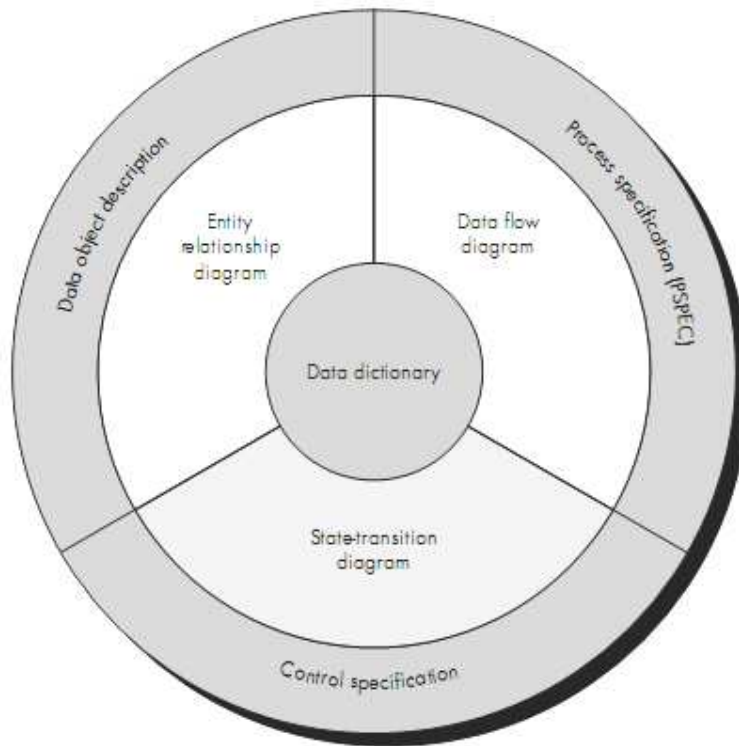


FIGURE 1.1 The structure of the analysis model

The data flow diagram (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and sub-functions) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a process specification (PSPEC). The state transition diagram (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called states) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control aspects of the software is contained in the control specification (CSPEC).

The analysis model encompasses each of the diagrams, specification, descriptions, and the dictionary noted in Figure 1.1. A more detailed discussion of these elements of the analysis model is presented in the sections that follow.

1.3 Data Modeling

Data modeling answers a set of specific questions that are relevant to any data processing application. What are the primary data objects to be processed by the system? What is the composition of each data object and what attributes describe the object? Where do the objects currently reside? What are the relationships between each object and other objects? What are the relationships between the objects and the processes that transform them? To answer these questions, data modeling methods make use of the entity relationship diagram. The ERD, described in detail later in this section, enables a software engineer to identify data objects and their relationships using a graphical notation.

In the context of structured analysis, the ERD defines all data that are entered, stored, transformed, and produced within an application. The entity relationship diagram focuses solely on data (and therefore satisfied the first operational analysis principles), representing a "data network" that exists for a given system. The ERD is especially useful for applications in which data and the relationships that govern data are complex. Unlike the data flow diagram, data modeling considers data independent of the processing that transforms the data.

1.3.1 Data Objects, Attributes, and Relationships

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

Data objects: A data object is a representation of almost any composite information that must be understood by software. By composite information, we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car (Figure 1.2) can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

Data objects (represented in bold) are related to one another. For example, person can own car, where the relationship own connotes a specific

"connection" between person and car. The relationships are always defined by the context of the problem that is being analyzed.

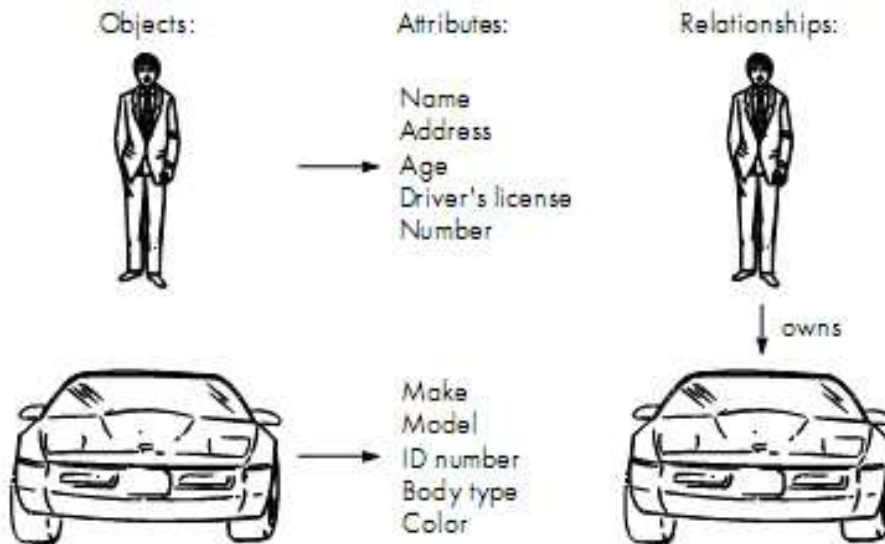


FIGURE 1.2 Data objects, attributes and relationships

A data object encapsulates data only there is no reference within a data object to operations that act on the data.¹ Therefore, the data object can be represented as a table as shown in Figure 1.3. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.

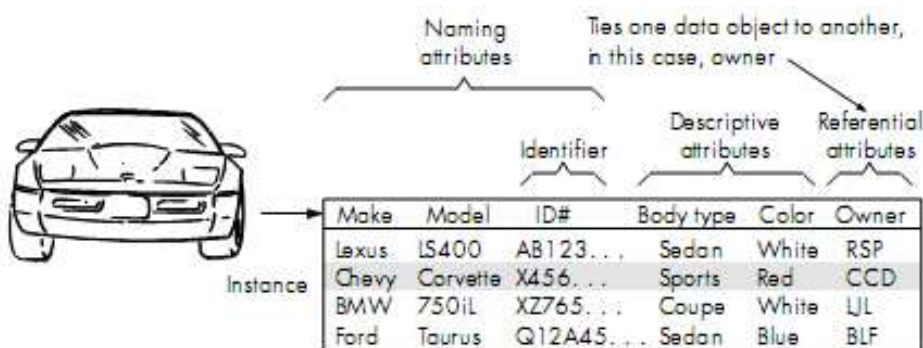


FIGURE 1.3 Tabular representations of data objects

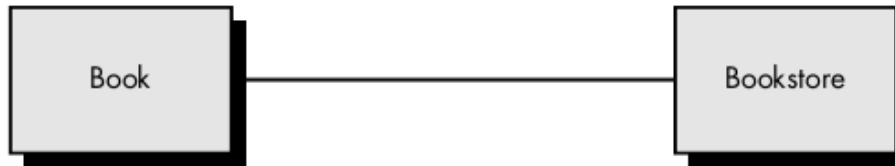
Attributes: Attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference

to another instance in another table. In addition, one or more of the attributes must be defined as an identifier that is; the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number.

The set of attributes that is appropriate for a given data objects is determined through an understanding of the problem context. The attributes for car might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for car might also include ID number, body type and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

Relationships: Data objects are connected to one another in different ways. Consider two data objects, book and bookstore. These objects can be represented using the simple notation illustrated in Figure 1.4a. A connection is established between book and bookstore because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and book stores within the context of the software to be built. We can define a set of object/relationship pairs that defined the relevant relationships. For example,

- A bookstore orders books.
- A bookstore displays books.
- A bookstore stocks books.
- A bookstore sells books.
- A bookstore returns books.



(a) A basic connection between objects



(b) Relationships between objects

FIGURE 1.4 Relationships

The relationships orders, displays, stocks, sells, and returns define the relevant connections between book and bookstore. Figure 1.4b illustrates these object/relationship pairs graphically.

It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

1.3.2 Cardinality and Modality

The elements of data modeling data objects, attributes, and relationships provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood. We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: object X relates to object Y does not provide enough information for software engineering purposes. We must understand how many occurrences of object X are related to how many occurrences of object Y. This leads to a data modeling concept called cardinality.

Cardinality: The data model must be capable of representing the number of occurrences objects in a given relationship. Tillman defines the cardinality of an object/relationship pair in the following manner:

Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply 'one' or 'many.' For example, a husband can have only one wife (in most cultures), while a parent can have many children. Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as

- **One-to-one (1:1)** An occurrence of 'A' can relate to one and only one occurrence of 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
- **One-to-many (1:N)** One occurrence of [object] 'A' can relate to one or many occurrences of 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.' For example, a mother can have many children, but a child can have only one mother.
- **Many-to-many (M:N)** An occurrence of 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.'

For example, an uncle can have many nephews, while a nephew can have many uncles. Cardinality defines "the maximum number of objects that can be participate in a relationship". It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

Modality: The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Figure 1.5 illustrates the relationship, cardinality, and modality between the data objects customer and repair action.

Referring to the figure 1.5, a one to many cardinality relationships is established. That is, a single customer can be provided with zero or many repair actions. The symbols on the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar indicates one and the three-pronged fork indicates many. Modality is indicated by the symbols that are further away from the data object rectangles. The second vertical bar on the left indicates that there must be a customer for a repair action to occur. The circle on the right indicates that there may be no repair action required for the type of problem reported by the customer.

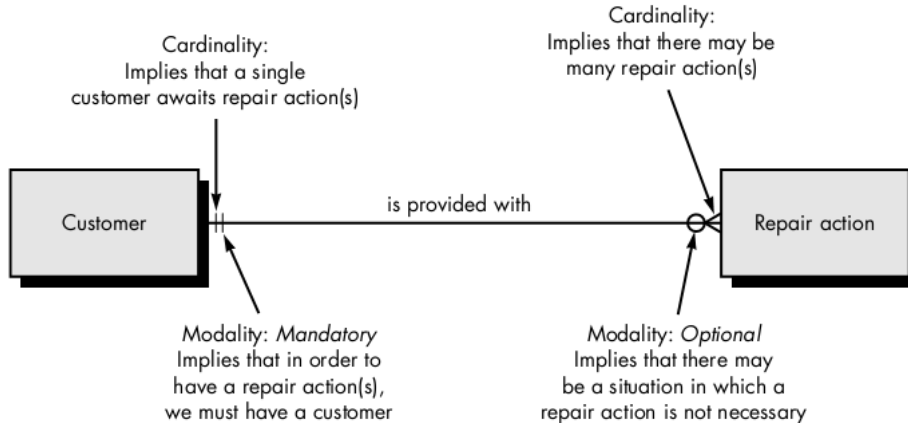


FIGURE 1.5 Cardinality and modality

1.3.3 Entity/Relationship Diagrams

The object/relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the entity/relationship diagram. The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality.

The relationship between the data objects car and manufacturer would be represented as shown in Figure 1.6. One manufacturer builds one or many cars. Given the context implied by the ERD, the specification of the data object car (data object table in Figure 1.6) would be radically different from the earlier specification (Figure 1.3). By examining the symbols at the end of the connection line between objects, it can be seen that the modality of both occurrences is mandatory (the vertical lines).

Expanding the model, we represent a grossly over simplified ERD (Figure 1.7) of the distribution element of the automobile business. New data objects, shipper and dealership, are introduced. In addition, new relationships transports, contracts, licenses, and stocks indicate how the data objects shown in the figure associate with one another. Tables for

each of the data objects contained in the ERD would have to be developed according to the rules introduced. In addition to the basic ERD notation introduced in Figures 1.6 and 1.7, the analyst can represent data object type hierarchies.

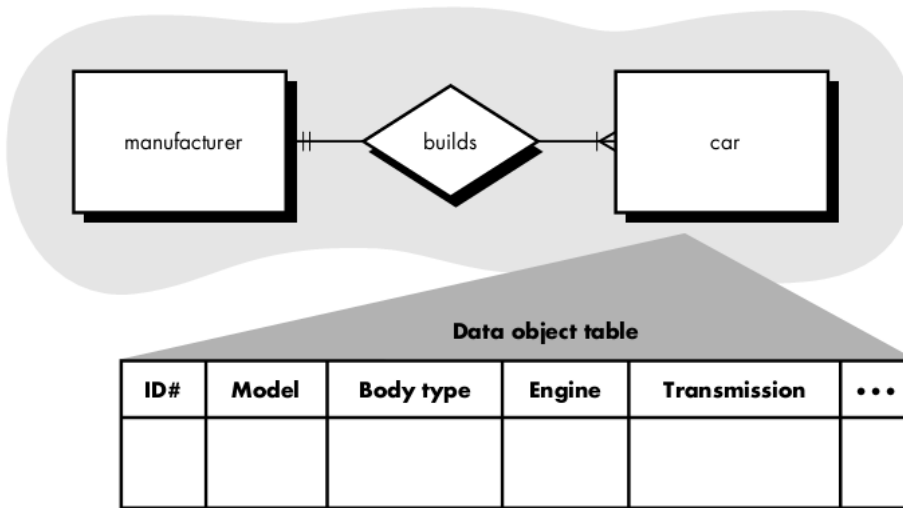


FIGURE 1.6 A simple ERD and data object table

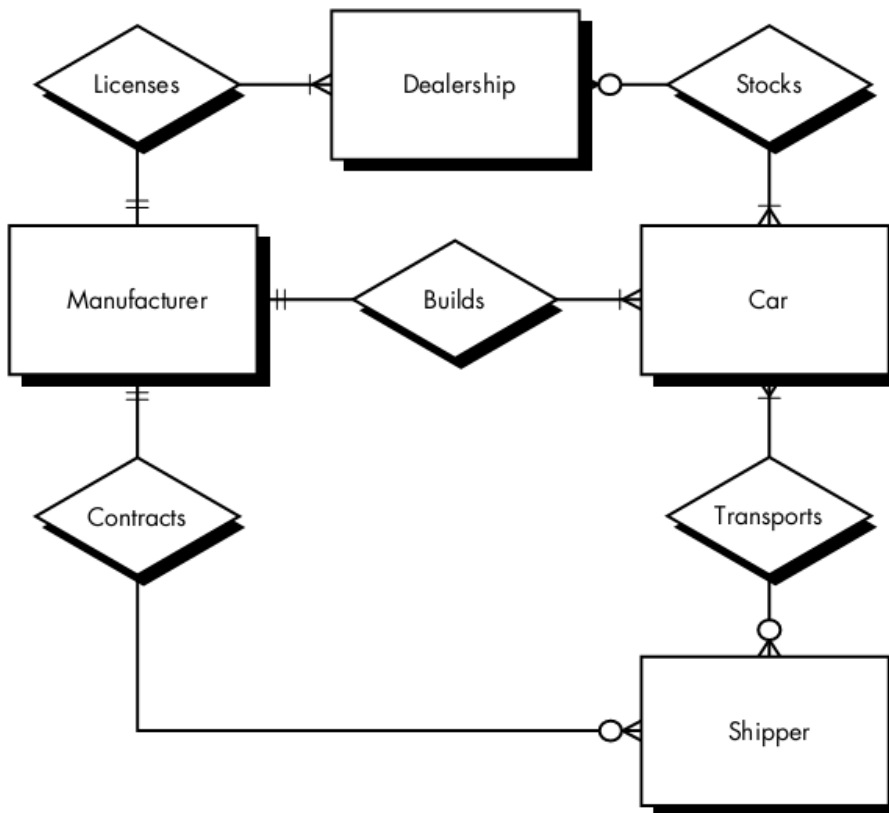


FIGURE 1.7 an expanded ERD

In many instances, a data object may actually represent a class or category of information. For example, the data object car can be categorized as domestic, European, or Asian. The ERD notation shown in Figure 1.8 represents this categorization in the form of a hierarchy.

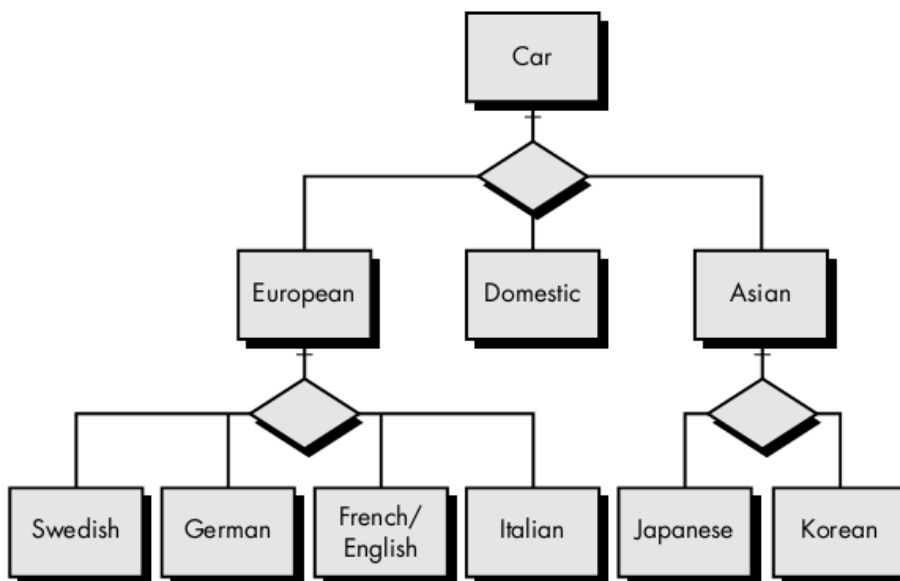


FIGURE 1.8 Data object type hierarchies

ERD notation also provides a mechanism that represents the associatively between objects. An associative data object is represented as shown in Figure 1.9. In the figure, each of the data objects that model the individual subsystems is associated with the data object car.

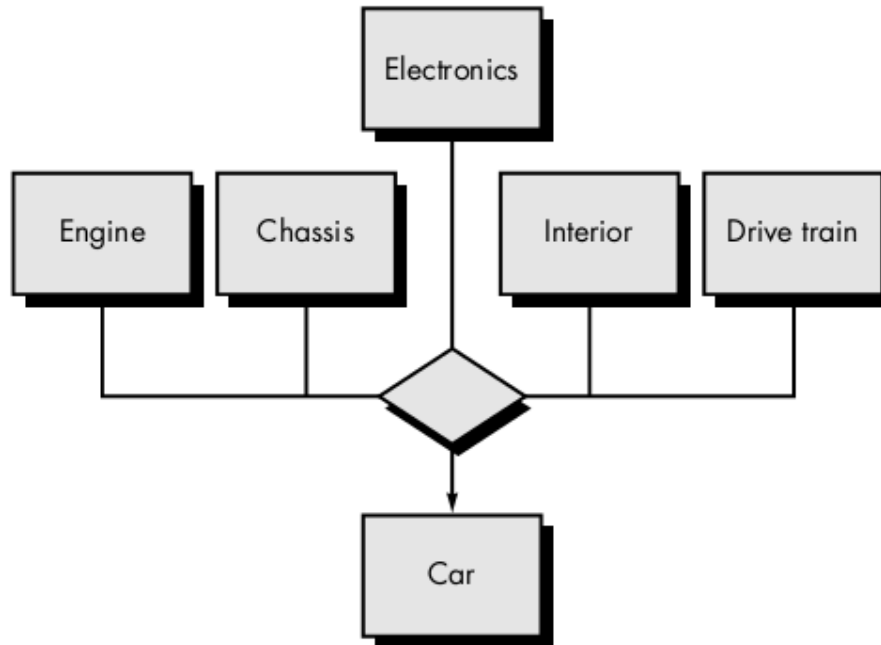


FIGURE 1.9 Associative data objects

Data modeling and the entity relationship diagram provide the analyst with a concise notation for examining data within the context of a software application. In most cases, the data modeling approach is used to create one piece of the analysis model, but it can also be used for database design and to support any other requirements analysis methods.

1.4 Flow Oriented Modeling

Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a flow model for any computer-based system, regardless of size and complexity. Structured analysis began as an information flow modeling technique. A computer-based system is represented as information transform as shown in Figure 1.10. A rectangle is used to represent an external entity; that is, a system element (e.g., hardware, a person, and another program) or another system that produces information for transformation by the software or receives information produced by the

software. A circle (sometimes called a bubble) represents a process or transform that is applied to data (or control) and changes it in some way. An arrow represents one or more data items (data objects). All arrows on a data flow diagram should be labeled.

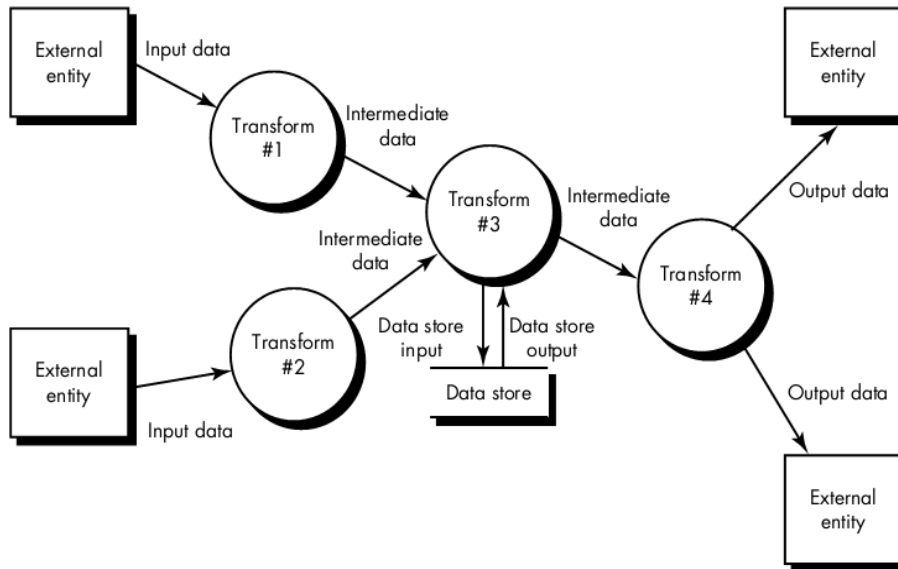


FIGURE 1.10 Information flow model

The double line represents data store stored information that is used by the software. The simplicity of DFD notation is one reason why structured analysis techniques are widely used.

It is important to note that no explicit indication of the sequence of processing or conditional logic is supplied by the diagram. Procedure or sequence may be implicit in the diagram, but explicit logical details are generally delayed until software design. It is important not to confuse a DFD with the flow chart.

1.4.1 Data Flow Diagrams

As information moves through software, it is modified by a series of transformations. A data flow diagram is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a data flow graph or a bubble chart, is illustrated in Figure 1.10.

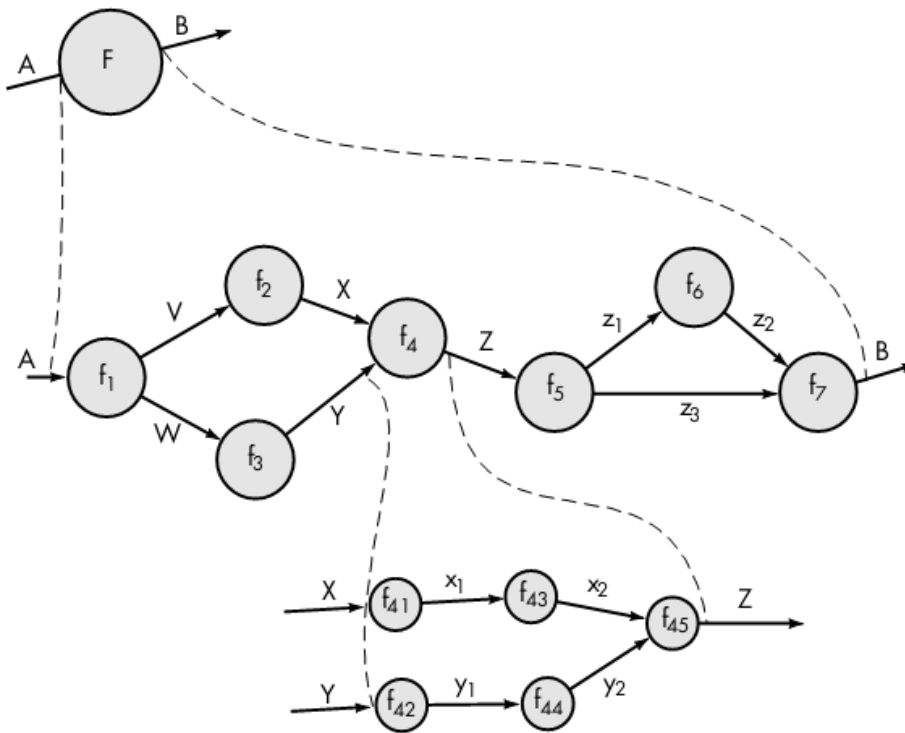


FIGURE 1.11 Information flow refinement

The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Therefore, the DFD provides a mechanism for functional modeling as well as information flow modeling. In so doing, satisfied the second operational analysis principle (i.e., creating a functional model).

