Paper – II (Software Engineering)

**Paper - II (Software Engineering)**

**Unit – I**

**1. Introduction To Software Engineering**

**1.1 The Evolving Role Of Software**

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 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 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., Internet) 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, pro-found 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.

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 The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.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?

M.Sc. Semester – I (Computer Science) 1

Paper – II (Software Engineering)

• 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,1 are a manifestation of the concern about software and the manner in which it is developed—a concern that has lead to the adoption of software engineering practice.

**1.2 Software**

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.

**1.2.1 Software Characteristics**

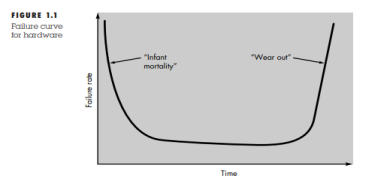
To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, 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

M.Sc. Semester – I (Computer Science) 2

Paper – II (Software Engineering)

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 1.1 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 affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.

Stated simply, the hardware begins to wear out. 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 1.2. 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.

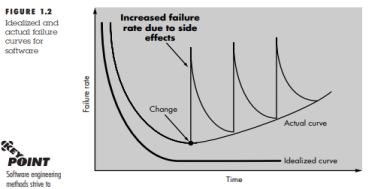
However, the implication is clear—software doesn't wear out. But it does deteriorate! This seeming contradiction can best be explained by considering the “actual curve” shown in Figure 1.2. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 1.2. Before the curve can return to the

M.Sc. Semester – I (Computer Science) 3

Paper – II (Software Engineering)

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. 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.

M.Sc. Semester – I (Computer Science) 4

Paper – II (Software Engineering)

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

**1.2.2 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.

M.Sc. Semester – I (Computer Science) 5

Paper – II (Software Engineering)

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 in

determinate. It is 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:**

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

**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 that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

**Business software:**

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

M.Sc. Semester – I (Computer Science) 6

Paper – II (Software Engineering)

**Engineering and scientific software:**

Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

**Embedded software:**

Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

**Personal computer software:**

The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

**Web-based software:**

The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

**Artificial intelligence software:**

Artificial intelligence (AI) software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge based systems, pattern recognition

M.Sc. Semester – I (Computer Science) 7

Paper – II (Software Engineering)

(image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

**1.4 SOFTWARE MYTHS**

Many causes of a software affliction can be traced to a mythology that arose during the early history of software development. Unlike ancient myths that often provide human lessons well worth heeding, software myths propagated misinformation and confusion. Software myths had a number of attributes that made them insidious; for instance, they appeared to be reasonable statements of fact (sometimes containing elements of truth), they had an intuitive feel, and they were often promulgated by experienced practitioners who "knew the score." Today, most knowledgeable professionals recognize myths for what they are— misleading attitudes that have caused serious problems for managers and technical people alike.

However, old attitudes and habits are difficult to modify, and remnants of software myths are still believed. 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).

M.Sc. Semester – I (Computer Science) 8

Paper – II (Software Engineering)

**Reality:** Software development is not a mechanistic process like manufacturing. In the words of Brooks "adding people to a late software project makes it 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 outsource3 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 1.3 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)

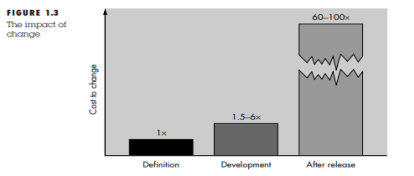
M.Sc. Semester – I (Computer Science) 9

Paper – II (Software Engineering)

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. 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 create 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.

M.Sc. Semester – I (Computer Science) 10

Paper – II (Software Engineering)

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

**2. Process models**

**2.1 Software Engineering: 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:

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. **2.1.1 Process, Methods, and Tools**

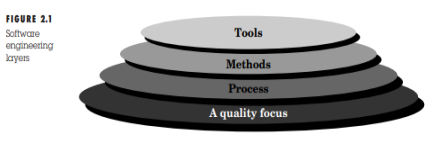
Software engineering is a layered technology. Referring to Figure 2.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and this culture ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely

M.Sc. Semester – I (Computer Science) 11

Paper – II (Software Engineering)

development of computer software. Process defines a framework for a set of key process areas (KPA) that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed. Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support.

Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques. Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established. CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment analogous to CAD/CAE (computer-aided design/engineering) for hardware.

**2.1.2 A Generic View of Software Engineering**

Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Regardless of the entity to be engineered, the following questions must be asked and answered:

• What is the problem to be solved?

• What characteristics of the entity are used to solve the problem?

• How will the entity (and the solution) be realized?

M.Sc. Semester – I (Computer Science) 12

Paper – II (Software Engineering)

• How will the entity be constructed?

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

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

We focus on a single entity—computer software. To engineer software adequately, a software engineering process must be defined. In this section, the generic characteristics of the software process are considered.

Specific process models are addressed. The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity. Each phase addresses one or more of the questions noted previously. The definition phase focuses on what. That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the software are identified. Although the methods applied during the definition phase will vary depending on the software engineering paradigm (or combination of paradigms) that is applied, three major tasks will occur in some form: system or information engineering, software project planning, and requirements analysis.

The development phase focuses on how. That is, during development a software engineer attempts to define how data are to be structured, how function is to be implemented within a software architecture, how procedural details are to be implemented, how interfaces are to be characterized, how the design will be translated into a programming language (or nonprocedural language), and how testing will be performed. The methods applied during the development phase will vary, but three specific technical tasks should always occur: software design, code generation, and software testing. The support phase focuses on change associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements. The support phase reapplies the steps of the definition and development phases but does so in the context of existing software. Four types of change are encountered during the support phase:

M.Sc. Semester – I (Computer Science) 13

Paper – II (Software Engineering)

**Correction:** Even with the best quality assurance activities, it is likely that the customer will uncover defects in the software. Corrective maintenance changes the software to correct defects. **Adaptation:** Over time, the original environment (e.g., CPU, operating system, business rules, external product characteristics) for which the software was developed is likely to change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.