A level 0 DFD, also called a fundamental system model or a context model, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a level 1 DFD might contain five or six bubbles with inter connecting arrows. Each of the processes represented at level 1 is a sub function of the overall system depicted in the context model. As we noted earlier, each of the bubbles may be refined or layered to depict more detail. Figure 1.11 illustrates this concept. A fundamental model for system F indicates the primary input is A and ultimate output is B. We refine the F model into transforms f_1 to f_7 . Note that information flow continuity must be maintained; that is, input and output to each refinement must remain the same. This concept, sometimes called balancing, is essential for the development of consistent models. Further refinement of f_4 depicts detail in

the form of transforms f_{41} to f_{45} . Again, the input (X, Y) and output (Z) remain unchanged.

The basic notation used to develop a DFD is not in itself sufficient to describe requirements for software. For example, an arrow shown in a DFD represents a data object that is input to or output from a process. A data store represents some organized collection of data. But what is the content of the data implied by the arrow or depicted by the store? If the arrow (or the store) represents a collection of objects, what are they? These questions are answered by applying another component of the basic notation for structured analysis the data dictionary.

DFD graphical notation must be augmented with descriptive text. A process specification (PSPEC) can be used to specify the processing details implied by a bubble within a DFD. The process specification describes the input to a function, the algorithm that is applied to transform the input, and the output that is produced. In addition, the PSPEC indicates restrictions and limitations imposed on the process (function), performance characteristics that are relevant to the process, and design constraints that may influence the way in which the process will be implemented.

1.4.2 Extensions for Real-Time Systems

Many software applications are time dependent and process as much or more control-oriented information as data. A real-time system must interact with the real world in a time frame dictated by the real world. Aircraft avionics, manufacturing process control, consumer products, and industrial instrumentation are but a few of hundreds of real-time software applications. To accommodate the analysis of real-time software, a number of extensions to the basic notation for structured analysis have been defined. These extensions developed by Ward and Mellor and Hatley and Pirbhai and illustrated in the sections that follow, enable the analyst to represent control flow and control processing as well as data flow and processing.

1.4.3 Ward and Mellor Extensions

Ward and Mellor extend basic structured analysis notation to accommodate the following demands imposed by a real-time system:

- Information flow is gathered or produced on a time-continuous basis.
- Control information is passed throughout the system and associated control processing.
- Multiple instances of the same transformation are sometimes encountered in multitasking situations.
- Systems have states and a mechanism causes transition between states.

In a significant percentage of real-time applications, the system must monitor time continuous information generated by some real-world process. For example, a real time test monitoring system for gas turbine engines might be required to monitor turbine speed, combustor temperature, and a variety of pressure probes on a continuous basis. Conventional data flow notation does not make a distinction between discrete data and time-continuous data. One extension to basic structured analysis notation, shown in Figure 1.12, provides a mechanism for representing time-continuous data flow. The double headed arrow is used to represent time-continuous flow while a single headed arrow is used to indicate discrete data flow. In the figure 1.12, monitored temperature is measured continuously while a single value for temperature set point is also provided. The process shown in the figure produces a time-continuous output, corrected value.

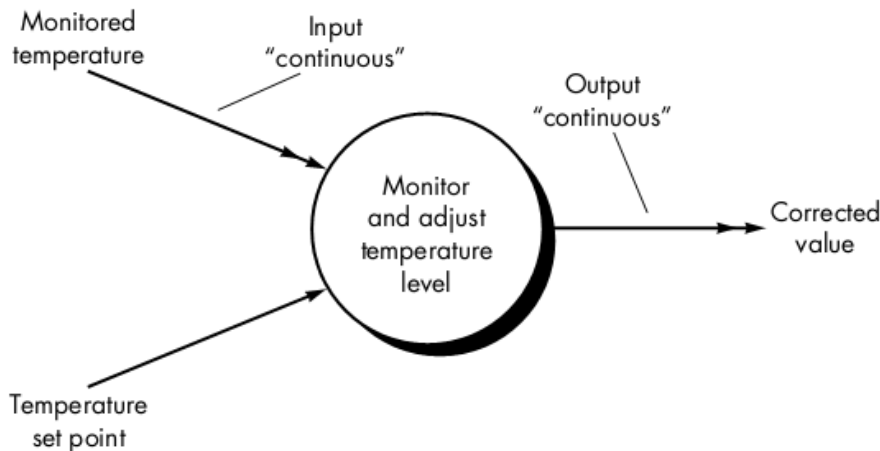


FIGURE 1.12 Time continuous data flow

The distinction between discrete and time continuous data flow has important implications for both the system engineer and the software designer. During the creation of the system model, a system engineer will be better able to isolate those processes that may be performance critical (it is often likely that the input and output of time-continuous data will be performance sensitive). As the physical or implementation model is created, the designer must establish a mechanism for collection of time-continuous data. Obviously, the digital system collects data in a quasi-continuous fashion using techniques such as high-speed polling. The notation indicates where analog-to-digital hardware will be required and which transforms are likely to demand high-performance software.

In conventional data flow diagrams, control or event flows are not represented explicitly. In fact, the software engineer is cautioned to

specifically exclude the representation of control flow from the data flow diagram. This exclusion is overly restrictive when real-time applications are considered, and for this reason, a specialized notation for representing event flows and control processing has been developed. Continuing the convention established for data flow diagrams, data flow is represented using a solid arrow. Control flow, however, is represented using a dashed or shaded arrow. A process that handles only control flows, called a control process, is similarly represented using a dashed bubble.

Control flow can be input directly to a conventional process or into a control process. Figure 1.13 illustrates control flow and processing as it would be represented using Ward and Mellor notation. The figure illustrates a top-level view of a data and control flow for a manufacturing cell. As components to be assembled by a robot are placed on fixtures, a status bit is set within a parts status buffer (a control store) that indicates the presence or absence of each component. Event information contained within the parts status buffer is passed as a bit string to a process, monitor fixture and operator interface. The process will read operator commands only when the control information, bit string, indicates that all fixtures contain components. An event flag, start/stop flag, is sent to robot initiation control, a control process that enables further command processing. Other data flows occur as a consequence of the process activate event that is sent to process robot commands. An event flag, start/stop flag, is sent to robot initiation control, a control process that enables further command processing. Other data flows occur as a consequence of the process activate event that is sent to process robot commands.

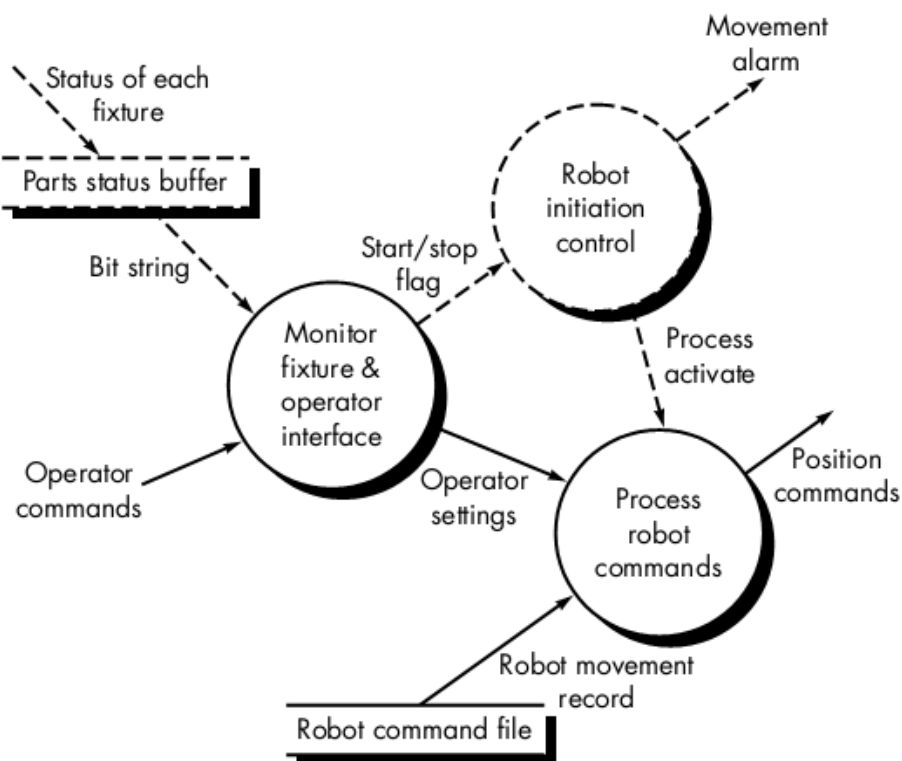


FIGURE 1.13 Data and control flows using Ward and Mellor notation

In some situations multiple instances of the same control or data transformation process may occur in a real-time system. This can occur in a multitasking environment when tasks are spawned as a result of internal processing or external events. For example, a number of part status buffers may be monitored so that different robots can be signaled at the appropriate time. In addition, each robot may have its own robot control system. The Ward and Mellor notation used to represent multiple equivalent instances simply overlays process (or control process) bubbles to indicate multiplicity.

1.4.4 Hatley and Pirbhai Extensions

The Hatley and Pirbhai extensions to basic structured analysis notation focus less on the creation of additional graphical symbols and more on the representation and specification of the control-oriented aspects of the software. The dashed arrow is once again used to represent control or event flow. Unlike Ward and Mellor, Hatley and Pirbhai suggest that dashed and solid notation be represented separately.

Therefore, a control flow diagram is defined. The CFD contains the same processes as the DFD, but shows control flow, rather than data flow. Instead of representing control processes directly within the flow model, a notational reference (a solid bar) to a control specification (CSPEC) is used. In essence, the solid bar can be viewed as a "window" into an "executive" (the CSPEC) that controls the processes (functions) represented in the DFD based on the event that is passed through the window. The CSPEC is used to indicate (1) how the software behaves when an event or control signal is sensed and (2) which processes are invoked as a consequence of the occurrence of the event. A process specification is used to describe the inner workings of a process represented in a flow diagram. Using the notation described in Figures 1.12 and 1.13, along with additional information contained in PSPECs and CSPECs, Hatley and Pirbhai create a model of a real-time system. Data flow diagrams are used to represent data and the processes that manipulate it. Control flow diagrams show how events flow among processes and illustrate those external events that cause various processes to be activated. The interrelationship between the process and control models is shown schematically in Figure 1.14. The process model is "connected" to the control model through data conditions. The control model is "connected" to the process model through process activation information contained in the CSPEC.

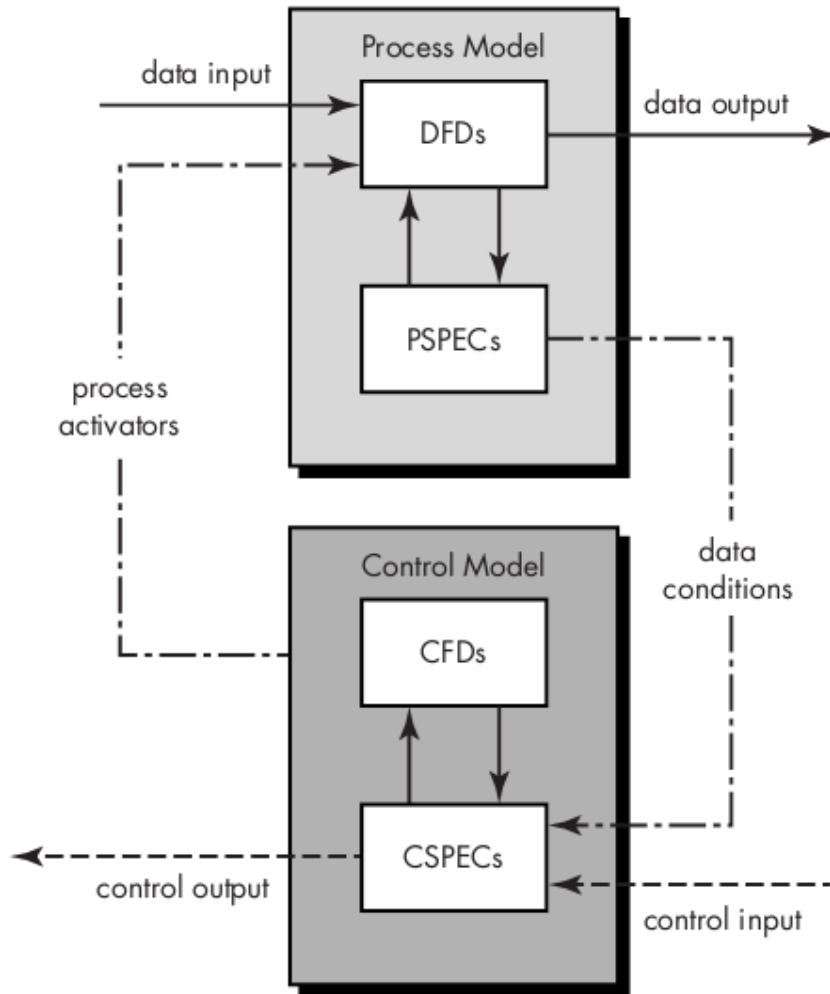
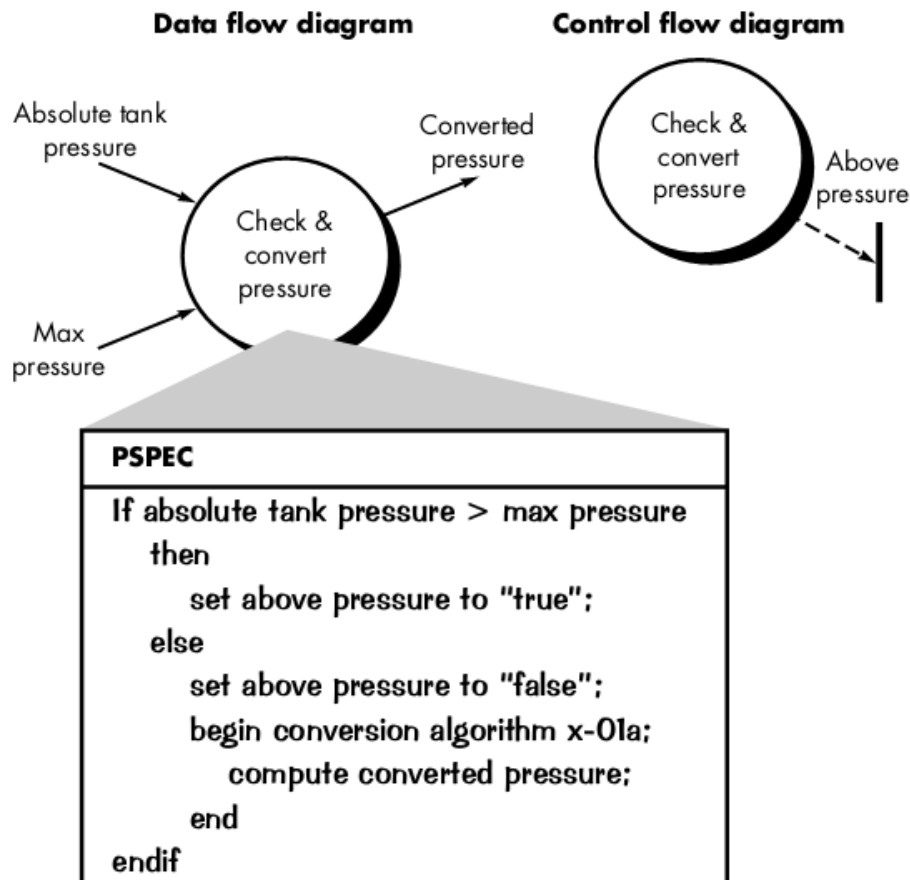


FIGURE 1.14 The relationship between data and control models

A data condition occurs whenever data input to a process result in control output. This situation is illustrated in Figure 1.15, part of a flow model for an automated monitoring and control system for pressure vessels in an oil refinery. The process check and convert pressure implements the algorithm described in the PSPEC pseudo code shown. When the absolute tank pressure is greater than an allowable maximum, an above pressure event is generated. Note that when Hatley and Pirbhai notation is used, the data flow is shown as part of a DFD, while the control flow is noted separately as part of a control flow diagram. As we noted earlier, the vertical solid bar into which the above pressure event flows is a pointer to the CSPEC.



Therefore, to determine what happens when this event occurs, we must check the CSPEC. The control specification (CSPEC) contains a number of important modeling tools. A process activation table is used to indicate which processes are activated by a given event. For example, a process activation table (PAT) for Figure 1.15 might indicate that the above pressure event would cause a process reduce tank pressure (not shown) to be invoked. In addition to the PAT, the CSPEC may contain a state transition diagram.

1.5 Creating Behavioral model

Behavioral modeling is an operational principle for all requirements analysis methods. Yet, only extended versions of structured analysis provide a notation for this type of modeling. The state transition diagram represents the behavior of a system by depicting its states and the events that cause the system to change state. In addition, the STD indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

A state is any observable mode of behavior. For example, states for a monitoring and control system for pressure vessels might be monitoring state, alarm state, pressure release state, and so on. Each of these states represents a mode of behavior of the system. A state transition diagram indicates how the system moves from state to state.

To illustrate the use of the Hatley and Pirbhai control and behavioral extensions, consider software embedded within an office photocopier machine. A simplified representation of the control flow for the photocopier software is shown in Figure 1.16. Data flow arrows have been lightly shaded for illustrative purposes, but in reality they are not shown as part of a control flow diagram.

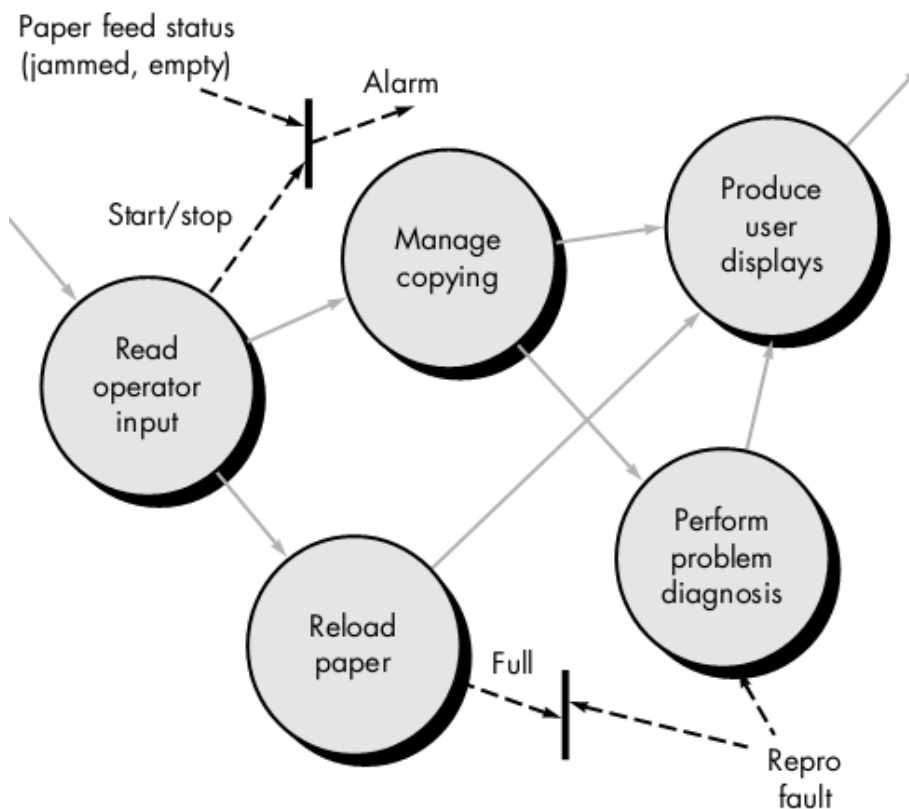


FIGURE 1.16 Level 1 CFD for photocopier software

Control flows are shown entering and exiting individual processes and the vertical bar representing the CSPEC "window." For example, the paper feed status and start/stop events flow into the CSPEC bar. This implies that each of these events will cause some process represented in the CFD to be activated. If we were to examine the CSPEC internals, the start/stop event would be shown to activate/deactivate the manage copying process. Similarly, the jammed event (part of paper feed status) would activate

perform problem diagnosis. It should be noted that all vertical bars within the CFD refer to the same CSPEC. An event flow can be input directly into a process as shown with repro fault. However, this flow does not activate the process but rather provides control information for the process algorithm.

A simplified state transition diagram for the photocopier software is shown in Figure 1.17. The rectangles represent system states and the arrows represent transitions between states. Each arrow is labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. Therefore, when the paper tray is full and the start button is pressed, the system moves from the reading commands state to the making copies state. Note that states do not necessarily correspond to processes on a one-to-one basis. For example, the state making copies would encompass both the manage copying and produce user displays processes shown in Figure 1.16.

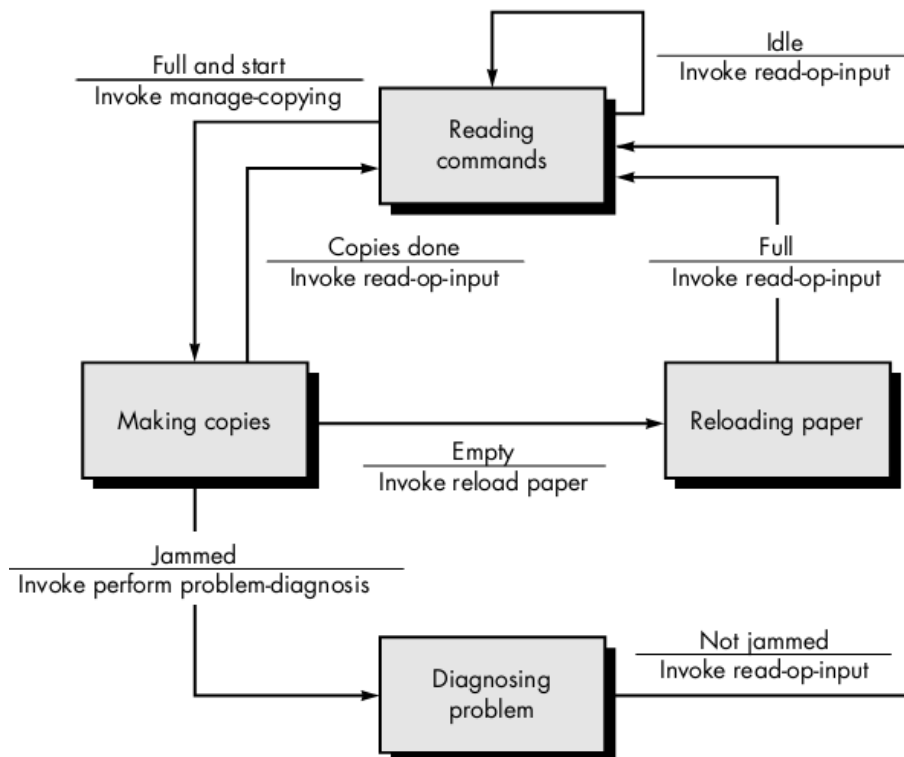


FIGURE 1.17 State transition diagrams for photocopier software

2. Design Engineering

2.1 Introduction

The designer's goal is to produce a model or representation of an entity that will later be built. Diversification and convergence combine intuition and judgment based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation. Software design, like engineering design approaches in other disciplines, changes continually as new methods, better analysis, and broader understanding evolve. Software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines. However, methods for software design do exist, criteria for design quality are available, and design notation can be applied. In this chapter, we explore the fundamental concepts and principles that are applicable to all software design.

2.2 The Process

By the time the systems designer comes to the design phase of the system life cycle, he or she has a pretty clear understanding of what the new system should do and why. This information is recorded in several documents:

- The feasibility study discusses the pros, cons, and costs of building the system.
- The project plan provides preliminary information about the project, its mission and goals, its schedule, and its cost estimate.
- The system requirements specification (SRS) contains detailed information about the requirements of the system.

In spite of this detailed documentation, there may still be some uncertainty regarding future capabilities of the new system due to the different and changing perspectives of the end users and other stakeholders. Different people will see different possibilities for the new system, which is why a push to propose alternative solutions may take place. The designer must then consider the different views, covering all structured requirements, and transform them into several competing design strategies. Only one design will eventually be pursued.

2.3 Software Design and Software Engineering

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities design, code generation, and test that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software. Each of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 2.1. Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods (discussed in later chapters), the design task produces a data design, an architectural design, an interface design, and a component design.

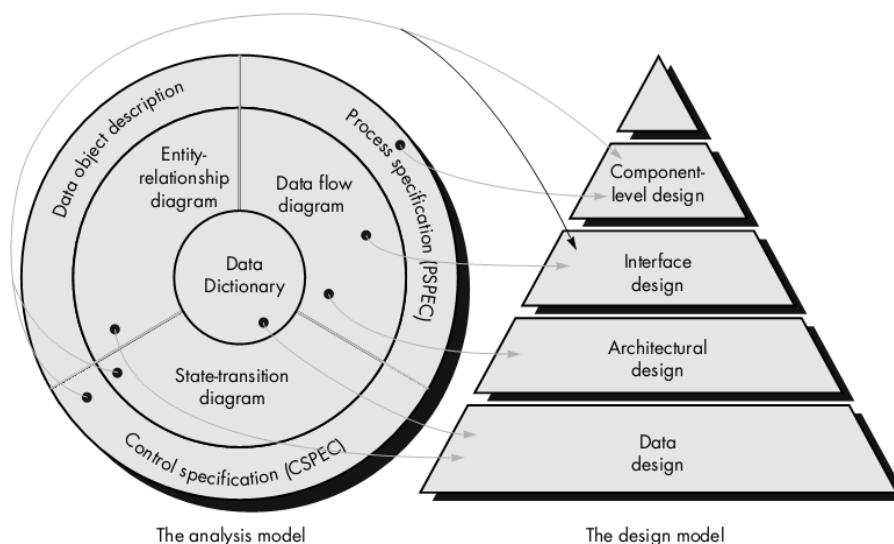


FIGURE 2.1 Translating the analysis model into a software design

The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.

The architectural design defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve

the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied. The architectural design representation the framework of a computer-based system can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model. The interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design. During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word quality. Design is the place where quality is fostered in software engineering. 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 the software engineering and software support steps that follow. Without design, we risk building an unstable system one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

2.4 The Design Process

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

2.4.1 Design and Software Quality

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

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved? In order to evaluate the quality of a design representation, we must establish technical criteria for good design. Later in this chapter, we discuss design quality criteria in some detail. For the time being, we present the following guidelines:

A design should exhibit an architectural structure that (1) has been created using recognizable design patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and sub-functions.
3. A design should contain distinct representations of data, architecture, interfaces, and components (modules).
4. A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

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

2.4.2 The Evolution of Software Design

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

Many design methods, growing out of the work just noted, are being applied throughout the industry. Each software design method introduces unique heuristics and notation, as well as a some- what parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of analysis model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

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

2.5 Design Principles

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

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

Basic design principles enable the software engineer to navigate the design process. Davis suggests a set¹ of principles for software design, which have been adapted and extended in the following list:

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

- **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts (Section 13.4) and design measures are available to assist the designer in assessing quality.
- **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

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

2.6 Design Concepts

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

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

2.6.1 Abstraction

When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. Wasserman provides a useful definition:

Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A procedural abstraction is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A data abstraction is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Many modern programming languages provide mechanisms for creating abstract data types. For example, the Ada package is a programming language mechanism that provides support for both data and procedural abstraction. The original abstract data type is used as a template or generic data structure from which other data structures can be instantiated.

Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the synchronization semaphore [KAI83] used to coordinate activities in an operating system.

2.6.2 Refinement

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

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements

analysis. The difference is in the level of implementation detail that is considered, not the approach. Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

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

2.6.3 Modularity

The concept of modularity in computer software has been espoused for almost five decades. Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called modules that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and over all complexity would make understanding close to impossible. To illustrate this point, consider the following argument based on observations of human problem solving.

Let $C(x)$ be a function that defines the perceived complexity of a problem x , and $E(x)$ be a function that defines the effort (in time) required to solve a problem x . For two problems, p_1 and p_2 , if

$$C(p_1) > C(p_2) \quad \text{-----}(1-1a)$$

it follows that

$$E(p_1) > E(p_2) \quad (1-1b)$$

As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

$$C(p_1 + p_2) > C(p_1) + C(p_2) \quad \text{-----} (1-2)$$

Expression (13-2) implies that the perceived complexity of a problem that combines p_1 and p_2 is greater than the perceived complexity when each problem is considered separately. Considering Expression (1-2) and the condition implied by Expressions (1-1), it follows that

$$E(p_1 + p_2) > E(p_1) + E(p_2) \text{ ----- (1-3)}$$

This leads to a "divide and conquer" conclusion it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in Expression (1-3) has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

It is possible to conclude from Expression (1-3) that, if we subdivide software indefinitely, the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 2.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

The curves shown in Figure 2.2 do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of M . Under modularity or over modularity should be avoided.

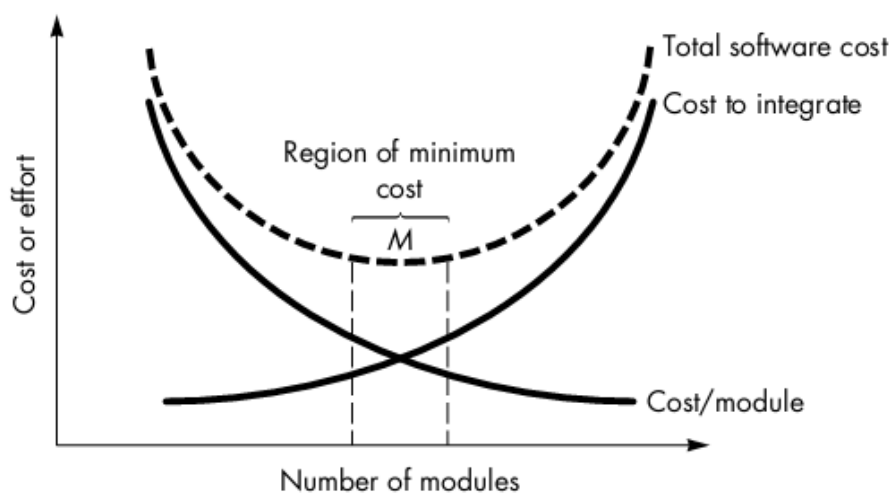


FIGURE 2.2 Modularity and software cost

Another important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system. Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

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

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

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

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

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

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

2.6.4 Software Architecture

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

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

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

Structural properties: This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

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

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

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [GAR95]. Structural models represent architecture as an organized collection of program components. Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. Process models focus on the design of the business or technical process that the system must accommodate. Finally, functional models can be used to represent the functional hierarchy of a system.

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

2.6.5 Control Hierarchy

Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as

sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation. The most common is the treelike diagram (Figure 2.3) that represents hierarchical control for call and return architectures. However, other notations, such as Warnier-Orr and Jackson diagrams may also be used with equal effectiveness. In order to facilitate later discussions of structure, we define a few simple measures and terms. Referring to Figure 2.3, depth and width provide an indication of the number of levels of control and overall span of control, respectively. Fan-out is a measure of the number of modules that are directly controlled by another module. Fan-in indicates how many modules directly control a given module.

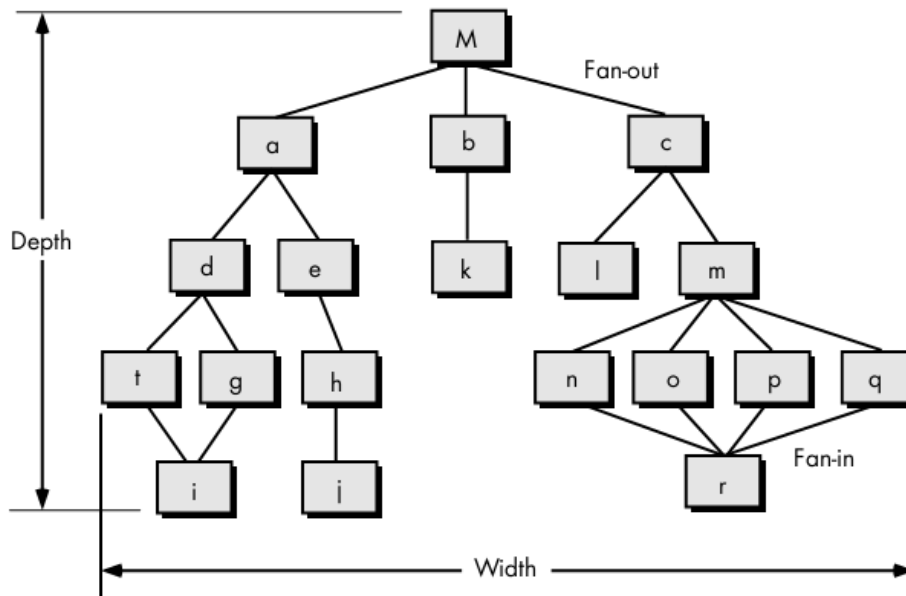


FIGURE 2.3 Structure terminologies for a call and return architectural style

The control relationship among modules is expressed in the following way: A module that controls another module is said to be superordinate to it, and conversely, a module controlled by another is said to be subordinate to the controller. For example, referring to Figure 2.3, module M is superordinate to modules a, b, and c. Module h is subordinate to module e and is ultimately subordinate to module M. Width-oriented relationships (e.g., between modules d and e) although possible to express in practice, need not be defined with explicit terminology.

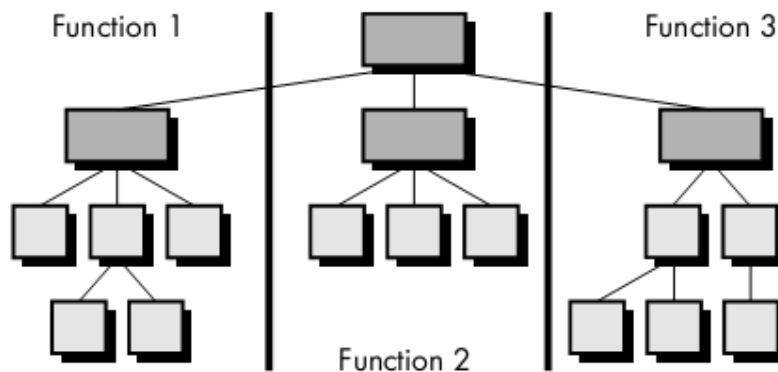
The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity. Visibility indicates the

set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module. Connectivity indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.

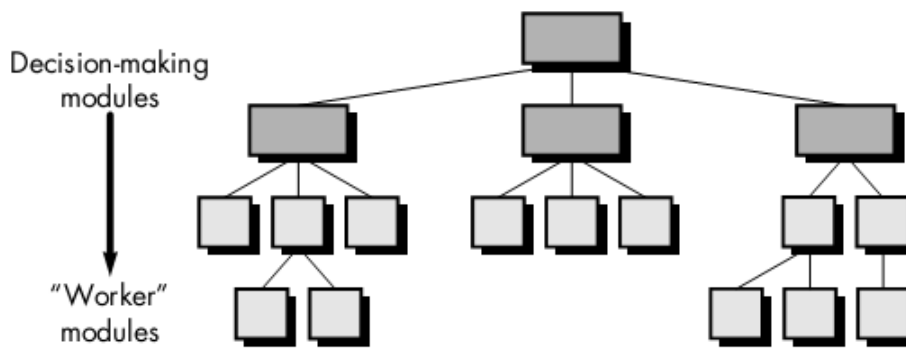
2.6.6 Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure 2.4a, horizontal partitioning defines separate branches of the modular hierarchy for each major program function. Control modules, represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions input, data transformation (often called processing) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

- software that is easier to test
- software that is easier to maintain
- propagation of fewer side effects
- software that is easier to extend



(a) Horizontal partitioning



(b) Vertical partitioning

FIGURE 2.4 Structural partitioning

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

Vertical partitioning (Figure 2.4b), often called factoring, suggests that control (decision making) and work should be distributed top-down in the program structure. Top-level modules should perform control functions and do little actual processing work.

Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks. The nature of change in program structures justifies the need for vertical partitioning. Referring to Figure 2.4b, it can be seen that a change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it. A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects. In general, changes to computer programs revolve around changes to input, computation or transformation, and output. The overall control structure of the program (i.e., its basic behavior is far less likely to change). For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable a key quality factor.

2.6.7 Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. Entire texts have been dedicated to these topics, and a complete discussion is beyond the scope of this book. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.

The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures. A scalar item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long.

When scalar items are organized as a list or contiguous group, a sequential vector is formed. Vectors are the most common of all data structures and open the door to variable indexing of information. When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an n -dimensional space is created. The most common n -dimensional space is the two-dimensional matrix. In many programming languages, an n dimensional space is called an array.

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

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

It is important to note that data structures, like program structure, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector

or a linked list. Depending on the level of design detail, the internal workings of a stack may or may not be specified.

2.6.8 Software Procedure

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

There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered as illustrated in Figure 2.5.

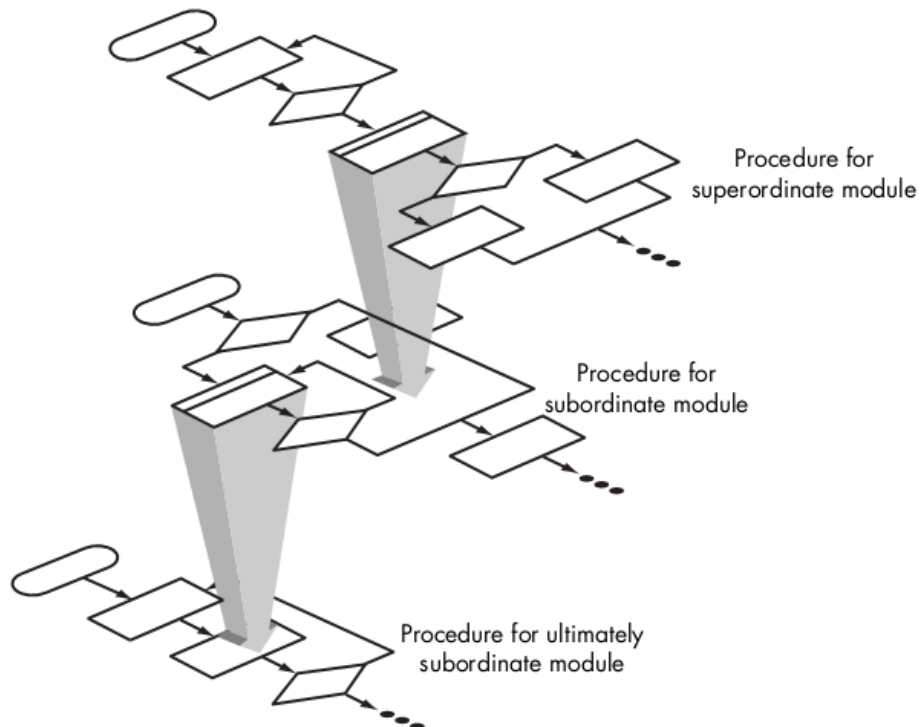


FIGURE 2.5 Procedure is layered

2.6.9 Software Procedure

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

decision points, repetitive operations, and even data organization and structure. There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered as illustrated in Figure 2.5

2.6.10 Information Hiding

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

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

2.7 Effective Modular Design

All the fundamental design concepts described in the preceding section serve to precipitate modular designs. In fact, modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity, facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

1.7.1 Functional Independence

The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding. In landmark papers on software design Parnas and Wirth allude to refinement techniques that

enhance module independence. Later work by Stevens, Myers, and Constantine solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is measured using two qualitative criteria: cohesion and coupling. Cohesion is a measure of the relative functional strength of a module. Coupling is a measure of the relative interdependence among modules.

1.7.2 Cohesion

A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed.

At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed coincidentally cohesive. A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is logically cohesive. When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits temporal cohesion.

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed prespecified bounds. It performs the following tasks: (1) computes supplementary data based on original computed data, (2) produces an error report

(with graphical content) on the user's workstation, (3) performs follow-up calculations requested by the user, (4) updates a database, and (5) enables menu selection for subsequent processing. Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can serve only to increase the likelihood of error propagation when a modification is made to one of its processing tasks.

Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, procedural cohesion exists. When all processing elements concentrate on one area of a data structure, communicational cohesion is present. High cohesion is characterized by a module that performs one distinct procedural task.

As we have already noted, it is unnecessary to determine the precise level of cohesion. Rather it is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

1.7.3 Coupling

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates through a system.

Figure 2.6 provides examples of different types of module coupling. Modules a and d are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module c is subordinate to module and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called data coupling) is exhibited in this portion of structure. A variation of data coupling, called stamp coupling is found when a portion of a data structure

(rather than simple arguments) is passed via a module interface. This occurs between modules b and a.

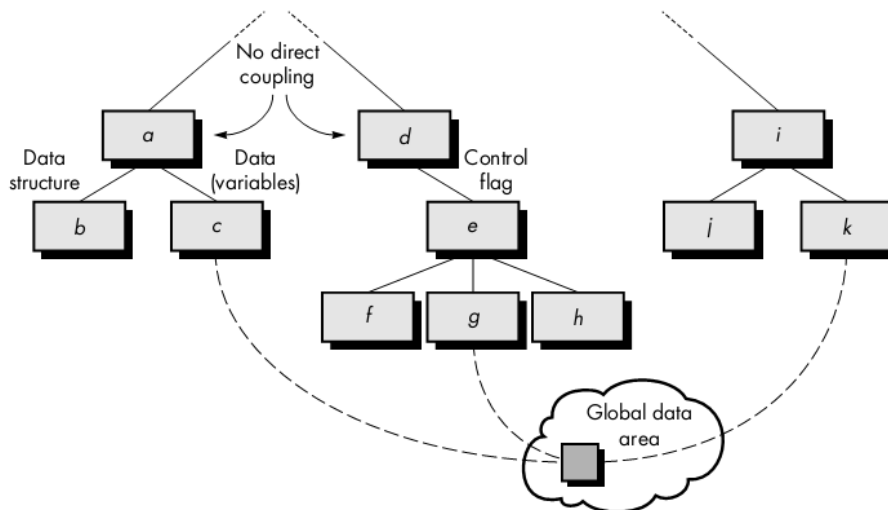


FIGURE 2.6 Types of coupling

At moderate levels, coupling is characterized by passage of control between modules. Control coupling is very common in most software designs and is shown in Figure 2.6 where a “control flag” (a variable that controls decisions in a subordinate or super ordinate module) is passed between modules d and e.

Relatively high levels of coupling occur when modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols. External coupling is essential, but should be limited to a small number of modules with a structure. High coupling also occurs when a number of modules reference a global data area. Common coupling, as this mode is called, is shown in Figure 2.6. Modules c, g, and k each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module c initializes the item. Later module g recomputed and updates the item. Let's assume that an error occurs and g updates the item incorrectly. Much later in processing module, k reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module k; the actual cause, module g. Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them. The highest degree of coupling, content coupling, occurs when one module makes use of data or control information maintained within the boundary of another

module. Secondly, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

The coupling modes just discussed occur because of design decisions made when structure was developed. Variants of external coupling, however, may be introduced during coding. For example, compiler coupling ties source code to specific (and often non-standard) attributes of a compiler; operating system (OS) coupling ties design and resultant code to operating system "hooks" that can create havoc when OS changes occur.

2.8 Design Heuristics for Effective Modularity

Once program structure has been developed, effective modularity can be achieved by applying the design concepts introduced earlier in this chapter. The program structure can be manipulated according to the following set of heuristics:

1. Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion. Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence. An exploded module becomes two or more modules in the final program structure. An imploded module is the result of combining the processing implied by two or more modules.

An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

2. Attempt to minimize structures with high fan-out; strive for fan-in as depth increases. The structure shown inside the cloud in Figure 2.7 does not make effective use of factoring. All modules are "pancaked" below a single control module. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

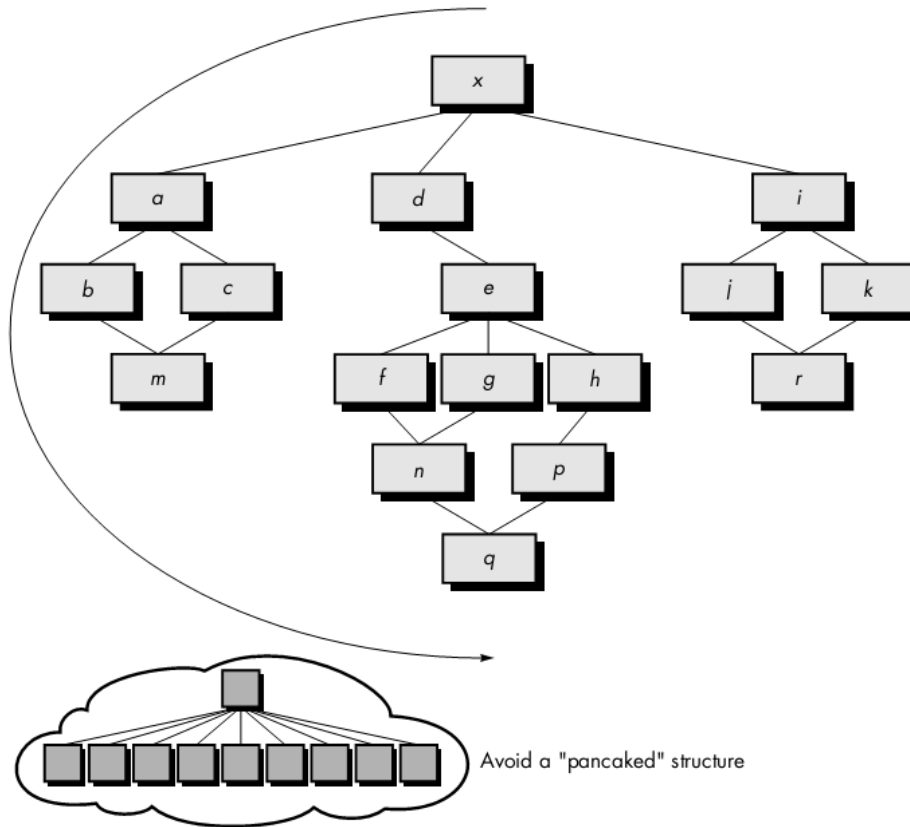


FIGURE 2.7 Program structures

3. Keep the scope of effect of a module within the scope of control of that module. The scope of effect of module *e* is defined as all other modules that are affected by a decision made in module *e*. The scope of control of module *e* is all modules that are subordinate and ultimately subordinate to module *e*.

Referring to Figure 2.7, if module *e* makes a decision that affects modular, we have a violation of this heuristic, because module *r* lies outside the scope of control of module *e*.

4. Evaluate module interfaces to reduce complexity and redundancy and improve consistency. Module interface complexity is a prime cause of software errors. Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be reevaluated.

5. Define modules whose function is predictable, but avoid modules that are overly restrictive. A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use. A module that restricts processing to a single sub function exhibits high cohesion and is viewed with favor by a designer. However, a module that arbitrarily restricts the size of a local data structure, options within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

6. Strive for "controlled entry" modules by avoiding "pathological connections." This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. Pathological connection refers to branches or references into the middle of a module.

2.9 The Design Model

The design principles and concepts discussed in this chapter establish a foundation for the creation of the design model that encompasses representations of data, architecture, interfaces, and components. Like the analysis model before it, each of these design representations is tied to the others, and all can be traced back to software requirements.

In Figure 2.1, the design model was represented as a pyramid. The symbolism of this shape is important. A pyramid is an extremely stable object with a wide base and a low center of gravity. Like the pyramid, we want to create a software design that is stable. By establishing a broad foundation using data design, a stable mid-region with architectural and interface design, and a sharp point by applying component-level design, we create a design model that is not easily "tipped over" by the winds of change.

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

2.10 Design Documentation

The Design Specification addresses different aspects of the design model and is completed as the designer refines his representation of the software. First, the overall scope of the design effort is described. Much of the

information presented here is derived from the System Specification and the analysis model (Software Requirements Specification).

Next, the data design is specified. Database structure, any external file structures, internal data structures, and a cross reference that connects data objects to specific files are all defined.

The architectural design indicates how the program architecture has been derived from the analysis model. In addition, structure charts are used to represent the module hierarchy (if applicable).

The design of external and internal program interfaces is represented and a detailed design of the human/machine interface is described. In some cases, a detailed prototype of a GUI may be represented.

Components separately addressable elements of software such as subroutines, functions, or procedures are initially described with an English-language processing narrative. The processing narrative explains the procedural function of a component (module). Later, a procedural design tool is used to translate the narrative into a structured description.

The Design Specification contains a requirements cross reference. The purpose of this cross reference (usually represented as a simple matrix) is (1) to establish that all requirements are satisfied by the software design and (2) to indicate which components are critical to the implementation of specific requirements. The first stage in the development of test documentation is also contained in the design document. Once program structure and interfaces have been established, we can develop guidelines for testing of individual modules and integration of the entire package. In some cases, a detailed specification of test procedures occurs in parallel with design. In such cases, this section may be deleted from the Design Specification.

Design constraints, such as physical memory limitations or the necessity for a specialized external interface, may dictate special requirements for assembling or packaging of software. Special considerations caused by the necessity for program overlay, virtual memory management, high-speed processing, or other factors may cause modification in design derived from information flow or structure. In addition, this section describes the approach that will be used to transfer software to a customer site.

The final section of the Design Specification contains supplementary data. Algorithm descriptions, alternative procedures, tabular data, excerpts from other documents, and other relevant information are presented as a special note or as a separate appendix. It may be advisable to develop a

Preliminary Operations/Installation Manual and include it as an appendix to the design document.

Summary

Structured analysis, a widely used method of requirements modeling, relies on data modeling and flow modeling to create the basis for a comprehensive analysis model. Using entity-relationship diagrams, the software engineer creates a representation of all data objects that are important for the system. Data and control flow diagrams are used as a basis for representing the transformation of data and control. At the same time, these models are used to create a functional model of the software and to provide a mechanism for partitioning function. A behavioral model is created using the state transition diagram.

Design is the technical kernel of software engineering. During design, progressive refinement of data structure, architecture, interfaces, and procedural detail of software components are developed, reviewed, and documented. Design results in representations of software that can be assessed for quality. A number of fundamental software design principles and concepts have been proposed over the past four decades. Design principles guide the software engineer as the design process proceeds. Design concepts provide basic criteria for design quality.

Design is a process of translating analysis model to design models that are further refined to produce detailed design models. The process of refinement is the process of elaboration to provides necessary details to the programmer. Data design deals with data structure selection and design. Modularity of program increases maintainability and encourages parallel development. The aim of good modular design is to produce highly cohesive and loosely coupled modules. Independence among modules is central to modularity. Good user interface design helps software to interact effectively to external environment. Tips for good interface design helps designer to achieve effective user interface. Quality is built into software during the design of software. The final word is: Design process should be given due weight age before rushing for coding.

Questions

1. If a software design is not a program, then what is it?
2. Describe software architecture.
3. How do we asses the quality of a software design?

4. A number of high-level programming languages support the internal procedure as a modular construct. How does this construct affect coupling? information hiding?
5. What is the purpose of developing a program structure that is factored?
6. Why is it a good idea to keep the scope of effect of a module within its scope of control?

UNIT – IV

Structure

1. TESTING TACTICS

- 1.1 Introduction
- 1.2 Software Testing Fundamentals
- 1.3 Test Case Design
- 1.4 White-Box Testing
- 1.5 Basis Path Testing
- 1.6 Control Structure Testing
- 1.7 Black-Box Testing
- 1.8 Basic Terms Used in Testing
- 1.4 Testing Activities
- 1.5 Debugging
- 1.6 Testing Tools

2. TESTING STRATEGIES

- 2.1 Introduction
- 2.2 A Strategic Approach to Software Testing
- 2.3 Strategic Issues
- 2.4 Unit Testing
- 2.5 Integration Testing
- 2.6 Validation Testing

2.7 System Testing

Objectives

After going through this unit, you should be able to:

- know the basic terms using in testing terminology;
- black box and White box testing techniques;
- other testing techniques; and
- some testing tools.

1. Testing Tactics

1.1 Introduction

Testing means executing a program in order to understand its behavior, that is, whether or not the program exhibits a failure, its response time or throughput for certain data sets, its mean time to failure, or the speed and accuracy with which users complete their designated tasks. In other words, it is a process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. Testing can also be described as part of the process of Validation and Verification.

Validation is the process of evaluating a system or component during or at the end of the development process to determine if it satisfies the requirements of the system, or, in other words, are we building the correct system? Verification is the process of evaluating a system or component at the end of a phase to determine if it satisfies the conditions imposed at the start of that phase, or, in other words, are we building the system correctly? Software testing gives an important set of methods that can be used to evaluate and assure that a program or system meets its non-functional requirements. To be more specific, software testing means that executing a program or its components in order to assure:

- The correctness of software with respect to requirements or intent;
- The performance of software under various conditions;
- The robustness of software, that is, its ability to handle erroneous inputs and unanticipated conditions;
- The usability of software under various conditions;
- The reliability, availability, survivability or other dependability measures of software; or
- Installability and other facets of a software release.

The purpose of testing is to show that the program has errors. The aim of most testing methods is to systematically and actively locate faults in the program and repair them. Debugging is the next stage of testing. Debugging is the activity of:

- Determining the exact nature and location of the suspected error within the program and
- Fixing the error. Usually, debugging begins with some indication of the existence of an error.

1.2 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 that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.

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 in human 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.

1.2.1 Testing Objectives

In an excellent book on software testing 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 show only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

1.2.2 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing.:

- 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 problem, 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 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). The software engineer who created the system is not the best person to conduct all tests for the software.

1.2.3 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? 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. Sometimes programmers are willing to do things that will help the testing process and a checklist 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 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 more efficiently 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.
- System states and variables are visible or queryable during execution.
- Past system states and variables are visible or queryable (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 better we can 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 retesting."

- 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 smarter we will 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.

The attributes suggested by Bach can be used by a software engineer to develop a software configuration (i.e., programs, data, and documents) that is amenable to testing. And what about the tests themselves?

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 (discussed in earlier chapters) 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.

1.3 Test Case Design

The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. Yet, for reasons that we have already discussed, software engineers often treat testing as an afterthought, developing test cases that may 'feel right' but have little assurance of being complete. Recalling the objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort.

A rich variety of test case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any engineered product (and most other things) can be tested in one of two ways:

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function; (2) knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach is called black-box testing and the second, white-box testing.

When computer software is considered, 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.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very

large. For example, consider the 100 line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10^{14} possible paths that may be executed in this program!

To put this number in perspective, we assume that a magic test processor ("magic" because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules. Exhaustive testing is impossible for large software systems.

White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity. The attributes of both black- and white-box testing can be combined to provide an approach that validates the software interface and selectively ensures that the internal workings of the software are correct.

1.4 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 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 control 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.

1.5 Basis Path Testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe

IMCC761. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

1.5.1 Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 1.1. Each structured construct has a corresponding flow graph symbol. To illustrate the use of a flow graph, we consider the procedural design representation in Figure 1.2 Here, a flowchart is used to depict program control structure.

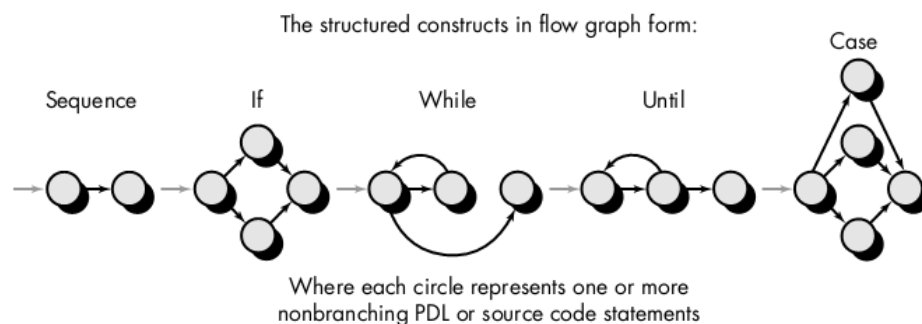
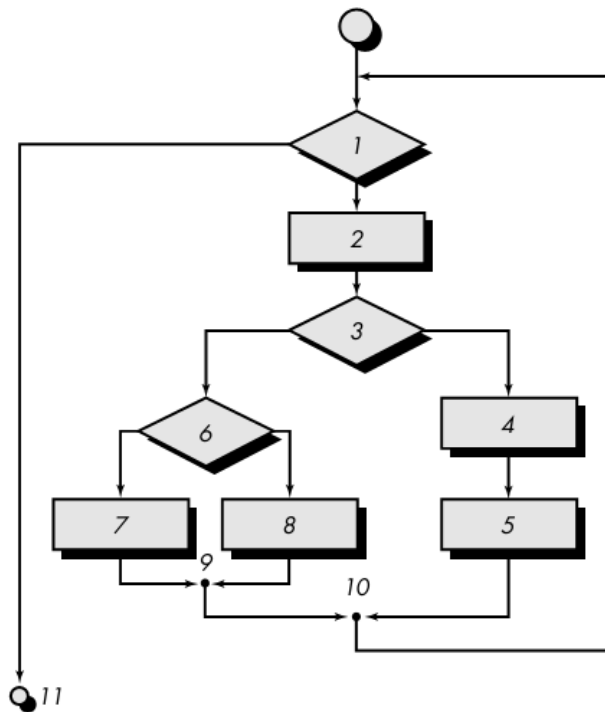


FIGURE 1.1 Flow graph notation

Figure 1.2B maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the

flowchart). Referring to Figure 1.2B, each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.



(A)

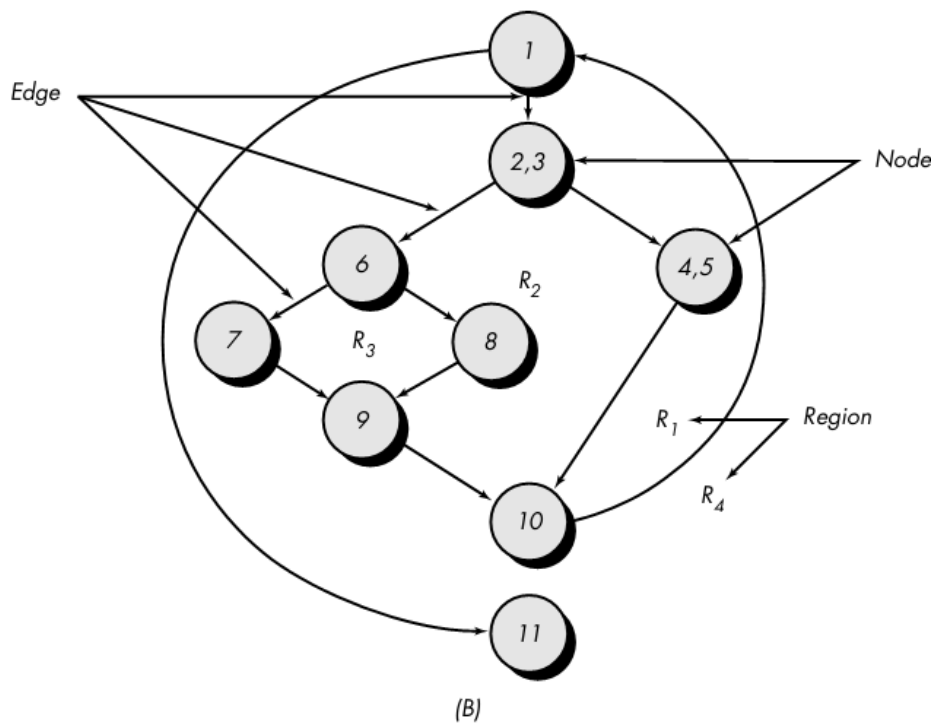


FIGURE 1.2 Flowchart, (A and flow graph (B))

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 1.3, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

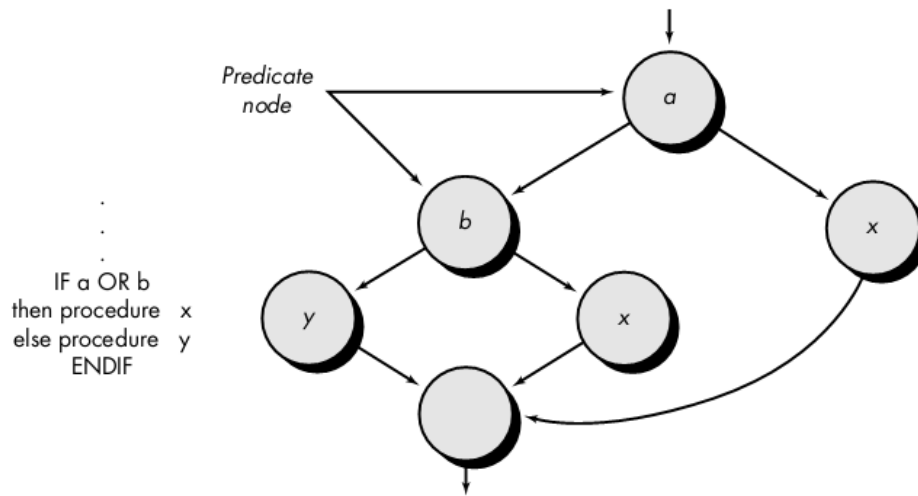


Figure 1.3

1.5.2 Cyclomatic Complexity

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

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 1.2B is

path 1: 1-11

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

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

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

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the flow graph in Figure 1.2B. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.

2. Cyclomatic complexity, $V(G)$, for a flow graph, G , is defined as

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

where E is the number of flow graph edges, N is the number of flow graph nodes.

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

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

where P is the number of predicate nodes contained in the flow graph G .

Referring once more to the flow graph in Figure 1.2 B, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

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

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

Therefore, the cyclomatic complexity of the flow graph in Figure 1.2B is 4.

More important, the value for $V(G)$ provides us with an upper bound for the number of independent paths that form the basis set and, by

implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

1.6 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.

1.6.1 Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (\neg) operator. A relational expression takes the form

E1 <relational-operator> E2

where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (non equality), >, or ≥. A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|) AND (&) and NOT (\neg). A condition without relational expressions is referred to as a Boolean expression. Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

The condition testing method focuses on testing each condition in the program. Condition testing strategies (discussed later in this section) generally have two advantages. First, measurement of test coverage of a condition is simple. Second, the test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program P is effective.

for detecting errors in the conditions contained in F it is likely that this test set is also effective for detecting other errors in P. In addition, if a testing strategy is effective for detecting errors in a condition, then it is likely that this strategy will also be effective for detecting errors in a program.

A number of condition testing strategies have been proposed. Branch testing is probably the simplest condition testing strategy. For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once.

Domain testing requires three or four tests to be derived for a relational expression. For a relational expression of the form

$E1 <\text{relational-operator}> E2$

three tests are required to make the value of E1 greater than, equal to, or less than that of E2. If $<\text{relational-operator}>$ is incorrect and E1 and E2 are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in E1 and E2, a test that makes the value of E1 greater or less than that of E2 should make the difference between these two values as small as possible.

For a Boolean expression with n variables, all of 2^n possible tests are required ($n > 0$). This strategy can detect Boolean operator, variable, and parenthesis errors, but it is practical only if n is small.

Error-sensitive tests for Boolean expressions can also be derived. For a singular Boolean expression (a Boolean expression in which each Boolean variable occurs only once) with n Boolean variables ($n > 0$), we can easily generate a test set with less than 2^n tests such that this test set guarantees the detection of multiple Boolean operator errors and is also effective for detecting other errors.

Tai suggests a condition testing strategy that builds on the techniques just outlined. Called BRO (branch and relational operator) testing, the technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variables.

1.6.2 Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. A number of data flow testing strategies have been studied and compared.

To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

$$\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains a definition of } X\}$$
$$\text{USE}(S) = \{X \mid \text{statement } S \text{ contains a use of } X\}$$

If statement S is a for loop statement, its DEF set is empty and its USE set is based on the condition of statement S . The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X .

A definition use (DU) chain of variable X is of the form $[X, S, S']$, where S and S' are statement numbers, X is in $\text{DEF}(S)$ and $\text{USE}(S)$, and the definition of X in statement S is live at statement S' .

One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the then part has no definition of any variable and the else part does not exist. In this situation, the else branch of the if statement is not necessarily covered by DU testing.

Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements. To illustrate this, consider the application of DU testing to select test paths for the PDL that follows:

```
proc x
  B1;
  do while C1
    if C2
      then
        if C4
          then B4;
          else B5;
        endif;
      else
        if C3
          then B2;
          else B3;
        endif;
      endif;
    enddo;
  B6;
end proc;
```

To apply the DU testing strategy to select test paths of the control flow diagram, we need to know the definitions and uses of variables in each condition or block in the PDL. Assume that variable X is defined in the last statement of blocks B₁, B₂, B₃, B₄, and B₅ and is used in the first statement of blocks B₂, B₃, B₄, B₅, and B₆. The DU testing strategy requires an execution of the shortest path from each of B_i $0 < i \leq 5$, to each of B_j $1 < j \leq 6$. (Such testing also covers any use of variable X in conditions C₁, C₂, C₃, and C₄.) Although there are 25 DU chains of variable X, we need only five paths to cover these DU chains. The reason is that five paths are needed to cover the DU chain of X from B_i $0 < i \leq 5$, to B₆ and other DU chains can be covered by making these five paths contain iterations of the loop.

If we apply the branch testing strategy to select test paths of the PDL just noted, we do not need any additional information. To select paths of the diagram for BRO testing, we need to know the structure of each condition or block. (After the selection of a path of a program, we need to determine whether the path is feasible for the program, that is, whether at least one input exists that exercises the path.)

Since the statements in a program are related to each other according to the definitions and uses of variables, the data flow testing approach is effective for error detection. However, the problems of measuring test coverage and selecting test paths for data flow testing are more difficult than the corresponding problems for condition testing.

1.6.3 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 1.4).

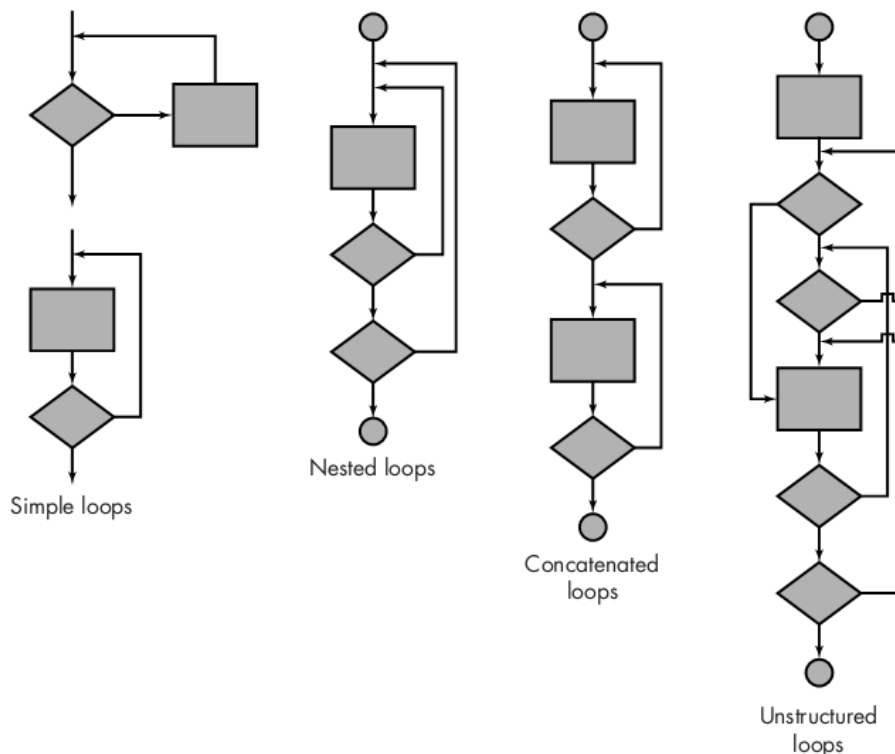


FIGURE 1.4 Classes of loops

Simple loops: The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one passes through the loop.
3. Two passes through the loop.
4. n passes through the loop where $n < n$.
5. $n - 1$, n , $n + 1$ passes through the loop.

Nested loops: If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
4. Continue until all loops have been tested.

Concatenated loops: Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

1.7 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 (see Chapter 18). 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.

1.7.1 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.

The symbolic representation of a graph is shown in Figure 1.5A. Nodes are represented as circles connected by links that take a number of different forms. A directed link (represented by an arrow) indicates that a relationship moves in only one direction. A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 1.5B) where

Object #1 = new file menu select

Object #2 document window

Object #3 document text

Referring to the figure, a menu select on new file generates a document window. The node weight of document window provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the new file menu select and document text, and parallel links indicate relationships between document window and document text. In reality, a far more detailed graph would have to be generated as a precursor to test case design. The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

Beizer describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps (e.g., flight.information.input is followed by validation /availability.proccssing).

The data flow diagram can be used to assist in creating graphs of this type.

Finite state modeling. The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., order-information is verified during inventory-available y look-up and is followed by customer-billing-information input). The state transition diagram can be used to assist in creating graphs of this type.

Data flow modeling. The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node FICA.tax.withheld (FTW) is computed from

gross.wages (GW) using the relationship, $FTW = 0.62 \times GW$.

Timing modeling. The nodes are program objects and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. The interested reader should see for a comprehensive discussion. It is worthwhile, however, to provide a generic outline of the graph-based testing approach.

Graph-based testing begins with the definition of all nodes and node weights. That is, objects and attributes are identified. The data model can be used as a starting point, but it is important to note that many nodes may be program objects (not explicitly represented in the data model). To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes.

Once nodes have been identified, links and link weights should be established. In general, links should be named, although links that represent control flow between program objects need not be named.

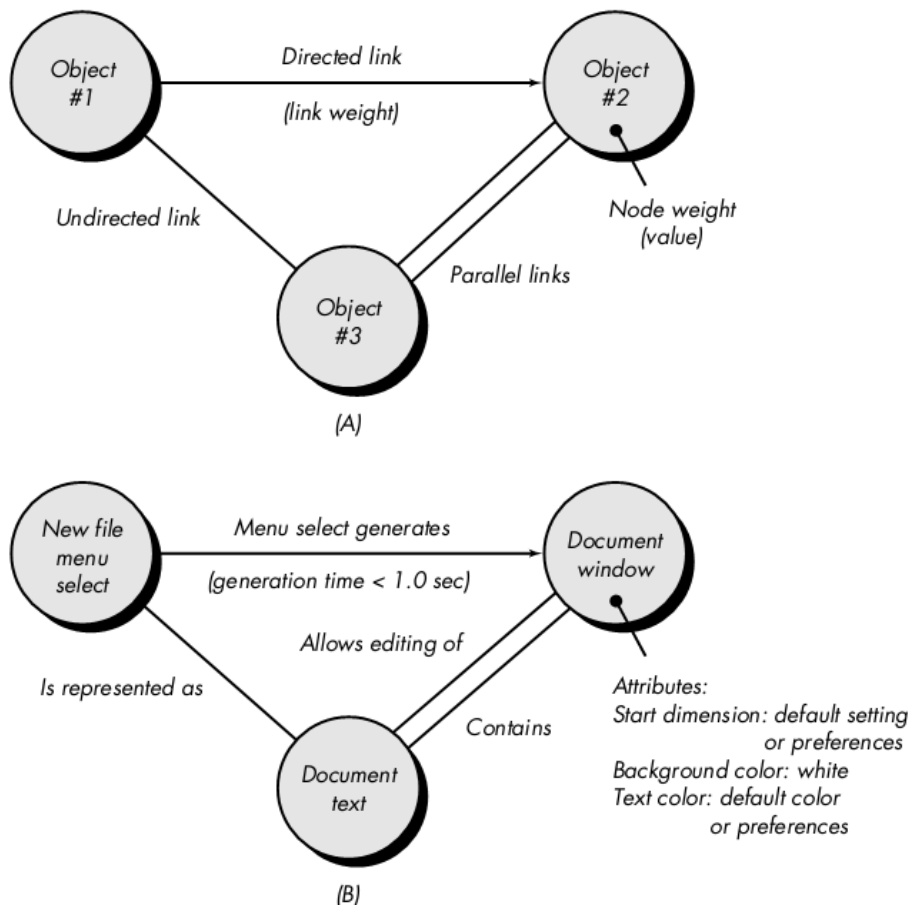


FIGURE 1.5 (A) Graph notation (B) Simple example

1.7.2 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 classes 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 [BE1951]. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a 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 class 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 application accepts 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 application 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 input domain 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.

1.7.3 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.

1.7.4 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.

1.7.5 Orthogonal Array Testing

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test processing of the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive

testing. The orthogonal array testing method is particularly useful in finding errors associated with region faults an error category associated with faulty logic within a software component.

1.8 Basic Terms Used in Testing

Failure: A failure occurs when there is a deviation of the observed behavior of a program, or system, from its specification. A failure can also occur if the observed behavior of a system, or program, deviates from its intended behavior.

Fault: A fault is an incorrect step, process, or data definition in a computer program. Faults are the source of failures. In normal language, software faults are usually referred to as “bugs”.

Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Test Cases: Ultimately, testing comes down to selecting and executing test cases. A test case for a specific component consists of three essential pieces of information:

- A set of test inputs;
- The expected results when the inputs are executed; and
- The execution conditions or environments in which the inputs are to be executed.

Some Testing Laws

- Testing can only be used to show the presence of errors, but never the absence of errors.
- A combination of different verification & validation (V&V) methods outperform any single method alone.
- Developers are unsuited to test their own code.
- Approximately 80% of the errors are found in 20% of the code.
- Partition testing, that is, methods that partition the input domain or the program and test according to those partitions. This is better than random testing.
- The adequacy of a test suite for coverage criterion can only be defined intuitively.

1.8.1 Input Domain

To conduct an analysis of the input, the sets of values making up the input domain are required. There are essentially two sources for the input domain. They are:

1. The software requirements specification in the case of black box testing method; and
2. The design and externally accessible program variables in the case of white box testing.

In the case of white box testing, input domain can be constructed from the following sources.

- Inputs passed in as parameters; Variables that are inputs to function under test can be:

(i) Structured data such as linked lists, files or trees, as well as atomic data such as integers and floating point numbers;

(ii) A reference or a value parameter as in the C function declaration `int P(int *power, int base) {`

`...}`

- Inputs entered by the user via the program interface;
- Inputs that are read in from files;
- Inputs that are constants and precomputed values; Constants declared in an enclosing scope of function under test, for example,

```
#define PI 3.14159
```

```
double circumference(double radius)
```

```
{
```

```
return 2*PI*radius;
```

```
}
```

In general, the inputs to a program or a function are stored in program variables. A program variable may be:

- A variable declared in a program as in the C declarations

For example: `int base; char s[];`

- Resulting from a read statement or similar interaction with the environment, For example: `scanf("%d\n", &x);`

1.8.2 Black Box and White Box Test Case Selection Strategies

- Black box Testing: In this method, where test cases are derived from the functional specification of the system; and

- White box Testing: In this method, where test cases are derived from the internal design specifications or actual code (Sometimes referred to as Glass box).

Black box test case selection can be done without any reference to the program design or the program code. Test case selection is only concerned with the functionality and features of the system but not with its internal operations.

- The real advantage of black box test case selection is that it can be done before the design or coding of a program. Black box test cases can also help to get the design and coding correct with respect to the specification. Black box testing methods are good at testing for missing functions or program behavior that deviates from the specification. Black box testing is ideal for evaluating products that you intend to use in your systems.

- The main disadvantage of black box testing is that black box test cases cannot detect additional functions or features that have been added to the code. This is especially important for systems that need to be safe (additional code may interfere with the safety of the system) or secure (additional code may be used to break security).

White box test cases are selected using the specification, design and code of the program or functions under test. This means that the testing team needs access to the internal designs or code for the program.

- The chief advantage of white box testing is that it tests the internal details of the code and tries to check all the paths that a program can execute to determine if a problem occurs. White box testing can check additional functions or code that has been implemented, but not specified.

- The main disadvantage of white box testing is that you must wait until after design and coding of the programs or functions under test have been completed in order to select test cases.

Methods for Black box testing strategies

A number of test case selection methods exist within the broad classification of black box and white box testing.

For Black box testing strategies, the following are the methods:

- Boundary-value Analysis;
- Equivalence Partitioning.

We will also study State Based Testing, which can be classified as opaque box selection strategies that is somewhere between black box and white box selection strategies.

Boundary-value-analysis

The basic concept used in Boundary-value-analysis is that if the specific test cases are designed to check the boundaries of the input domain then the probability of detecting an error will increase. If we want to test a program written as a function F with two input variables x and y , then these input variables are defined with some boundaries like $a1 \leq x \leq a2$ and $b1 \leq y \leq b2$. It means that inputs x and y are bounded by two intervals $[a1, a2]$ and $[b1, b2]$.

Test Case Selection Guidelines for Boundary Value Analysis

The following set of guidelines is for the selection of test cases according to the principles of boundary value analysis. The guidelines do not constitute a firm set of rules for every case. You will need to develop some judgment in applying these guidelines.

1. If an input condition specifies a range of values, then construct valid test cases for the ends of the range, and invalid input test cases for input points just beyond the ends of the range.
2. If an input condition specifies a number of values, construct test cases for the minimum and maximum values; and one beneath and beyond these values.
3. If an output condition specifies a range of values, then construct valid test cases for the ends of the output range, and invalid input test cases for situations just beyond the ends of the output range.
4. If an output condition specifies a number of values, construct test cases for the minimum and maximum values; and one beneath and beyond these values.
5. If the input or output of a program is an ordered set (e.g., a sequential file, linear list, table), focus attention on the first and last elements of the set.

Example: Boundary Value Analysis for the Triangle Program

Consider a simple program to classify a triangle. Its input consists of three positive integers (say x, y, z) and the data types for input parameters ensures that these will be integers greater than zero and less than or equal to 100. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled.

Solution: Following possible boundary conditions are formed:

1. Given sides (A; B; C) for a scalene triangle, the sum of any two sides is greater than the third and so, we have boundary conditions $A + B > C$, $B + C > A$ and $A + C > B$.

2. Given sides (A; B; C) for an isosceles triangle two sides must be equal and so we have boundary conditions $A = B$, $B = C$ or $A = C$.

3. Continuing in the same way for an equilateral triangle the sides must all be of equal length and we have only one boundary where $A = B = C$.

4. For right-angled triangles, we must have $A^2 + B^2 = C^2$.

On the basis of the above boundary conditions, test cases are designed as follows (Table):

Table: Test cases for Example

Test case	x	y	z	Expected Output
1	100	100	100	Equilateral/ triangle
2	50	3	50	Isosceles triangle
3	40	50	40	Equilateral/ triangle
4	3	4	5	Right-angled triangles
5	10	10	10	Equilateral/ triangle
6	2	2	5	Isosceles triangle
7	100	50	100	Scalene triangle
8	1	2	3	Non-triangles
9	2	3	4	Scalene triangle
10	1	3	1	Isosceles triangle

Equivalence Partitioning

Equivalence Partitioning is a method for selecting test cases based on a partitioning of the input domain. The aim of equivalence partitioning is to divide the input domain of the program or module into classes (sets) of test cases that have a similar effect on the program. The classes are called Equivalence classes.

Equivalence Classes

An Equivalence Class is a set of inputs that the program treats identically when the program is tested. In other words, a test input taken from an equivalence class is representative of all of the test inputs taken from that class. Equivalence classes are determined from the specification of a program or module. Each equivalence class is used to represent

certain conditions (or predicates) on the input domain. For equivalence partitioning it is usual to also consider valid and invalid inputs. The terms input conditions, valid and invalid inputs, are not used consistently. But, the following definition spells out how we will use them in this subject. An input condition on the input domain is a predicate over the values of the input domain. A Valid input to a program or module is an element of the input domain that is expected to return a non-error value. An Invalid input is an input that is expected to return an error value. Equivalence partitioning is then a systematic method for identifying interesting input conditions to be tested. An input condition can be applied to a set of values of a specific input variable, or a set of input variables as well.

A Method for Choosing Equivalence Classes

The aim is to minimize the number of test cases required to cover all of the identified equivalence classes. The following are two distinct steps in achieving this goal:

Step 1: Identify the equivalence classes

If an input condition specifies a range of values, then identify one valid equivalence class and two invalid equivalence classes.

For example, if an input condition specifies a range of values from 1 to 99, then, three equivalence classes can be identified:

- One valid equivalence class: $1 < X < 99$
- Two invalid equivalence classes $X < 1$ and $X > 99$

Step 2: Choose test cases

The next step is to generate test cases using the equivalence classes identified in the previous step. The guideline is to choose test cases on the boundaries of partitions and test cases close to the midpoint of the partition. In general, the idea is to select at least one element from each equivalence class.

Example: Selecting Test Cases for the Triangle Program

In this example, we will select a set of test cases for the following triangle program based on its specification. Consider the following informal specification for the Triangle Classification Program. The program reads three integer values from the standard input. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled. The specification of the triangle classification program lists a number of inputs for the program as well as the form of output. Further, we

require that each of the inputs “must be” a positive integer. Now, we can determine valid and invalid equivalence classes for the input conditions. Here, we have a range of values. If the three integers we have called x, y and z are all greater than zero, then, they are valid and we have the equivalence class.

$$EC_{\text{valid}} = f(x, y, z) \quad x > 0 \text{ and } y > 0 \text{ and } z > 0.$$

For the invalid classes, we need to consider the case where each of the three variables in turn can be negative and so we have the following equivalence classes:

$$EC_{\text{Invalid1}} = f(x, y, z) \quad x < 0 \text{ and } y > 0 \text{ and } z > 0$$

$$EC_{\text{Invalid2}} = f(x, y, z) \quad x > 0 \text{ and } y < 0 \text{ and } z > 0$$

$$EC_{\text{Invalid3}} = f(x, y, z) \quad x > 0 \text{ and } y > 0 \text{ and } z < 0$$

Note that we can combine the valid equivalence classes. But, we are not allowed to combine the invalid equivalence classes. The output domain consists of the text ‘strings’ ‘isosceles’, ‘scalene’, ‘equilateral’ and ‘right-angled’. Now, different values in the input domain map to different elements of the output domain to get the equivalence classes in Table. According to the equivalence partitioning method we only need to choose one element from each of the classes above in order to test the triangle program.

Table: The equivalence classes for the triangle program

Equivalence class	Test Inputs	Expected Outputs
EC _{scalene}	f(3, 5, 7), . . . g	“Scalene”
EC _{isosceles}	f(2, 3, 3), . . . g f(2, 3, 3), . . . g	“Isosceles”
EC _{equilateral}	f(7, 7, 7), . . . g f(7, 7, 7), . . . g	“Equilateral”
EC _{right angled}	f(3, 4, 5), . . .	“Right Angled”
EC _{non_triangle}	f(1, 1, 3), . . .	“Not a Triangle”
EC _{invalid1}	f(-1, 2, 3), (0, 1, 3), . . .	“Error Value”
EC _{invalid2}	f(1, -2, 3), (1, 0, 3), . . .	“Error Value”
EC _{invalid3}	f(1, 2, -3), (1, 2, 0), . . .	“Error Value”

Methods for White box testing strategies

In this approach, complete knowledge about the internal structure of the source code is required. For White-box testing strategies, the methods are:

1. Coverage Based Testing
2. Cyclomatic Complexity

3. Mutation Testing

Coverage based testing

The aim of coverage based testing methods is to 'cover' the program with test cases that satisfy some fixed coverage criteria. Put another way, we choose test cases to exercise as much of the program as possible according to some criteria.

Coverage Based Testing Criteria

Coverage based testing works by choosing test cases according to well-defined 'coverage' criteria. The more common coverage criteria are the following.

- Statement Coverage or Node Coverage: Every statement of the program should be exercised at least once.
- Branch Coverage or Decision Coverage: Every possible alternative in a branch or decision of the program should be exercised at least once. For if statements, this means that the branch must be made to take on the values true or false.
- Decision/Condition Coverage: Each condition in a branch is made to evaluate to both true and false and each branch is made to evaluate to both true and false.
- Multiple condition coverage: All possible combinations of condition outcomes within each branch should be exercised at least once.
- Path coverage: Every execution path of the program should be exercised at least once.

In this section, we will use the control flow graph to choose white box test cases according to the criteria above. To motivate the selection of test cases, consider the simple program given in Program.

Example:

```
void main(void)
{
int x1, x2, x3;

scanf("%d %d %d", &x1, &x2, &x3);

if ((x1 > 1) && (x2 == 0))
```

```
x3 = x3 / x1;  
if ((x1 == 2) || (x3 > 1))  
x3 = x3 + 1;  
while (x1 >= 2)  
x1 = x1 - 2;  
printf("%d %d %d", x1, x2, x3);  
}
```

Program: A simple program for white box testing

The first step in the analysis is to generate the flow chart, which is given in Figure 1.6. Now what is needed for statement coverage? If all of the branches are true, at least once, we will have executed every statement in the flow chart. Put another way to execute every statement at least once, we must execute the path ABCDEFGF. Now, looking at the conditions inside each of the three branches, we derive a set of constraints on the values of x1, x2 and x3 such that all the three branches are extended. A test case of the form (x1; x2; x3) = (2; 0; 3) will execute all of the statements in the program.

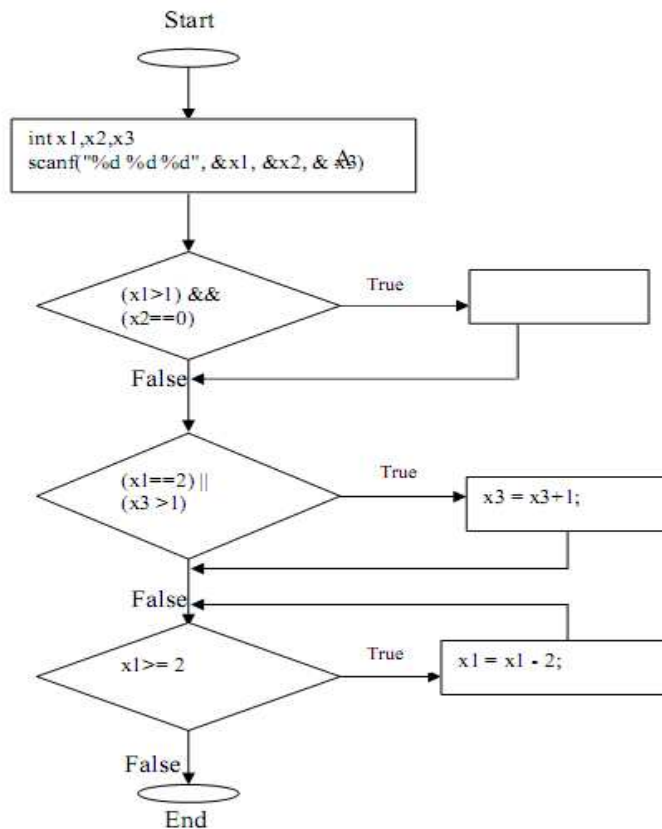


Figure 1.6: The flow chart for the program

Note that we need not make every branch evaluate to true and false, nor have we make every condition evaluate to true and false, nor we traverse every path in the program.

To make the first branch true, we have test input (2; 0; 3) that will make all of the branches true. We need a test input that will now make each one false. Again looking at all of the conditions, the test input (1; 1; 1) will make all of the branches false.

For any of the criteria involving condition coverage, we need to look at each of the five conditions in the program: C1 = (x1>1), C2 = (x2 == 0), C3 = (x1 == 2), C4 = (x3>1) and C5 = (x1 >= 2). The test input (1; 0; 3) will make C1 false, C2 true, C3 false, C4 true and C5 false.

Examples of sets of test inputs and the criteria that they meet are given in Table. The set of test cases meeting the multiple condition criteria is given in Table. In the table, we let the branches B1 = C1&&C2, B2 = C3||C4 and B3 = C5.

Table: Test cases for the various coverage criteria for the program

Coverage Criteria	Test Inputs (x1, x2, x3)	Execution Paths
Statement	(2, 0, 3)	ABCDEF GF
Branch	(2, 0, 3), (1, 1, 1)	ABCDEF GF ABDF
Condition	(1, 0, 3), (2, 1, 1)	ABDEF ABDFGF
Decision/ Condition	(2, 0, 4), (1, 1, 1)	ABCDEF GF ABDF
Multiple Condition	(2, 0, 4), (2, 1, 1), (1, 0, 2), (1, 1, 1)	ABCDEF GF ABDFGF ABDEF ABDF
Path	(2, 0, 4), (2, 1, 1), (1, 0, 2), (4, 0, 0),	ABCDEF GF ABDFGF ABDEF ABCDFGFGF

Table: Multiple condition coverage for the program in Figure

Test cases	C1 x1 > 1	C2 x2 == 0	B1	C3 x1 == 2	C4 x3 > 1	B2	B3 C5 x1 ≥ 2
(1,0,3)	F	T	F	F	T	T	F
(2,1,1)	T	F	F	T	F	F	T
(2,0,4)	T	T	T	T	T	T	T
(1,1,1)	F	F	F	F	F	F	F
(2,0,4)	T	T	T	T	T	T	T
(2,1,1)	T	F	F	T	F	T	T
(1,0,2)	F	T	F	F	T	T	F
(1,1,1)	F	F	F	F	F	F	F

1.8.3 Cyclomatic Complexity Control flow graph (CFG)

A control flow graph describes the sequence in which different instructions of a program get executed. It also describes how the flow of control passes through the program. In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph. An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. Following structured programming constructs are represented as CFG:



FIGURE 1.7: sequence

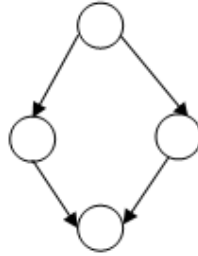


FIGURE 1.8: if – else

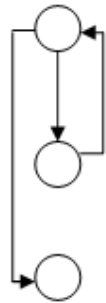


Figure 1.9 : while-loop

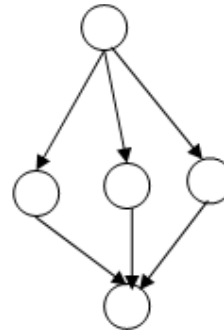


FIGURE 1.10: case

Example: Draw CFG for the program given below.

```

int sample (a,b)
int a,b;
{
1 while (a!= b) {
2 if (a > b)
3 a = a-b;
4 else b = b-a;}
5 return a;
}
  
```

Program: A program in the above, two control constructs are used, namely, while-loop and if-then-else. A complete CFG for the program of Program is given below: Figure 1.11).

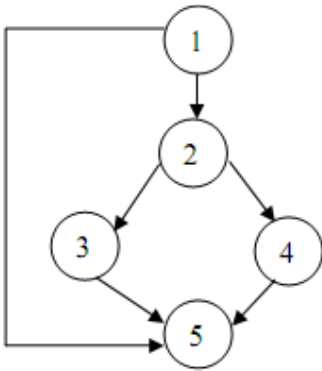


Figure 1.11: CFG for program

Cyclomatic Complexity: This technique is used to find the number of independent paths through a program. If CFG of a program is given, the Cyclomatic complexity $V(G)$ can be computed as: $V(G) = E - N + 2$, where N is the number of nodes of the CFG and E is the number of edges in the CFG. For the example 4, the Cyclomatic Complexity = $6 - 5 + 2 = 3$.

The following are the properties of Cyclomatic complexity:

- $V(G)$ is the maximum number of independent paths in graph G
- Inserting and deleting functional statements to G does not affect $V(G)$
- G has only one path if and only if $V(G) = 1$.

Mutation Testing

Mutation Testing is a powerful method for finding errors in software programs. In this technique, multiple copies of programs are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program. The mutant that is detected by a test case is termed “killed” and the goal of the mutation procedure is to find a set of test cases that are able to kill groups of mutant programs. Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. It is essential that all mutants must be killed by the test cases or shown to be equivalent to the original expression. If we run a mutated program, there are two possibilities:

1. The results of the program were affected by the code change and the test suite detects it. We assumed that the test suite is perfect, which means that it must detect the change. If this happens, the mutant is called a killed mutant.

2. The results of the program are not changed and the test suite does not detect the mutation. The mutant is called an equivalent mutant.

If we take the ratio of killed mutants to all the mutants that were created, we get a number that is smaller than 1. This number gives an indication of the sensitivity of program to the changes in code. In real life, we may not have a perfect program and we may not have a perfect test suite. Hence, we can have one more scenario:

3. The results of the program are different, but the test suite does not detect it because it does not have the right test case.

Consider the following program:

```
main(argc, argv)          /* line 1 */
int argc;                 /* line 2 */
char *argv[];            /* line 3 */
{                          /* line 4 */
    int c=0;              /* line 5 */
                          /* line 6 */
    if(atoi(argv[1]) < 3){ /* line 7 */
        printf("Got less than 3\n"); /* line 8 */
        if(atoi(argv[2]) > 5) /* line 9 */
            c = 2; /* line 10 */
    } /* line 11 */
    else /* line 12 */
        printf("Got more than 3\n"); /* line 13 */
    exit(0); /* line 14 */
} /* line 15 */
```

Program: The program reads its arguments and prints messages accordingly. Now let us assume that we have the following test suite that tests the program:

Test case 1:

Input: 2 4

Output: Got less than 3

Test case 2:

Input: 4 4

Output: Got more than 3

Test case 3:

Input: 4 6

Output: Got more than 3

Test case 4:

Input: 2 6

Output: Got less than 3

Test case 5:

Input: 4

Output: Got more than 3

Now, let's mutate the program. We can start with the following simple changes:

Mutant 1: change line 9 to the form

```
if(atoi(argv[2]) <= 5)
```

Mutant 2: change line 7 to the form

```
if(atoi(argv[1]) >= 3)
```

Mutant 3: change line 5 to the form

```
int c=3;
```

If we take the ratio of all the killed mutants to all the mutants generated, we get a number smaller than 1 that also contains information about accuracy of the test suite. In practice, there is no way to separate the effect that is related to test suite inaccuracy and that which is related to equivalent mutants. In the absence of other possibilities, one can accept

the ratio of killed mutants to all the mutants as the measure of the test suite accuracy. The manner by which a test suite is evaluated via mutation testing is as follows: For a specific test suite and a specific set of mutants, there will be three types of mutants in the code (i) killed or dead (ii) live (iii) equivalent. The score (evaluation of test suite) associated with a test suite T and mutants M is simply computed as follows:

killed Mutants

total mutants - # equivalent mutants

1.9 Testing Activities

Although testing varies between organisations, there is a cycle to testing:

Requirements Analysis: Testing should begin in the requirements phase of the software development life cycle (SDLC).

Design Analysis: During the design phase, testers work with developers in determining what aspects of a design are testable and under what parameters should the testers work.

Test Planning: Test Strategy, Test Plan(s).

Test Development: Test Procedures, Test Scenarios, Test Cases, Test Scripts to use in testing software.

Test Execution: Testers execute the software based on the plans and tests and report any errors found to the development team.

Test Reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.

Retesting the Defects: Defects are once again tested to find whether they got eliminated or not.

Levels of Testing:

Mainly, Software goes through three levels of testing:

- Unit testing
- Integration testing
- System testing.

Unit Testing

Unit testing is a procedure used to verify that a particular segment of source code is working properly. The idea about unit tests is to write test cases for all functions or methods. Ideally, each test case is separate from the others. This type of testing is mostly done by developers and not by end users. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. Unit testing provides a strict, written contract that the piece of code must satisfy. Unit testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves. Therefore, it will not catch integration errors, performance problems and any other system-wide issues. A unit test can only show the presence of errors; it cannot show the absence of errors.

Integration Testing

Integration testing is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing. Integration testing takes as its input, modules that have been checked out by unit testing, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output, the integrated system ready for system testing. The purpose of Integration testing is to verify functional, performance and reliability requirements placed on major design items.

System Testing

System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). In system testing, the entire system can be tested as a whole against the software requirements specification (SRS). There are rules that describe the functionality that the vendor (developer) and a customer have agreed upon. System testing tends to be more of an investigatory testing phase, where the focus is to have a destructive attitude and test not only the design, but also the behavior and even the believed expectations of the customer. System testing is intended to test up to and beyond the bounds defined in the software requirements specifications.

Acceptance tests are conducted in case the software developed was a custom software and not product based. These tests are conducted by customer to check whether the software meets all requirements or not. These tests may range from a few weeks to several months.

1.10 Debugging

Debugging occurs as a consequence of successful testing. Debugging refers to the process of identifying the cause for defective

behavior of a system and addressing that problem. In less complex terms fixing a bug. When a test case uncovers an error, debugging is the process that results in the removal of the error. The debugging process begins with the execution of a test case. The debugging process attempts to match symptoms with cause, thereby leading to error correction. The following are two alternative outcomes of the debugging:

1. The cause will be found and necessary action such as correction or removal will be taken.
2. The cause will not be found.

Characteristics of bugs

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Life Cycle of a Debugging Task

The following are various steps involved in debugging:

a) Defect Identification/Confirmation

- A problem is identified in a system and a defect report created
- Defect assigned to a software engineer

- The engineer analyzes the defect report, performing the following actions:
- What is the expected/desired behaviour of the system?
- What is the actual behaviour?
- Is this really a defect in the system?
- Can the defect be reproduced? (While many times, confirming a defect is straight forward. There will be defects that often exhibit quantum behaviour.)

b) Defect Analysis

Assuming that the software engineer concludes that the defect is genuine, the focus shifts to understanding the root cause of the problem. This is often the most challenging step in any debugging task, particularly when the software engineer is debugging complex software.

Many engineers debug by starting a debugging tool, generally a debugger and try to understand the root cause of the problem by following the execution of the program step-by-step. This approach may eventually yield success. However, in many situations, it takes too much time, and in some cases is not feasible, due to the complex nature of the program(s).

c) Defect Resolution

Once the root cause of a problem is identified, the defect can then be resolved by making an appropriate change to the system, which fixes the root cause.

Debugging Approaches

Three categories for debugging approaches are:

- Brute force
- Backtracking
- Cause elimination.

Brute force is probably the most popular despite being the least successful. We apply brute force debugging methods when all else fails. Using a “let the computer find the error” technique, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. Backtracking is a common debugging method that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backwards till the error is found. In Cause elimination, lists of possible causes of an error are identified and tests are conducted until each one is eliminated.

1.11 Testing Tools

The following are different categories of tools that can be used for testing:

- **Data Acquisition:** Tools that acquire data to be used during testing.
- **Static Measurement:** Tools that analyse source code without executing test cases.
- **Dynamic Measurement:** Tools that analyse source code during execution.
- **Simulation:** Tools that simulate functions of hardware or other externals.
- **Test Management:** Tools that assist in planning, development and control of testing.
- **Cross-Functional tools:** Tools that cross the bounds of preceding categories.

The following are some of the examples of commercial software testing tools:

Rational Test Real Time Unit Testing

- **Kind of Tool**

Rational Test RealTime's Unit Testing feature automates C, C++ software component testing.

- **Organisation**

IBM Rational Software

- **Software Description**

Rational Test RealTime Unit Testing performs black-box/functional testing, i.e., verifies that all units behave according to their specifications without regard to how that functionality is implemented. The Unit Testing feature has the flexibility to naturally fit any development process by matching and automating developers' and testers' work patterns, allowing them to focus on value-added tasks. Rational Test RealTime is integrated with native development environments (Unix and Windows) as well as with a large variety of cross-development environments.

- **Platforms**

Rational Test RealTime is available for most development and target systems including Windows and Unix.

AQtest**• Kind of Tool**

Automated support for functional, unit, and regression testing

• Organisation

AutomatedQA Corp.

• Software Description

AQtest automates and manages functional tests, unit tests and regression tests, for applications written with VC++, VB, Delphi, C++Builder, Java or VS.NET. It also supports white-box testing, down to private properties or methods. External tests can be recorded or written in three scripting languages (VBScript, JScript, DelphiScript). Using AQtest as an OLE server, unit-test drivers can also run it directly from application code. AQtest automatically integrates AQtime when it is on the machine. Entirely COM-based, AQtest is easily extended through plug-ins using the complete IDL libraries supplied. Plug-ins currently support Win32 API calls, direct ADO access, direct BDE access, etc.

• Platforms

Windows 95, 98, NT, or 2000.

csUnit**• Kind of Tool**

“Complete Solution Unit Testing” for Microsoft .NET (freeware)

• Organisation

csUnit.org

• Software Description

csUnit is a unit testing framework for the Microsoft .NET Framework. It targets test driven development using .NET languages such as C#, Visual Basic .NET, and managed C++.

• Platforms

Microsoft Windows

Software Description

Sahi is automation and testing tool for web applications, with the facility to record and playback scripts. Developed in Java and JavaScript, it uses simple JavaScript to execute events on the browser. Features include in-browser controls, text based scripts, Ant support for playback of suites of tests, and multi-threaded playback. It supports HTTP and HTTPS. Sahi runs as a proxy server and the browser needs to use the Sahi server as its proxy. Sahi then injects JavaScript so that it can access elements in the webpage. This makes the tool independent of the website/ web application.

- **Platforms**

OS independent. Needs at least JDK1.4

2. Testing Strategies

2.1 Introduction

A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in the successful construction of software. The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to promote reasonable planning and management tracking as the project progresses.

2.2 A Strategic Approach to Software Testing

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing a set of steps into which we can place specific test case design techniques and testing methods should be defined for the software process. A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

- Testing begins at the component level² 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.

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

2.2.1 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 [BOE81] states this 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). 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.

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 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."

2.2.2 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.

From a psychological point of view, software analysis and design (along with coding) are constructive tasks. The software engineer creates a computer program, its documentation, and related data structures. Like any builder, the software engineer is proud of the edifice that has been built and looks askance at anyone who attempts to tear it down. When testing

commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) destructive. So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than uncovering errors. Unfortunately, errors will be present. And, if the software engineer doesn't find them, the customer will!

There are often a number of misconceptions that can be erroneously inferred from the preceding discussion: (1) that the developer of software should do no testing at all, (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly, (3) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function for which it was designed. In many cases, the developer also conducts integration testing a testing step that leads to the construction (and test) of the complete program structure. Only after the software architecture is complete does an independent test group become involved.

The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. After all, personnel in the independent group team are paid to find errors. However, the software engineer doesn't turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered. The ITG is part of the software development project team in the sense that it becomes involved during the specification activity and stays involved (planning and specifying test procedures) throughout a large project. However, in many cases the ITG reports to the software quality assurance organization, thereby achieving a degree of independence that might not be possible if it were a part of the software engineering organization.

2.2.3 A Software Testing Strategy

The software engineering process may be viewed as the spiral illustrated in Figure 2.1. 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 (Figure 2.1). 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 by moving outward 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 streamlines that broaden the scope of testing with each turn.

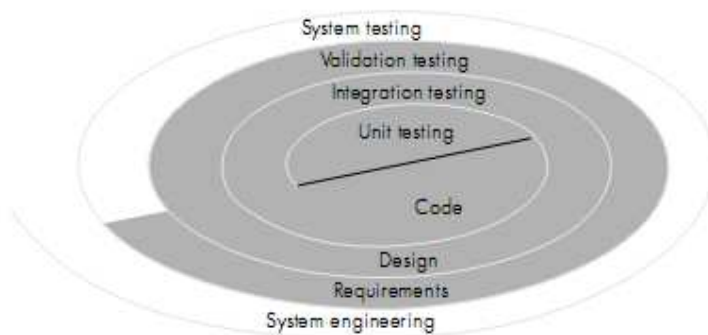


FIGURE 2.1 Testing strategy

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 2.2. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence the name 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.

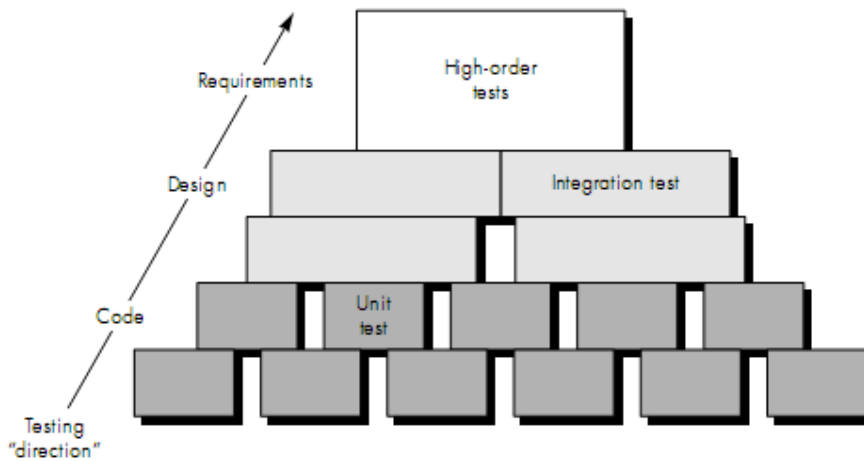


FIGURE 2.2 Software testing steps

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, and databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

2.2.4 Criteria for Completion of Testing

A classic question arises every time software testing is discussed: "When are we done testing how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

One response to the question is: "You're never done testing, the burden simply shifts from you (the software engineer) to your customer." Every time the customer/user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities.

Another response (somewhat cynical but nonetheless accurate) is: "You're done testing when you run out of time or you run out of money."

Although few practitioners would argue with these responses, a software engineer needs more rigorous criteria for determining when sufficient testing has been conducted. Musa and Ackerman suggest a response that is based on statistical criteria: "No, we cannot be absolutely certain that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95 percent confidence that the probability of 1000 CPU hours of failure free operation in a probabilistically defined environment is at least 0.995."

Using statistical modeling and software reliability theory, models of software failures (uncovered during testing) as a function of execution time can be developed. A version of the failure model, called a logarithmic Poisson execution-time model, takes the form

$$f(t) = (1/p) \ln [l_0 pt + 1]$$

where $f(t)$ = cumulative number of failures that are expected to occur once the software has been tested for a certain amount of execution time, t ,

l_0 = the initial software failure intensity (failures per time unit) at the beginning of testing,

p = the exponential reduction in failure intensity as errors are uncovered and repairs are made. The instantaneous failure intensity,

$l(t)$ can be derived by taking the derivative of $f(t)$

$$l(t) = l_0 / (l_0 pt + 1)$$

Using the relationship noted in Equation, testers can predict the drop-off of errors as testing progresses. The actual error intensity can be plotted against the predicted curve (Figure 2.3). If the actual data gathered during testing and the logarithmic Poisson execution time model are reasonably close to one another over a number of data points, the model can be used to predict total testing time required to achieve acceptably low failure intensity.

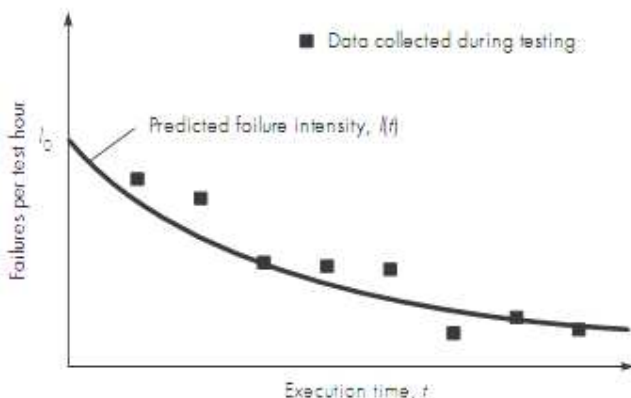


FIGURE 2.3 Failure intensity as a function of execution time

By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?" There is little debate that further work remains to be done before quantitative rules for testing can be established, but the empirical approaches that currently exist are considerably better than raw intuition.

2.3 Strategic Issues

The following issues must be addressed if a successful software testing strategy is to be implemented:

Specify product requirements in a quantifiable manner long before testing commences: Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability (Chapter 19). These should be specified in a way that is measurable so that testing results are unambiguous.

State testing objectives explicitly: The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, mean time to failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours per regression test all should be stated within the test plan.

Understand the users of the software and develop a profile for each user category: Use-cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.

Develop a testing plan that emphasizes “rapid cycle testing.” A software engineering team “learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field ‘trialable,’ increments of functionality and/or quality improvement.” The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Build “robust” software that is designed to test itself: Software should be designed in a manner that uses antidebugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.

Use effective formal technical reviews as a filter prior to testing: 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.

Conduct formal technical reviews to assess the test strategy and test cases themselves: Formal technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.

Develop a continuous improvement approach for the testing process: The test strategy should be measured. The metrics collected during testing

should be used as part of a statistical process control approach for software testing.

2.4 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.

2.4.1 Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure 2.4. 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.

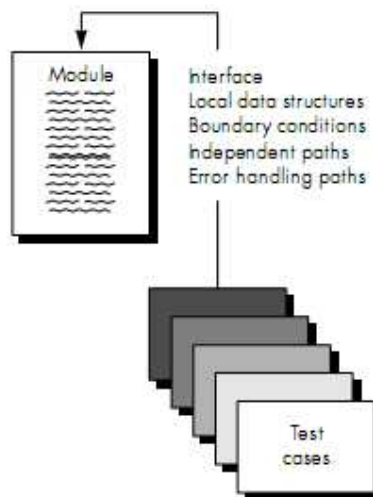


FIGURE 2.4 Unit test

Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing. Selective testing of execution paths is an essential task during

the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

Among the more common errors in computation are (1) misunderstood or incorrect arithmetic precedence, (2) mixed mode operations, (3) incorrect initialization, (4) precision inaccuracy, (5) incorrect symbolic representation of an expression.

Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as (1) comparison of different data types, (2) incorrect logical operators or precedence, (3) expectation of equality when precision error makes equality unlikely, (4) incorrect comparison of variables, (5) improper or nonexistent loop termination, (6) failure to exit when divergent iteration is encountered, and (7) improperly modified loop variables.

Good design dictates that error conditions be anticipated and error-handling paths set up to reroute or cleanly terminate processing when an error does occur. Unfortunately, there is a tendency to incorporate error handling into software and then never test it. A true story may serve to illustrate:

A major interactive design system was developed under contract. In one transaction processing module, a practical joker placed the following error handling message after a series of conditional tests that invoked various control flow branches: ERROR! THERE IS NO WAY YOU CAN GET HERE. This "error message" was uncovered by a customer during user training!

Among the potential errors that should be tested when error handling is evaluated are

1. Error description is unintelligible.
2. Error noted does not correspond to error encountered.
3. Error condition causes system intervention prior to error handling.
4. Exception-condition processing is incorrect.
5. Error description does not provide enough information to assist in the location of the cause of the error.

Boundary testing is the last (and probably most important) task of the unit test step. Software often fails at its boundaries. That is, errors often occur when the n th element of an n -dimensional array is processed,

when the i th repetition of a loop with i passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

2.2.2 Unit Test Procedures

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure 2.5. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used). Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

2.5 Integration Testing

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together" interfacing. Data can be lost across an interface; one module can have an inadvertent, adverse affect on another; subfunctions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested

components and build a program structure that has been dictated by design.

There is often a tendency to attempt nonincremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the sections that follow, a number of different incremental integration strategies are discussed.

2.5.1 Top-down Integration

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.

Referring to Figure 2.6, depth-first integration would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the lefthand path, components M_1 , M_2 , M_5 would be integrated first. Next, M_8 or (if necessary for proper functioning of M_2) M_6 would be integrated. Then, the central and righthand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M_2 , M_3 , and M_4 (a replacement for stub S_4) would be integrated first. The next control level, M_5 , M_6 , and so on, follows.

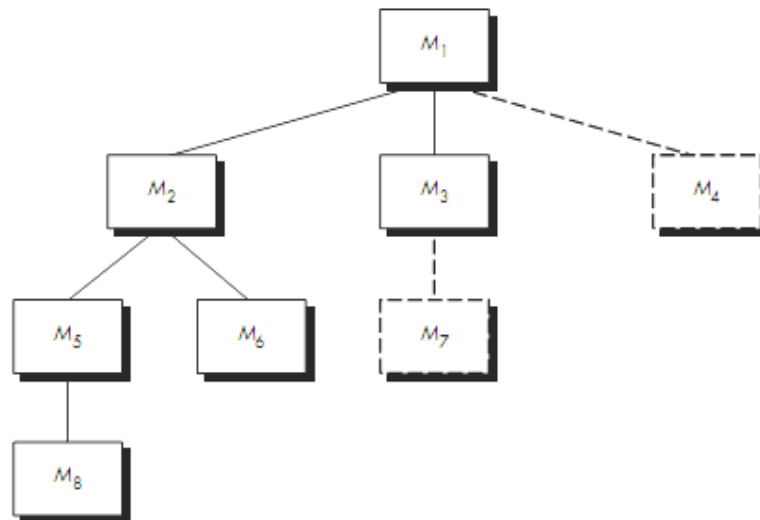


FIGURE 2.6 Top-down integration

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built. The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure in which a complex series of interactive inputs is requested, acquired, and validated via an incoming path. The incoming path may be integrated in a top-down manner. All input processing (for subsequent transaction dispatching) may be demonstrated before other elements of the structure have been integrated. Early demonstration of

functional capability is a confidence builder for both the developer and the customer.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. The tester is left with three choices: (1) delay many tests until stubs are replaced with actual modules, (2) develop stubs that perform limited functions that simulate the actual module, or (3) integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) causes us to lose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex. The third approach, called bottom-up testing, is discussed in the next section.

2.5.2 Bottom-up Integration

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.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in Figure 2.7. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a . Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a . Similarly, driver D_3 for cluster 3 is removed prior

to integration with module M_b . Both M_a and M_b will ultimately be integrated with component M_c , and so forth.

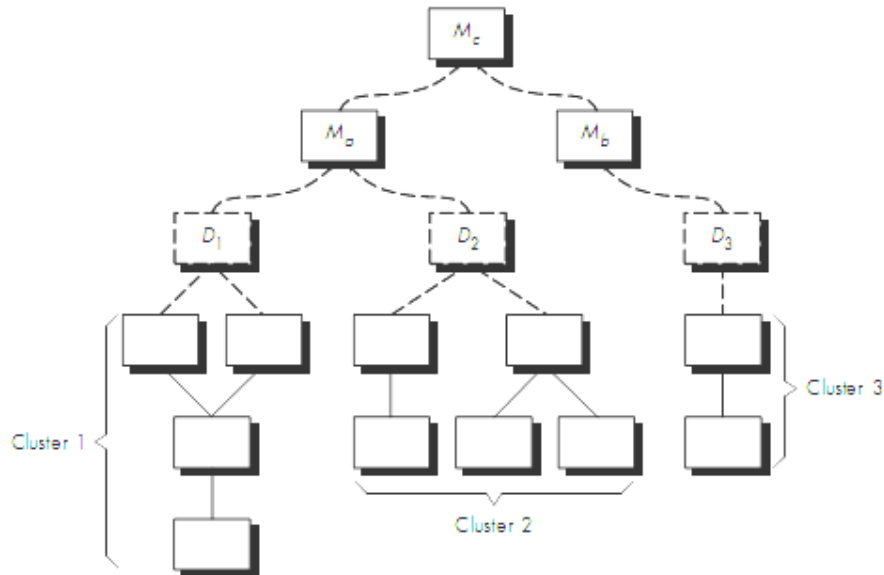


FIGURE 2.7 Bottom-up integration

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

2.5.3 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 functions that previously worked flawlessly. In the context of an integration test strategy, regression testing is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

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 comparison.

The regression test suite (the subset of tests to be executed) 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.

2.5.4 Smoke Testing

Smoke testing is an integration testing approach that is commonly used when “shrink-wrapped” software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a “build.” A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
3. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell [MCO96] describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the

build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of benefits when it is applied on complex, timecritical software engineering projects:

Integration risk is minimized: Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.

The quality of the end-product is improved: Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design defects. If these defects are corrected early, better product quality will result.

Error diagnosis and correction are simplified: Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments" that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.

Progress is easier to assess: With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

2.6 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 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 Specificationa document that describes all user-visible attributes of the software. The specification contains a section called Validation Criteria. Information contained in that section forms the basis for a validation testing approach.

2.6.1 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 procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, 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 two possible conditions exist: (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 delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

2.6.2 Configuration Review

An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometimes called an audit.

2.6.3 Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find.

The alpha test is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The beta test is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally

not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

2.7 System Testing

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem.

Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger-pointing does occur, and (4) participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests that are worthwhile for software-based systems.

2.7.1 Recovery Testing

Many computer based systems must recover from faults and resume processing within a prespecified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery

is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

2.7.2 Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer "The system's security must, of course, be tested for invulnerability from frontal attack but must also be tested for invulnerability from flank or rear attack." During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

2.7.3 Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called sensitivity testing. In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

2.7.4 Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure

Summary

The importance of software testing and its impact on software is explained in this unit. Software testing is a fundamental component of software development life cycle and represents a review of specification, design and coding. The objective of testing is to have the highest likelihood of finding most of the errors within a minimum amount of time and minimal effort. A large number of test case design methods have been developed that offer a systematic approach to testing to the developer.

Knowing the specified functions that the product has been designed to perform, tests can be performed that show that each function is fully operational. A strategy for software testing may be to move upwards along the spiral. Unit testing happens at the vortex of the spiral and concentrates on each unit of the software as implemented by the source code. Testing happens upwards along the spiral to integration testing, where the focus is on design and production of the software architecture. Finally, we perform system testing, where software and other system elements are tested together.

Debugging is not testing, but always happens as a response of testing. The debugging process will have one of two outcomes:

- 1) The cause will be found, then corrected or removed, or
- 2) The cause will not be found. Regardless of the approach that is used, debugging has one main aim: to determine and correct errors. In general, three kinds of debugging approaches have been put forward: Brute force, Backtracking and Cause elimination.

Software testing accounts for the largest percentage of technical effort in the software 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.

Questions

1. Why is a highly coupled module difficult to unit test?
2. How can project scheduling affect integration testing?
3. List some problems that might be associated with the creation of an independent test group.
4. Specify, design, and implement a software tool that will compute the cyclomatic complexity for the programming language.
5. Test a user manual for an application that you use frequently.

UNIT – V

1. Product Metrics

Structure

1. PRODUCT METRICS

1.1 Introduction

1.2 Software Quality

1.3 A Framework for Technical Software Metrics

1.4 Metrics for the Analysis Model

1.5 Metrics for the Design model

1.6 Metrics for Source Code

1.7 Metrics for Testing

1.8 Metrics for Maintenance

1.1 Introduction

A key element of any engineering process is measurement. We use measures to better understand the attributes of the models that we create and to assess the quality of the engineered products or systems that we build. But unlike other engineering disciplines, software engineering is not grounded in the basic quantitative laws of physics. Absolute measures, such as voltage, mass, velocity, or temperature, is uncommon in the software world. Instead, we attempt to derive a set of indirect measures that lead to metrics that provide an indication of the quality of some representation of software. Because software measures and metrics are not absolute, they are open to debate.

Although technical metrics for computer software are not absolute, they provide us with a systematic way to assess quality based on a set of clearly defined rules. They also provide the software engineer with on-the-spot, rather than after-the-fact insight. This enables the engineer to discover and correct potential problems before they become catastrophic defects.

1.2 Software Quality

Quality is defined as conformance to the stated and implied needs of customer. Quality also refers to the measurable characteristics of a software product and these can be compared based on a given set of standards. In the same way, software quality can be defined as conformance to explicitly state and implicitly stated functional requirements. Here, the explicitly stated functional requirement can be derived from the

requirements stated by the customer which are generally documented in some form. Implicit requirements are requirements which are not stated explicitly but are intended. Implicit functional requirements are standards which a software development company adheres to during the development process. Implicit functional requirements also include the requirements of good maintainability.

Quality software is reasonably bug-free, delivered on time and within budget, meets requirements and is maintainable. However, as discussed above, quality is a subjective term. It will depend on who the 'customer' is and their overall influence in the scheme of things. Each type of 'customer' will have their own slant on 'quality'. The end-user might define quality as something which is user-friendly and bug-free.

Good quality software satisfies both explicit and implicit requirements. Software quality is a complex mix of characteristics and varies from application to application and the customer who requests for it.

1.2.1 Attributes of Quality

The following are some of the attributes of quality:

Auditability: The ability of software being tested against conformance to standard.

Compatibility : The ability of two or more systems or components to perform their required functions while sharing the same hardware or software environment.

Completeness: The degree to which all of the software's required functions and design constraints are present and fully developed in the requirements specification, design document and code.

Consistency: The degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component.

Correctness: The degree to which a system or component is free from faults in its specification, design, and implementation. The degree to which software, documentation, or other items meet specified requirements.

Feasibility: The degree to which the requirements, design, or plans for a system or component can be implemented under existing constraints.

Modularity : The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

Predictability: The degree to which the functionality and performance of the software are determinable for a specified set of inputs.

Robustness: The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

Structuredness: The degree to which the SDD (System Design Document) and code possess a definite pattern in their interdependent parts. This implies that the design has proceeded in an orderly and systematic manner (e.g., top-down, bottom-up). The modules are cohesive and the software has minimized coupling between modules.

Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

Traceability: The degree to which a relationship can be established between two or more products of the development process. The degree to which each element in a software development product establishes its reason for existing (e.g., the degree to which each element in a bubble chart references the requirement that it satisfies). For example, the system's functionality must be traceable to user requirements.

Understandability: The degree to which the meaning of the SRS, SDD, and code are clear and understandable to the reader.

Verifiability : The degree to which the SRS, SDD, and code have been written to facilitate verification and testing.

1.2.2 Causes of error in Software

- Misinterpretation of customers' requirements/communication
- Incomplete/erroneous system specification
- Error in logic
- Not following programming/software standards
- Incomplete testing
- Inaccurate documentation/no documentation
- Deviation from specification
- Error in data modeling and representation.

1.2.3 Measurement of Software Quality (Quality metrics)

Software quality is a set of characteristics that can be measured in all phases of software development.

Defect metrics

- Number of design changes required
- Number of errors in the code

- Number of bugs during different stages of testing
- Reliability metrics
- It measures the mean time to failure (MTTF), that may be defined as probability of failure during a particular interval of time. This will be discussed in software reliability.

Maintainability metrics

- Complexity metrics are used to determine the maintainability of software. The complexity of software can be measured from its control flow.

Consider the graph of Figure 1.1. Each node represents one program segment and edges represent the control flow. The complexity of the software module represented by the graph can be given by simple formulae of graph theory as follows:

$$V(G) = e - n + 2 \text{ where}$$

$V(G)$: is called Cyclomatic complexity of the program

e = number of edges

n = number of nodes

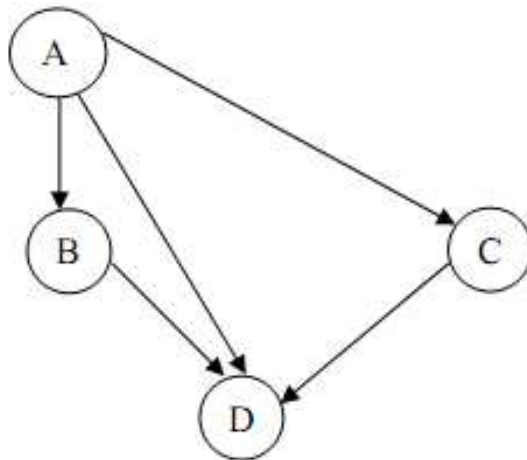


FIGURE 1.1 A software module

Applying the above equation the complexity $V(G)$ of the graph is found to be 1. The cyclomatic complexity has been related to programming effort, maintenance effort and debugging effort. Although cyclomatic complexity measures program complexity, it fails to measure the complexity of a program without multiple conditions.

The information flow within a program can provide a measure for program complexity.

Important parameters for measurement of Software Quality

- To the extent it satisfies user requirements; they form the foundation to measure software quality.
- Use of specific standards for building the software product. Standards could be organisation's own standards or standards referred in a contractual agreement.
- Implicit requirements which are not stated by the user but are essential for quality software.

Software Quality Assurance

The aim of the Software Quality Assurance process is to develop high-quality software product.

The purpose of Software Quality Assurance is to provide management with appropriate visibility into the process of the software project and of the products being built. Software Quality Assurance involves reviewing and auditing the software products throughout the development lifecycle to verify that they conform to explicit requirements and implicit requirements such as applicable procedures and standards.

Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits.

Software Quality Assurance (SQA) is a planned, coordinated and systematic actions necessary to provide adequate confidence that a software product conforms to established technical requirements.

Software Quality Assurance is a set of activities designed to evaluate the process by which software is developed and/or maintained.

The process of Software Quality Assurance

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

Software engineers, project managers, customers and Software Quality Assurance groups are involved in software quality assurance activity. The role of various groups in software quality assurance are as follows:

- **Software engineers:** They ensure that appropriate methods are applied to develop the software, perform testing of the software product and participate in formal technical reviews.

- **SQA group:** They assist the software engineer in developing high quality product. They plan quality assurance activities and report the results of review.

1.3 A Framework for Technical Software Metrics

A measurement assigns numbers or symbols to attributes of entities in the real world. To accomplish this, a measurement model encompassing a consistent set of rules is required. Although the theory of measurement (e.g., [KYB84]) and its application to computer software are topics that are beyond the scope of this book, it is worthwhile to establish a fundamental framework and a set of basic principles for the measurement of technical metrics for software.

1.3.1 The Challenge of Technical Metrics

Over the past three decades, many researchers have attempted to develop a single metric that provides a comprehensive measure of software complexity. Fenton characterizes this research as a search for “the impossible holy grail.” By analogy, consider a metric for evaluating an attractive car. Some observers might emphasize body design, others might consider mechanical characteristics, still others might tout cost, or performance, or fuel economy, or the ability to recycle when the car is junked. Since any one of these characteristics may be at odds with others, it is difficult to derive a single value for “attractiveness.” The same problem occurs with computer software.

1.3.2 Measurement Principles

Before we introduce a series of technical metrics that (1) assist in the evaluation of the analysis and design models, (2) provide an indication of the complexity of procedural designs and source code, and (3) facilitate the design of more effective testing, it is important to understand basic measurement principles. A measurement process that can be characterized by five activities:

Formulation: The derivation of software measures and metrics that are appropriate for the representation of the software that is being considered.

Collection: The mechanism used to accumulate data required to derive the formulated metrics.

Analysis: The computation of metrics and the application of mathematical tools.

Interpretation: The evaluation of metrics results in an effort to gain insight into the quality of the representation.

Feedback: Recommendations derived from the interpretation of technical metrics transmitted to the software team.

The principles that can be associated with the formulation of technical metrics are

- The objectives of measurement should be established before data collection begins.
- Each technical metric should be defined in an unambiguous manner.
- Metrics should be derived based on a theory that is valid for the domain of application (e.g., metrics for design should draw upon basic design concepts and principles and attempt to provide an indication of the presence of an attribute that is deemed desirable).
- Metrics should be tailored to best accommodate specific products and processes.

Although formulation is a critical starting point, collection and analysis are the activities that drive the measurement process. Roche suggests the following principles for these activities:

- Whenever possible, data collection and analysis should be automated.
- Valid statistical techniques should be applied to establish relationships

between internal product attributes and external quality characteristics (e.g., is the level of architectural complexity correlated with the number of defects reported in production use?).

- Interpretative guidelines and recommendations should be established for each metric.

In addition to these principles, the success of a metrics activity is tied to management support. Funding, training, and promotion must all be considered if a technical measurement program is to be established and sustained.

1.3.3 The Attributes of Effective Software Metrics

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

The derived metric and the measures that lead to it should be

Simple and computable: It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time.

Empirically and intuitively persuasive: The metric should satisfy the engineer's intuitive notions about the product attribute under consideration (e.g., a metric that measures module cohesion should increase in value as the level of cohesion increases).

Consistent and objective: The metric should always yield results that are unambiguous. An independent third party should be able to derive the same metric value using the same information about the software.

Consistent in its use of units and dimensions: The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units. For example, multiplying people on the project teams by programming language variables in the program results in a suspicious mix of units that are not intuitively persuasive.

Programming language independent: Metrics should be based on the analysis model, the design model, or the structure of the program itself. They should not be dependent on the vagaries of programming language syntax or semantics.

An effective mechanism for high-quality feedback: That is, the metric should provide a software engineer with information that can lead to a higher quality end product.

1.4 Metrics for the Analysis Model

Technical work in software engineering begins with the creation of the analysis model. It is at this stage that requirements are derived and that a foundation for design is established. Therefore, technical metrics that provide insight into the quality of the analysis model are desirable.

1.4.1 Function-Based Metrics

The function point metric can be used effectively as a means for predicting the size of a system that will be derived from the analysis model. The data flow diagram is evaluated to determine the key measures required for computation of the function point metric:

- number of user inputs
- number of user outputs
- number of user inquiries
- number of files
- number of external interfaces

1.4.2 The Bang Metric

Like the function point metric, the bang metric can be used to develop an indication of the size of the software to be implemented as a consequence of the analysis model. Primitives are determined by evaluating the analysis model and developing counts for the following forms:

Functional primitives (FuP): The number of transformations (bubbles) that appear at the lowest level of a data flow diagram.

Data elements (DE): The number of attributes of a data object, data elements are not composite data and appear within the data dictionary.

Objects (OB): The number of data objects.

Relationships (RE): The number of connections between data objects.

States (ST): The number of user observable states in the state transition diagram.

Transitions (TR): The number of state transitions in the state transition diagram.

In addition to these six primitives, additional counts are determined for

Modified manual function primitives (FuPM): Functions that lie outside the system boundary but must be modified to accommodate the new system.

Input data elements (DEI): Those data elements that are input to the system.

Output data elements (DEO): Those data elements that are output from the system.

Retained data elements (DER): Those data elements that are retained (stored) by the system.

Data tokens (TC_i): The data tokens (data items that are not subdivided within a functional primitive) that exist at the boundary of the *i*th functional primitive (evaluated for each primitive).

Relationship connections (RE_i): The relationships that connect the *i*th object in the data model to other objects.

1.5 Metrics for the Design model

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative.

In the sections that follow, we examine some of the more common design metrics for computer software. Each can provide the designer with

improved insight and all can help the design to evolve to a higher level of quality.

1.5.1 Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

1.5.2 Component-Level Design Metrics

Component-level design metrics focus on internal characteristics of a software component and include measures of the "three Cs" module cohesion, coupling, and complexity. These measures can help a software engineer to judge the quality of a component-level design.

The metrics presented in this section are glass box in the sense that they require knowledge of the inner working of the module under consideration. Component-level design metrics may be applied once a procedural design has been developed. Alternatively, they may be delayed until source code is available.

1.6 Metrics for Source Code

Halstead's theory of software science is one of "the best known and most thoroughly studied composite measures of (software) complexity". Software science proposed the first analytical "laws" for computer software.

Software science assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. These follow:

n_1 = the number of distinct operators that appear in a program.

n_2 = the number of distinct operands that appear in a program.

N_1 = the total number of operator occurrences.

N_2 = the total number of operand occurrences.

Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language), and other features such as development effort, development time, and even the projected number of faults in the software. Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2 \quad (19-10)$$

and program volume may be defined

$$V = N \log_2 (n_1 + n_2) \quad (19-11)$$

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program. Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, L must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

$$L = 2/n_1 \times n_2/N_2$$

1.7 Metrics for Testing

Although much has been written on software metrics for testing, the majority of metrics proposed focus on the process of testing, not the technical characteristics of the tests themselves. In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases. Function-based metrics can be used as a predictor for overall testing effort. Various project-level characteristics (e.g., testing effort and time, errors uncovered, number of test cases produced) for past projects can be collected and correlated with the number of FP produced by a project team. The team can then project “expected values” of these characteristics for the current project.

The number of functional primitives (FuP), data elements (DE), objects (OB), relationships (RE), states (ST), and transitions (TR) can be used to project the number and types of black-box and white-box tests for the software. For example, the number of tests associated with the human/computer interface can be estimated by (1) examining the number of transitions (TR) contained in the state transition representation of the HCI and evaluating the tests required to exercise each transition; (2) examining the number of data objects (OB) that move across the interface, and (3) the number of data elements that are input or output.

Architectural design metrics provide information on the ease or difficulty associated with integration testing and the need for specialized testing software (e.g., stubs and drivers). Cyclomatic complexity (a component-level design metric) lies at the core of basis path testing. In addition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing. Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, the tester should expend above average effort to uncover errors in such modules before they are integrated in a system. Testing effort can also be estimated using metrics derived from Halstead measures (Section 19.5). Using the definitions for program volume, V , and program level, PL , software science effort, e , can be computed as

$$PL = 1/[(n_1/2) \cdot (N_2/n_2)]$$

$$e = V/PL$$

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship: percentage of testing effort (k) = $e(k) / \sum e(i)$ where $e(k)$ is computed for module k using Equations and the summation in the denominator of Equation is the sum of software science effort across all modules of the system.

As tests are conducted, three different measures provide an indication of testing completeness. A measure of the breadth of testing provides an indication of how many requirements (of the total number of requirements) have been tested. This provides an indication of the completeness of the test plan. Depth of testing is a measure of the percentage of independent basis paths covered by testing versus the total number of basis paths in the program. A reasonably accurate estimate of the number of basis paths can be computed by adding the cyclomatic complexity of all program modules. Finally, as tests are conducted and error data are collected, fault profiles may be used to rank and categorize errors uncovered. Priority indicates the severity of the problem. Fault categories provide a description of an error so that statistical error analysis can be conducted.

1.8 Metrics for Maintenance

All of the software metrics introduced in this chapter can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed.

IEEE Std. 982.1-1988 suggests a software maturity index (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

M_T = the number of modules in the current release

F_c = the number of modules in the current release that have been changed

F_a = the number of modules in the current release that have been added

F_d = the number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$SMI = [M_T - (F_a + F_c + F_d)]/M_T$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI and empirical models for maintenance effort can be developed.

Summary

Knowing the specified functions that the product has been designed to perform, tests can be performed that show that each function is fully operational. A strategy for software testing may be to move upwards along the spiral. Unit testing happens at the vortex of the spiral and concentrates on each unit of the software as implemented by the source code. Testing happens upwards along the spiral to integration testing, where the focus is on design and production of the software architecture. Finally, we perform system testing, where software and other system elements are tested together.

Software metrics provide a quantitative way to assess the quality of internal product attributes, thereby enabling the software engineer to assess quality before the product is built. Metrics provide the insight necessary to create effective analysis and design models, solid code, and thorough tests.

GLOSSARY

Abstract Data Types (ADT): A type whose internal form is hidden behind a set of access functions. Objects of the type are created and inspected only by calls to the access functions. This allows the implementation of the type to be changed without requiring any changes outside the module in which it is defined. Abstract data types are central to object-oriented programming where every class is an ADT. A classic example of an ADT is a stack data type for which functions might be provided to create an empty stack, to push values onto a stack and to pop values from a stack.

Abstract System: An abstract system is the system defined by a functional design. It is not a physical system to be found in the real world, but a conceptual system behaving as specified in the functional design. It is our understandings of the abstract systems that enables us to reason about the specified behaviour and verify that it will satisfy the functional requirements.

Abstraction: Generalization, ignoring or hiding details. Examples are abstract data types (the representation details are hidden), abstract syntax (the details of the concrete syntax are ignored), abstract interpretation (details are ignored to analyse specific properties).

Algorithm Animation: Algorithm animation is the process of abstracting the data, operations and semantics of computer programs and then creating animated graphical views of those abstractions.

American National Standards Institute (ANSI): The United States government body responsible for approving US standards in many areas, including computers and communications. ANSI is a member of ISO. ANSI sells ANSI and ISO (international) standards.

Application family: An application family is a generic representation of application systems. One purpose is to allow several abstract systems, possibly defined using different design languages, to be composed in one application. A second purpose is to factor out the support systems which are generic and evolve independently from applications. The concept support heterogeneous applications, not easily covered by a single system description expressed in one of the design languages.

Application Generator: An application generator takes as input a specification of the required product. This specification can be a 4GL program. The product of the generator is usually only modified by rerunning the generator with a changed specification. Building an application generator for some problem domains is difficult and requires much foresight.

Application System: The application part of a system instance implementation. An application system defines the application (the behaviour) a customer wants to buy in terms of implementation code. It is normally a partial implementation lacking the necessary support to execute. An application system is used to produce the concrete systems that actually execute and is expressed using some high-level programming language.

Applications Programmer Interface (API): The interface (calling conventions) by which an application program accesses operating system and other services. An API is defined at source code level and provides a level of abstraction between the application and the kernel (or other privileged utilities) to ensure the portability of the code. An API can also provide an interface

between a high level language and lower level utilities and services which were written without consideration for the calling conventions supported by compiled languages. In this case, the API's main task may be the translation of parameter lists from one format to another and the interpretation of call-by-value and call-by-reference arguments in one or both directions.

Architectural Design: Large systems are divided into smaller subsystems and modules which import from each other. The subsystems and modules and their use relationship is called the architectural design.

Attachment: An encapsulated data object inside a document.[sun]

Automated Reverse Architectural Design: The architectural design of a system and its components can be recovered automatically using certain tools. Some approaches are able to subsume modules into an automatically derived subsystem structure.

CASE Based Reasoning: A technique for problem solving which looks for previous examples which are similar to the current problem. This is useful where heuristic knowledge is not available. Some key research areas are efficient indexing, how to define "similarity" between cases and how to use temporal information.

Class: A class defines a software object's interface and implementation. It specifies the object's internal representation and defines the operations that the object can be instructed to perform.

Client: A computer system or process that requests a service of another computer system or process (a server). For example, a workstation requesting the contents of a file from a file server is a client of the file server.

Client-Server: A common form of distributed system in which software is split between server tasks and client tasks. A client sends requests to a server, according to some protocol, asking for information or action, and the server responds. There may be either one centralized server or several distributed ones. This model allows clients and servers to be placed independently on nodes in a network, possibly on different hardware and operating systems appropriate to their function, e.g. fast server/cheap client.

Client-Server, Three-Tier: Application partitioning, or three-tier architectures are expected to be the next generation of client-server systems. A three-tier system adds a third component (the application server) in between the current client and server. The application server maintains some data and behaviour which reflects business rules. With a two-tier system if business rules change, then all client workstations have to be upgraded. In contrast a three-tier's application server provides a central location and transparent updating of the rules with respect to the clients.

Cohesion: A measure of the level of functional integration within a module. High cohesion implies well defined modules serving one dedicated purpose, low cohesion implies ambiguity. See also coupling.

Common Object Request Broker Architecture (CORBA): An Object Management Group specification which provides the standard interface definition between OMG-compliant objects.

Complexity: A code measure, which is a combination of code, data, data flow, structure and control flow metrics.

Component Integration Laboratories (CIL): An effort to create a common framework for interoperability between application programs on desktop platforms, formed by Apple Computer Inc., IBM, Novell, Oracle, Taligent, WordPerfect and Xerox.

Component Software (Component Object Model, COM): Compound documents and component software define object-based models that facilitate interactions between independent programs. These approaches aim to simplify the design and implementation of applications, and simplify human-computer interaction. Component software addresses the general problem of designing systems from application elements that were constructed independently by different vendors using different languages, tools, and computing platforms. The goal is to have end-users and developers enjoying the same level of plug-and-play application interoperability that are available to hardware manufacturers. Compound documents are one example of component interoperability.

Compound Document (Compound Object Model, COM): Compound documents and component software define object-based models that facilitate interactions between independent programs. These approaches aim to simplify the design and implementation of applications, and simplify human-computer interaction. A compound document is a container for sharing heterogeneous data, which includes mechanisms which manage containment, association with an application, presentation of data/applications, user interaction with data/applications, provision of interfaces for data exchange, and more notably linking and embedding. Data can be incorporated into a document by a pointer (link) to the data contained elsewhere in the document, or in another document. Linking reduces storage requirements, and facilitates automatic transparent updates. Embedding is where the data is physically located within a compound document.

Computer Aided Software Engineering (CASE): A technique for using computers to help with one or more phases of the software life-cycle, including the systematic analysis, design, implementation and maintenance of software. Adopting the CASE approach to building and maintaining systems involves software tools and training for the developers who will use them.

Computer Supported Cooperative/Collaborative Work (CSCW): Software tools and technology to support groups of people working together on a project, often at different sites.

Conceptual Abstraction: Semi-formal, human-oriented and domain-specific abstractions play a critical role both in reverse and forward engineering, and therefore also in reengineering. Such conceptual abstractions are fundamental to the reengineering process whether it is a totally manual or partially automated process.

Concrete System: A concrete system implements the behaviour of one or more abstract systems. A typical concrete system will consist of hardware and executable code, and is what users actually use. Each concrete system will be distinct and will operate in a particular application context.

Configuration Item: Hardware or software, or an aggregate of both, which is designated by the project configuration manager (or contracting agency) for configuration management.

Configuration Management: A discipline applying technical and administrative controls to:

- Identification and documentation of physical and functional characteristics of configuration items.
- Any changes to characteristics of those configuration items.
- Recording and reporting of change processing and implementation of the system.

Configuration Programming: An approach that advocates the use of a separate configuration language to specify the coarse grain structure of programs. Configuration programming is particularly attractive for concurrent, parallel and distributed systems that have inherently complex program structures.

Coupling: The degree to which components depend on one another. There are two types of coupling, "tight" and "loose". Loose coupling is desirable for good software engineering but tight coupling may be necessary for maximum performance. Coupling is increased when the data exchanged between components becomes larger or more complex.

Database: One or more large structured sets of persistent data, usually associated with software to update and query the data. A simple database might be a single file containing many records, each of which contains the same set of fields where each field is a certain fixed width.

Database Management System: A suite of programs which typically manage large structured sets of persistent data, offering adhoc query facilities to many users. A database management system is a complex set of software programs that controls the organisation, storage and retrieval of data (fields, records and files) in a database. It also controls the security and integrity of the database. The DBMS accepts requests for data from the application program and instructs the operating system to transfer the appropriate data. When a DBMS is used, information systems can be changed much more easily as the organization's information requirements change. New categories of data can be added to the database without disruption to the existing system.

Database Object Oriented: A system offering DBMS facilities in an object-oriented programming environment. Data is stored as objects and can be interpreted only using the methods specified by its class. The relationship between similar objects is preserved (inheritance) as are references between objects. Queries can be faster because joins are often not needed (as in a relational database). This is because an object can be retrieved directly without a search, by following its object id. The same programming language can be used for both data definition and data manipulation. The full power of the database programming language's type system can be used to model data structures and the relationship between the different data items

Database Relational: A database based on the relational model developed by E.F. Codd. A relational database allows the definition of data structures, storage and retrieval operations, and integrity constraints. In such a database, the data and relations between them are organised in tables. A table is a collection of records and each record in a table contains the same fields. Certain fields may be designated as keys, which means that searches for specific values of that field will use indexing to speed them up. Records in different tables may be linked if they have

the same value in one particular field in each table. INGRES, Oracle and Microsoft Access are well-known examples.

Data Centered Program Understanding: Instead of focusing on the control structure of a program (such as call graphs, control-flow graphs and paths) data centered program understanding focuses on data and data-relationships.

Data Flow Analysis: A process to discover the dependencies between different data items manipulated by a program. The order of execution in a data driven language is determined solely by the data dependencies.

Data Flow Diagram (DFD):A graphical notation used to describe how data flows between processes in a system. An important tool of most structured analysis techniques.

Data Name Rationalization (DNR): A special case of data re-engineering. DNR tools enforce uniform naming conventions across all software systems.

Design: The phase of software development following analysis, and concerned with how the problem is to be solved.

Design Pattern: A design pattern systematically names, motivates and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

Design Recovery: A subset of reverse engineering in which domain knowledge, external information and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself. Design recovery recreated design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains. The design recovery process consists of three steps:

(a) Supporting program understanding for maintenance (what are the modules? What are the key data items? What are the software engineering artifacts? What are the other informal design abstractions?);

(b) Supporting population of reuse and recovery libraries (The design abstractions of the former step are generalized and integrated into the reuse library and the recovery knowledge base);

(c) Applying the results of design recovery (The abstract design components stored in the domain model now become the starting point for discovering candidate concrete realizations of themselves in a new system's code).

Distributed Computing Environment (DCE): An architecture consisting of standard programming interfaces, conventions and server functionalities (e.g. naming, distributed file

system, remote procedure call) for distributing applications transparently across networks of heterogeneous computers. DCE is promoted and controlled by the Open Software Foundation.

Domain: A functional area covered by a family of systems.

Domain Analysis: The set of activities aiming at identifying, collecting, organizing, analyzing and representing the relevant information in a domain, based on the study of existing systems and their development history, knowledge captured from domain experts, underlying theory, and emerging technologies within the domain. Domain analysis aims at producing domain models and analyzing commonalities and variants among a family of products.

Domain Architecture / Domain Architectural Model / Family Design: architecture applicable to a family of applications belonging to the domain. Some times called “generic architecture”.

Domain Engineering: An encompassing process which includes domain analysis and the subsequent methods and tools that address the problem of development through the application of domain analysis products (e.g., domain implementation).

Domain Model: Domain models are the result of domain analysis. A domain model is a definition of domain abstractions (objects, relationships, functions, events, etc.) It consists off a concise and classified representation of the commonalities and variability of the problems in the domain and of their solutions. It is a representation of a family. Domain models include domain requirements models (the problem) and domain architecture (the solution).

Encapsulation: The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object's state.

Entity :(1) International Organization for Standardization's open systems interconnection (OSI) terminology for a layer protocol machine. An entity within a layer performs the functions of the layer within a single computer system, accessing the layer entity below and providing services to the layer entity above at local service access points.

(2) In object-oriented programming, an entity is part of the definition of a class (group) of objects. In this instance, an entity might be an attribute of the class (as feathers are an attribute of birds), or it might be a variable or an argument in a routine associated with the class.

(3) In database design, an entity is an object of interest about which data can be collected. In a retail database application, customers, products, and suppliers might be entities. An entry can subsume a number of attributes: Product attributes might be color, size, and price; customer attributes might include name, address, and credit rating.

Family: A set of systems sharing some commonalities (equivalent to product line).

Forward Engineering: The set of engineering activities, using the output of software reengineering, that consume the products and artifacts derived from legacy software and new requirements to produce a new target system.

Framework: In object-oriented systems, a set of classes that embodies an abstract design for solutions to a number of related problems.

Groupware: See Computer-support collaborative work.

Heterogeneous Network: A network composed of systems of more than one architecture. Contrast with homogeneous network.

Homogeneous Network: A network composed of systems of only one architecture. Contrast with heterogeneous network.

Hypermedia: An extension of hypertext to include graphics, sound, video and other kinds of data.

Hypertext: A term coined by Ted Nelson around 1965 for a collection of documents (or nodes) containing cross-references or links which, with the aid of an interactive browser program, allow the reader to move easily from one document to another.

Implementation Design Description: An implementation design description is a complete description of all the information needed to produce a concrete system from one or more functional designs. It describes which components of a generic system family must be used, and which tools must be invoked with which parameters. The implementation design description acts as meta-description referring to other descriptions.

Implementation Model: The implementation model consists of the code files and the used work structure. It includes the application software description as well as the support software description. While the design model is a more abstract view, the implementation model contains the full information necessary to build the system.

Informal Reasoning: An approach to knowledge-based concept assignment. Informal reasoning is based on human oriented concepts, but takes knowledge like natural language comments, or grouping in account too. The needed base of knowledge about the problem for the informal reasoning is called the domain model. It is assumed that the domain model (problem, program and application) knowledge can be usefully represented as patterns of informal and semi-formal information, which are called conceptual abstractions.

Information Base: The main repository of information about the software. It can be created by decomposing any number of views of a system.

Interaction Diagram: A diagram that shows the flow of interaction between objects.

Interface: A boundary across which two systems communicate. An interface might be a hardware connector used to link to other devices, or it might be a convention used to allow communication between two software systems. Often there is some intermediate component between the two systems which connects their interfaces together.

(1) The point at which independent systems or diverse groups interact. The devices, rules, or conventions by which one component of a system communicates with another. Also, the point of communication between a person and a computer.

(2) The part of a program that defines constants, variables, and data structures, rather than procedures.

(3) The equipment that accepts electrical signals from one part of a computer system and renders them into a form that can be used by another part.

(4) Hardware or software that links the computer to a device.

(5) To convert signals from one form to another and pass them between two pieces of equipment.

International Organization for Standardization (ISO): A voluntary, non-treaty organisation founded in 1946, responsible for creating international standards in many areas, including computers and communications. ISO produced the seven layer model for network architecture (Open Systems Interconnection). Its members are the national standards organisations of 89 countries, including the American National Standards Institute.

Internationalization: The process of altering a program so that it is portable across several native languages. This portability may support both different character sets, such as the 8-bit ISO 8859/1 (ISO Latin 1) character set and the 7-bit ASCII character set, and different languages for documentation, help screens, and so on.

Language 3rd Generation (3GL): A language designed to be easier for a human to understand, including things like named variables. A fragment might be `let c = c + 2 * d.` FORTRAN, ALGOL and COBOL are early examples of this sort of language. Most "modern" languages (BASIC, C, C++) are third generation. Most 3GLs support structured programming.

Language 4th Generation (4GL): An application specific language. The term was invented to refer to non-procedural high level languages built around database systems. The first three generations were developed fairly quickly, but it was still frustrating, slow, and error prone to program computers, leading to the first "programming crisis", in which the amount of work that might be assigned to programmers greatly exceeded the amount of programmer time available to do it. Meanwhile, a lot of experience was gathered in certain areas, and it became clear that certain applications could be generalised by adding limited programming languages to them. Thus were born report-generator languages, which were fed a description of the data format and the report to generate and turned that into a COBOL (or other language) program which actually contained the commands to read and process the data and place the results on the page. Some other successful 4th-generation languages are: database query languages, e.g. SQL; Focus, Metafont, PostScript, RPG-II, S, IDL-PV/WAVE, Gauss, Mathematica and data-stream languages such as AVS, APE, Iris Explorer.

Language Interface Definition (IDL): To accomplish interoperability across languages and tools, an object model specifies standards for defining application interfaces in terms of a language independent - an interface definition language. Interface definitions are typically stored in a repository which clients can query at run-time.

Language Markup: Languages for annotation of source code to simply improve the source code's appearance with the means of bold-faced key words, slanted comments, etc. See also reformatting.

Language Module Interconnection (MIL): A module interconnection language is a language that is separate from and complementary to a program implementation language. MILs are concerned with the overall architecture of software systems. They deal with the composition of large systems out of modules, the interfaces between these modules & their specification, and the versioning of the resulting architecture over time. The purpose of MILs is to describe a system so that it can be constructed, unequivocally identified, and identically reproduced. A MIL is both a notation for design, documentation & communication, and a means of enforcing system architecture.

Language Object Oriented: A language for object oriented programming. The basic concept in this approach is that of an object which is a data structure (abstract data type) encapsulated with a set of routines, called methods which operate on the data. Operations on the data can only be performed via these methods, which are common to all objects which are instances of a particular class (see inheritance). Thus the interface to objects is well defined, and allows the code implementing the methods to be changed so long as the interface remains the same. Each class is a separate module and has a position in a class hierarchy. Methods or code in one class can be passed down the hierarchy to a subclass or inherited from a superclass. Procedure calls are described in term of message passing. A message names a method and may optionally include other arguments. When a message is sent to an object, the method is looked up in the object's class to find out how to perform that operation on the given object. If the method is not defined for the object's class, it is looked for in its superclass and so on up the class hierarchy until it is found or there is no higher superclass. Procedure calls always return a result object, which may be an error, as in the case where no superclass defines the requested method.

Language Program Description/Design (PDL): Any of a large class of formal and profoundly useless pseudo-languages in which management forces one to design programs. Too often, management expects PDL descriptions to be maintained in parallel with the code, imposing massive overhead of little or no benefit.

Language Structured: A programming language where the program may be broken down into blocks or procedures which can be written without detailed knowledge of the inner workings of other blocks, thus allowing a top-down design approach.

Language Specification and Description (SDL): A language standardised by the ITU-T well suited to functional design of reactive systems comprising concurrent processes with state-transition behaviour.

Language Structured Query (SQL): A language which provides a user interface to relational database management systems, developed by IBM in the 1970s. SQL is the de facto standard, as well as being an ISO and ANSI standard. It is often embedded in other programming languages. SQL provides provided basic language constructs for defining and manipulating tables of data, language extensions for referential integrity and generalised integrity constraints, facilities for schema manipulation and data administration, and capabilities for data definition and data manipulation. Development is currently underway to enhance SQL into a computationally complete language for the definition and management of persistent, complex objects. This includes: generalization and specialisation hierarchies, multiple inheritance, user defined data types, triggers and assertions, support for knowledge based systems, recursive query expressions, and additional data administration tools. It also includes the specification of

abstract data types (ADTs), object identifiers, methods, inheritance, polymorphism, encapsulation, and all of the other facilities normally associated with object data management.

Language Visual Programming (VPL): Any programming language that allows the user to specify a program in a two-(or more)-dimensional way. Conventional textual languages are not considered two-dimensional since the compiler or interpreter processes them as one-dimensional streams of characters. A VPL allows programming with visual expressions - spatial arrangements of textual and graphical symbols. VPLs may be further classified, according to the type and extent of visual expression used, into icon-based languages, form-based languages and diagram languages. Visual programming environments provide graphical or iconic elements which can be manipulated by the user in an interactive way according to some specific spatial grammar for program construction. A visually transformed language is a non-visual language with a superimposed visual representation. Naturally visual languages have an inherent visual expression for which there is no obvious textual equivalent.

Legacy System: A typical computer legacy system may be 10-25 years old, have been developed using archaic methods, have experienced several personnel changes, one for which current maintenance is very expensive, and one for which integration with current or modern technology or software systems is difficult or impossible. Legacy systems require reengineering to put them in a form where they may better suit modern requirements and may evolve more efficiently.

Maintenance / Maintainability: An important part of the software life-cycle. Maintenance is expensive in manpower and resources, and software engineering aims to reduce its cost. Maintenance activities include:

- Perfective maintenance - Changes which improve the system in some way without changing its functionality;
- Adaptive maintenance - Maintenance which is required because of changes in the environment of a program;
- Corrective maintenance - The correction of previously undiscovered system errors.

Metric: A measure of software quality which indicate the complexity, understandability, testability, description and intricacy of code.

Metric Maintenance: Metrics that try to give a quantifying answer on how good a certain program is to maintain.

Middleware: Software that mediates between an application program and a network. It manages the interaction between disparate applications across the heterogeneous computing platforms. The Object Request Broker (ORB), software that manages communication between objects, is an example of a middleware program. A middleware service is a general purpose service that sits between platforms and applications. It is defined by the APIs and protocols it supports. Middleware is generally not application specific, not platform specific, distributed, and supports standard interfaces and protocols.

Object: A run-time entity that packages both data and the procedures that operates on that data.

Object Linking and Embedding (OLE): A distributed object system and protocol from Microsoft also used on the Acorn Archimedes. OLE allows an editor to “farm out” part of a document to another editor and then re-import it. For example, a desk-top publishing system might send some text to a word processor or a picture to a bitmap editor using OLE.

Object Management Group (OMG): consortium aimed at setting standards in object-oriented programming. The Common Object Request Broker Architecture (CORBA) specifies what it takes to be OMG-compliant.

Object Modelling Technique (OMT): An object-oriented analysis and design method used for domain modelling.

Object Oriented Design/Analysis (OOD/OOA): A design method in which a system is modeled as a collection of cooperating objects and individual objects are treated as instances of a class within a class hierarchy. Four stages can be identified: identify the classes and objects, identify their semantics, identify their relationships and specify class and object interfaces and implementation. Object-Oriented design is one of the stages of object-oriented programming.

Object Oriented Programming (OOP): The basic concept in this approach is that of an object which is a data structure (abstract data type) encapsulated with a set of routines, called methods which operate on the data. Operations on the data can only be performed via these methods, which are common to all objects which are instances of a particular class (see inheritance). Thus the interface to objects is well defined, and allows the code implementing the methods to be changed so long as the interface remains the same. Each class is a separate module and has a position in a class hierarchy. Methods or code in one class can be passed down the hierarchy to a subclass or inherited from a superclass. Procedure calls are described in terms of message passing. A message names a method and may optionally include other arguments. When a message is sent to an object, the method is looked up in the object's class to find out how to perform that operation on the given object. If the method is not defined for the object's class, it is looked for in its superclass and so on up the class hierarchy until it is found or there is no higher superclass. Procedure calls always return a result object, which may be an error, as in the case where no superclass defines the requested method.

Object Request Broker (ORB): ORBs are fundamental to CORBA. In a distributed environment they provide a common platform for client objects to request data and services from server objects, and for server objects to pass their responses back to clients. ORBs hide interoperability details from objects (i.e., programming language and operating system used, local or remote, etc.)

OLE Custom Controls (OCX): An Object Linking and Embedding (OLE) custom control allowing infinite extension of the Microsoft Access control set. OCX is similar in purpose to VBX used in Visual Basic.

OMT Model: A model built with the Object Modelling Technique. An OMT model may have three views (object, dynamic and functional). It consists of a set of diagrams and associated information necessary for characterizing the domain (e.g., data dictionary, modelling rationale, costs, etc.)

Open Doc: compound document architecture from CIL based on CORBA. It aims to enable embedding of features from different application programs into a single working document.

Open Scripting Architecture (OSA): An automation technology that works with OpenDoc and lets parts of a component document be manipulated programmatically and coordinated to work together.

Open Software Foundation (OSF): A foundation created by nine computer vendors, (Apollo, DEC, Hewlett-Packard, IBM, Bull, Nixdorf, Philips, Siemens and Hitachi) to promote open computing. It is planned that common operating systems and interfaces, based on developments of Unix and the X Window System will be forthcoming for a wide range of different hardware architectures. OSF announced the release of the industry's first open operating system - OSF/1 on 23 October 1990.

Plug-and-Play: Hardware or software that, after being installed (plugged in), can immediately be used (played with), as opposed to hardware or software which first requires configuration.

Portable Common Tool Environment (PCTE): A European Computer Manufacturers Association standard framework for software tools developed in the Esprit programme. It is based on an entity-relationship Object Management System and defines the way in which tools access this

Potpourri Module: A potpourri module is a module that provides more than one service to a program. This form of module violates the idea of a module being considered a responsibility assignment. The existence of this form of module increases considerably the effort that a programmer has to expend on a maintenance operation, and increases the likelihood of an error being introduced to a program as a result of maintenance work.

Process Model: A model for a set of partially ordered steps required to reach a goal.

Product A product is broader than a system. It incorporates all components that the producer uses, and all the items delivered to customers. Documentation and confidential source code, for instance, is part of the product but not part of the system.

Product Family: A product family is a collection of all the components used to produce concrete systems and any other items delivered to customers. It is generic in the sense that many distinct product instances can be produced from one generic product.

Product Instance: A product instance is what a customer buys. A product instance consists of a copy of executable implementation code and any other items sold with it, typically documentation.

Program Analysis: Program analysis tools are designed to aid the task of understanding existing source code by providing a large amount of detailed information about the program. Analysis tools help focus on the structure and attributes of the system. The relevant information can be extracted from a program by either analyzing the program text (static analysis), or by observing its behavior (dynamic analysis).

Program Heuristics: A general rule of programming concept which captures the conventions in programming and governs the composition of the program plans into programs.

Program Plan Recognition: The use of program plans to identify similar code fragments. Existing source code is often reused within a system via “cut and paste” text operations. Detection of cloned code fragments must be done using program heuristics since the decision whether two arbitrary programs perform the same function is undecidable.

Program Plans/Concepts: An approach to knowledge-based concept assignment. They are schemes in which certain programming problems are usually solved. They are specified in terms of control and data flow and other structural information. A parsing approach (or sometime only pattern matching) will match plans stored in a plan base with the source code to assign concepts.

Program Slicing: A program slice is fragments of a program in which some statements are omitted that are not necessary to understand a certain property of the program. For example if someone is interested in how the value for a certain returned value of a function is arrived at then only code that has a bearing, direct or indirect, on that value is relevant.

Program Understanding: A related term to reverse engineering. Program understanding implies always that understanding begins with the source code while reverse engineering can start at a binary and executable form of the system or at high level descriptions of the design. The science of program understanding includes the cognitive science of human mental processes in program understanding. Program understanding can be achieved in an ad hoc manner and no external representation has to arise. While reverse engineering is the systematic approach to develop an external representation of the subject system, program understanding is comparable with design recovery because both of them start at source code level.

Program Visualization: Program visualization is defined as a mapping from programs to graphical representations.

Protocol: A set of formal rules describing how to transmit data, especially across a network. Low level protocols define the electrical and physical standards to be observed, bit- and byte-ordering and the transmission and error detection and correction of the bit stream. High level protocols deal with the data formatting, including the syntax of messages, the terminal to computer dialogue, character sets, sequencing of messages etc.

Prototyping: The creation of a model and the simulation of all aspects of a product. CASE tools support different degrees of prototyping. Some offer the end-user the ability to review all aspects of the user interface and the structure of documentation and reports before code is generated.

Quality: The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs. Not to be mistaken for "degree of excellence" or "fitness for use" which meet only part of the definition?

Quality Low: Low quality is defined as delivered software which does not work at all, or repeatedly fails in operation. A project where users report more than 0.5 bugs or defects per function point per calendar year are of low quality.

Rapid Application Development (RAD): A loose term for any software life-cycle designed to give faster development and better results and to take maximum advantage of recent advances in development software. RAD is associated with a wide range of approaches to software

development: from hacking away in a GUI builder with little in the way of analysis and design to complete methodologies expanding on an information engineering framework. Some of the current RAD techniques are: CASE tools, iterative life-cycles, prototyping, workshops, SWAT teams, time box development, and reuse of applications, templates and code.

Recode: Changes to implementation characteristics. Language translation and control-flow restructuring are source code level changes. Other possible changes include conforming to coding standards, improving source code readability, renaming programming items, etc.

Redesign: Changes to design characteristics. Possible changes include restructuring design architecture, altering a system's data model as incorporated in data structures or in a database, improvements to an algorithm, etc.

Reengineering: The examination and modification of a system to reconstitute it in a new form and the subsequent re-implementation of the new form.

Reengineering, Business Process (BPR): The fundamental rethinking and radical redesign of business procedures to achieve dramatic improvements in critical, contemporary measures of performance, such as cost, quality, service and speed. An example BPR tool would be process modeling, to facilitate the experimentation with "what if?" scenarios on business processes.

Reformatting: The functional equivalent transformation of source code which changes only the structure to improve readability. Examples are pretty-printers and tools that replace GOTO loops with equivalent loops.

Remodularisation: Changing a module's structure in light of coupling analysis, in order to redefine the boundaries between modules and function of modules.

Remote Procedure Call (RPC): A protocol which allows a program running on one host to cause code to be executed on another host without the programmer needing to explicitly code for this. RPC is an easy and popular paradigm for implementing the client-server model of distributed computing. An RPC is implemented by sending request message to a remote system (the server) to execute a designated procedure, using arguments supplied, and a result message returned to the caller (the client). There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols.

Representation Problem: Building models to understand software systems is an important part of reverse engineering. Formal and explicit model building is important because it focuses attention on modeling as an aid to understanding and results in artifacts that may be useful to others. The representation used to build models has great influence over the success and value of the result. Choosing the proper representation during reverse engineering is the representation problem.

Requirements Model: The requirements model describes the functionality and behaviour of a system. The requirements model is a functional design description of the system to be built.

Respecify: Changes to requirements characteristics. This type of change can refer to changing only the form of existing requirements. For example, taking informal requirements expressed in English and generating a formal specification expressed in a formal language such as Z. This

type of change can also refer to changing system requirements. Requirements changes include the addition of new requirements or the deletion or alteration of existing requirements.

Restructuring: The engineering process of transforming the system from one representational form to another at the same relative level of abstraction, whilst preserving the subject system's external functional behaviour.

Retargeting: The engineering process of transforming and hosting or porting the existing system in a new configuration. This could be a new hardware platform, new operating system or a new CASE platform.

Reuse; Using code developed for one application program in another application. Traditionally achieved using program libraries. Object-oriented programming offers reusability of code via its techniques of inheritance and genericity. Class libraries with intelligent browsers and application generators are under development to help in this process. Polymorphic functional languages also support reusability while retaining the benefits of strong typing.

Reuse Black Box: A style of reuse based on object composition. Composed objects reveal no internal details to each other and are thus analogous to 'black-boxes'.

Reuse Engineering: The modification of software to make it more reusable, usually rebuilding parts to be put into a library.

Reuse White Box: A style of reuse based on class inheritance. A subclass reuses the interface and implementation of its parent class, but it may have access.

Reverse Engineering: The process of analyzing an existing system to identify its components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. Reverse engineering is usually undertaken in order to redesign the system for better maintainability or to produce a copy of a system without access to the design from which it was originally produced.

Reverse Specification: A kind of reverse engineering where a specification is abstracted from the source code or design description. Specification in this context means an abstract description of what the software does. In forward engineering the specification tells us what the software has to do, but this information is not included in the source code. Only in rare cases can it be recovered from comments in the source code and from the people involved in the original forward engineering process.

Risk: Risk is defined as the possibility of loss or injury. Risk exposure is defined by the relationship $RE = P(UO) * L(UO)$ Where RE is the risk exposure, P(UO) is the probability of an unsatisfactory outcome, and L(UO) is the loss to the parties affected by if the outcome is unsatisfactory. Examples of unsatisfactory outcome include schedule slips, budget overruns, wrong functionality, compromised non-functional requirements, user-interface shortfalls and poor quality.

Risk Analysis (and Prioritization): The assessment of the loss probability and loss magnitude for each identified risk item. Prioritization involves producing a ranked and relative ordering of the risk items identified and analysed.

Risk Identification: The production of a list of project specific risk items that are likely to compromise a project's success. An example risk identification is the generation of checklists of likely risk factors.

Risk Management: Risk management is divided into the following tasks:

- Risk assessment
- Risk identification
- Risk analysis and prioritization
- Risk control
- Risk management planning
- Risk resolution and monitoring [boehm91]

Risk Management Planning: Plans which lay out the activities necessary to bring the risk items under control. Activities include prototyping, simulation, modelling, tuning, etc. All management plans should be integrated to reuse parts of each where possible, and to be factored into the overall schedule.

Risk Resolution (and Monitoring): Production of a situation in which the risk items are eliminated or resolved. Risk monitoring involves tracking the project's progress towards resolving its risk items and taking corrective action where appropriate.

Semantics: The meaning of a string in some language, as opposed to syntax which describes how symbols may be combined independent of their meaning. The semantics of a programming language is a function from programs to answers. A program is a closed term and, in practical languages, an answer is a member of the syntactic category of values. The two main kinds are denotational semantics and operational semantics.

Server: A program which provides some service to other (client) programs. The connection between client and server is normally by means of message passing, often over a network, and uses some protocol to encode the client's requests and the server's responses. The server may run continuously (as a daemon), waiting for requests to arrive or it may be invoked by some higher level daemon which controls a number of specific servers.

Software Engineering: A systematic approach to the analysis, design, implementation and maintenance of software. It often involves the use of CASE tools. There are various models of the software life-cycle and many methodologies for the different phases.

Software Evolution: The accommodation of perfective, corrective and adaptive maintenance, which may involve some reengineering activity.

Software Life-Cycle: The software life-cycle consists of: requirements analysis, design, construction, testing (validation) and maintenance. The development process tends to run iteratively through these phases rather than linearly; several models (spiral, waterfall etc.) have

been proposed to describe this process. Other processes associated with a software product are: quality assurance, marketing, sales and support.

Software Methodology: The study of how to navigate through each phase of the software process model (determining data, control, or uses hierarchies, partitioning functions, and allocating requirements) and how to represent phase products (structure charts, stimulus-response threads, and state transition diagrams).

Software Psychology: Software psychology attempts to discover and describe human limitations in interacting with computers. These limitations can place restrictions on and form requirements for computing systems intended for human interaction.

Structured System Analysis and Design Method (SSADM): A software engineering method and toolset required by some UK government agencies.

Subsystem: An independent group of classes that collaborate to fulfill a set of responsibilities.

Subsystem Composition: The process of constructing composite software components out of building blocks such as variables, procedures, modules and subsystems. [mueller90]

Support Families: This is the collection of families used to compose support systems. The generic support is likely to be layer structured. Therefore several general support families are likely, e.g. operating system, I/O system, communication system, user interface, database management system. It is part of the implementation design to determine which support families to use, their composition and how instances are to be configured.

Support System: The support system contains the support needed to actually execute an application system, e.g., the operating system, the user-interface library. It will normally consist of several layers of support where the lower layers provide services to the higher.

Synchronised Refinement: Synchronised refinement is a systematic approach to detecting design decisions in source code and relating the detected decisions to the functionality of the system.

System Building: System building is the process of transforming descriptions using tools to create some less abstract description. This may involve converting designs to source programs to object code. However, the building process may include other transformations such as the construction of system documentation from document fragments.

System Modelling: System modelling is a technique to express, visualize, analyse and transform the architecture of a system. Here a system may consist of software components, hardware components, or both and the connections between these components. A system model is then a skeletal model of the system. It is intended to assist in developing and maintaining large systems with emphasis on the construction phase.

System Object Model (SOM): SOM is IBM's CORBA-compliant object request broker for a single address space architecture. A similar distributed system object model framework exists to allow objects to communicate across address spaces and networks.

Task Interaction Graph: Task interaction graphs divide a program into maximal sequential regions connected by edges representing task interactions. Task interaction graphs can be used to generate concurrency graph representations, and both facilitate analysis of concurrent programs.

Time-To-Market: The time between project start-up and delivery of the final concrete system. This duration is affected by organisation factors and non-software elements of the system,

Transaction: A unit of interaction with a DBMS or similar system. It must be treated in a coherent and reliable way independent of other transactions.

Transaction Processing (TP): The exchange of transactions in a client-server system to achieve the same ends as would be performed by the equivalent single complex application.

Transaction Processing Monitor (TPM): For mission-critical applications it is vital to manage the programs which operate on the data. TP monitors achieve this by breaking complex applications down into transactions. TPMs were invented for applications which serve thousands of clients. A TP monitor can manage transaction resources on a single server or across multiple servers.

Transformation/Translation Program/Software: Transformation of source code from one language to another or from one version of a language to another version of the same language. For example, converting from COBOL-74 to COBOL-85.

Uniform Resource Locator (URL): A draft standard for specifying an object on the Internet, such as a file or newsgroup. URLs are used extensively on the World-Wide Web. They are used in HTML documents to specify the target of a hyperlink.

User Interface (UI): The aspects of a computer system or program which can be seen (or heard or otherwise perceived) by the human user, and the commands and mechanisms the user uses to control its operation and input data. A graphical user interface emphasizes the use of pictures for output and a pointing device such as a mouse for input and control whereas a command line interface requires the user to type textual commands and input at a keyboard and produces a single stream of text as output.

User Interface Graphical (GUI): The use of pictures rather than just words to represent the input and output of a program. A program with a GUI runs under some windowing system (e.g. The X Window System, Microsoft Windows, Acorn RISC OS, and NEXTSTEP). The program displays certain icons, buttons, dialogue boxes etc. in its windows on the screen and the user controls it mainly by moving a pointer on the screen (typically controlled by a mouse) and selecting certain objects by pressing buttons on the mouse while the pointer is pointing at them.

Version: (1) A version is a concrete instance of an object. There could exist multiple versions of one object.

(2) A concrete configuration with concrete versions of the different objects belonging to this configuration. Also known as release.

View: A view is a software representation or a document about software. Example views are requirements and specification documents, hierarchy charts, flowcharts, petri nets, test data, etc. Each view is classified according to a particular view type:

- Non-procedural - e.g. requirements documents
- Pseudo-procedural- e.g. software architecture documents
- Procedural - e.g. source code, data definition
- Analysis views which may accompany any other view.

Views Code: Representations of the source code which cover the same information as the code (or parts of it) but in a manner that accelerates the comprehension process. Examples are program slices, call graphs, data-flow, definition-use graphs, or control dependencies.

Visual Basic: An event-driven visual programming system for Microsoft Windows, in which fragments of BASIC code are invoked when the user performs certain operations on graphical objects on screen. Widely used for in-house applications development by users and for prototyping.

Visual Programming Environment: Software which allows the use of visual expressions (such as graphics, drawings, animation or icons) in the process of programming. These visual expressions may be used as graphical interfaces for textual programming languages. They may be used to form the syntax of new visual programming languages leading to new paradigms such as programming by demonstration or they may be used in graphical presentations of the behavior or structure of a program.

Visual Modelling: A class of RAD tool which allow for the construction and execution of models during design.

Workflow: A workflow is composed of multiple tasks / steps / activities, of which there are two types:

- (1) Simple, representing indivisible activities, and
- (2) Compound, representing those which can be decomposed into sub-activities. An entire workflow can be regarded as a large compound task.

Workflow Management (WFM): Workflow management is a technology that supports the reengineering and automation of business and information processes. It involves:

- (a) Defining workflows, i.e., those aspects of process that are relevant to control and coordinate the execution of its tasks, and
- (b) Providing for fast (re)design and (re)implementation of the processes as business/information needs change.

Workflow Management Coalition (WFMC): A standards body formed in 1993 by a group of companies, intended to address the lack of standards in WFMSs.

Workflow Management System (WFMS): A workflow management system provides procedural automation of a business process by management of the sequence of work activities and the invocation of appropriate human/IT resources associated with the various activity steps. Workflow products are typically client-server software products in which the work is performed within defined time-scales.

World Wide Web (WWW, W3, and Web): An Internet client-server hypertext distributed information retrieval system which originated from the CERN High-Energy Physics laboratories in Geneva, Switzerland. On the WWW everything (documents, menus, and indices) is represented to the user as a hypertext object in HTML format. Hypertext links refer to other documents by their URLs. These can refer to local or remote resources accessible via FTP, Gopher, Telnet or news, as well as those available via the HTTP protocol used to transfer hypertext documents. The client program (known as a browser), e.g. Mosaic, Netscape, runs on the user's computer and provides two basic navigation operations: to follow a link or to send a query to a server. A variety of client and server software is freely available.

X/Open: An international consortium of vendors whose purpose is to define the X/Open Common Applications Environment to provide applications portability. They also produced the X/open Portability Guide (XPG).

---0000---