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

**Prevention**: Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to serve the needs of its end users.

In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced. In addition to these support activities, the users of software require continuing support.

In-house technical assistants, telephone-help desks, and application-specific Web sites are often implemented as part of the support phase. Today, a growing population of legacy programs1 is forcing many companies to pursue software reengineering strategies. In a global sense, software reengineering is often considered as part of business process reengineering. The phases and related steps described in our generic view of software engineering are complemented by a number of umbrella activities. Typical activities in this category include:

• Software project tracking and control

• Formal technical reviews

• Software quality assurance

• Software configuration management

• Document preparation and production

• Reusability management

• Measurement

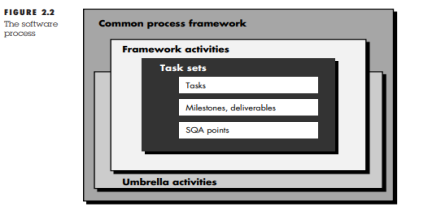
• Risk management

M.Sc. Semester – I (Computer Science) 14

Paper – II (Software Engineering)

**2.2 The Software Process**

A software process can be characterized as shown in Figure 2.2. A common process framework is established by defining a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. A number of task sets—each a collection of software engineering work tasks, project milestones,

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:

M.Sc. Semester – I (Computer Science) 15

Paper – II (Software Engineering)

**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.

M.Sc. Semester – I (Computer Science) 16

Paper – II (Software Engineering)

• **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:3 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

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

**2.3 Software Process Models**

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and

M.Sc. Semester – I (Computer Science) 17

Paper – II (Software Engineering)

tools layers and the generic phases. This strategy is often referred to as a process model or a software engineering paradigm. A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. In an intriguing paper on the nature of the software process, L. B. S. Raccoon uses fractals as the basis for a discussion of the true nature of the software process

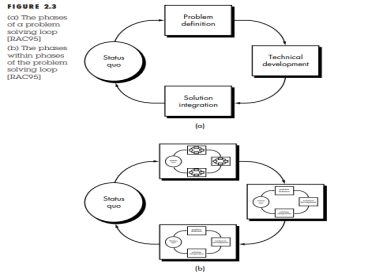
All software development can be characterized as a problem solving loop (Figure 2.3a) in which four distinct stages are encountered: status quo, problem definition, technical development, and solution integration. Status quo “represents the current state of affairs”; problem definition identifies the specific problem to be solved; technical development solves the problem through the application of some technology, and solution integration delivers the results (e.g., documents, programs, data, new business function, new product) to those who requested the solution in the first place.

The generic software engineering phases and stages. This problem solving loop applies to software engineering work at many different levels of resolution. It can be used at the macro level when the entire application is considered, at a mid-level when program components are being engineered, and even at the line of code level.

Therefore, a fractal representation can be used to provide an idealized view of process. In Figure 2.3b, each stage in the problem solving loop contains an identical problem solving loop, which contains still another problem solving loop (this continues to some rational boundary; for software, a line of code). Realistically, it is difficult to compartmentalize activities as neatly as Figure 2.3b implies because cross talk occurs within and across stages. Yet, this simplified view leads to a very important idea: regardless of the process model that is chosen for a software project, all of the stages—status quo, problem definition, technical development, and solution integration—coexist simultaneously at some level of detail.

M.Sc. Semester – I (Computer Science) 18

Paper – II (Software Engineering)



Given the recursive nature of Figure 2.3b, the four stages discussed apply equally to the analysis of a complete application and to the generation of a small segment of code. Raccoon suggests a “Chaos model” that describes “software development as a continuum from the user to the developer to the technology.” As work progresses toward a complete system, the stages are applied recursively to user needs and the developer’s technical specification of the software. In the sections that follow, a variety of different process models for software engineering are discussed. Each represents an attempt to bring order to an inherently chaotic activity. It is important to remember that each of the models has been characterized in a way that (ideally) assists in the control and coordination of a real software project. And yet, at their core, all of the models exhibit characteristics of the Chaos model.

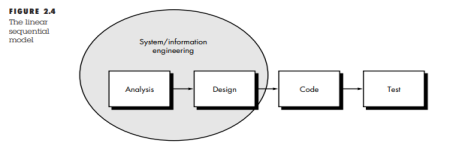
**2.4 The Linear Sequential Model**

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

M.Sc. Semester – I (Computer Science) 19

Paper – II (Software Engineering)

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

Information engineering encompasses requirements gathering at the strategic business level and at the business area level.

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

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

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

**Testing:** Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the

M.Sc. Semester – I (Computer Science) 20

Paper – II (Software Engineering)

functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

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

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

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

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

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

In an interesting analysis of actual projects Bradac, found that the linear nature of the classic life cycle leads to “blocking states” in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking state tends to be more prevalent at the beginning and end of a linear sequential process. Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and support can be placed. The classic life cycle remains a widely used procedural model for software engineering.

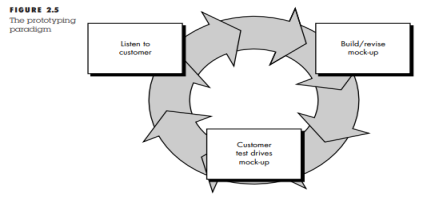
M.Sc. Semester – I (Computer Science) 21

Paper – II (Software Engineering)

While it does have weaknesses, it is significantly better than a haphazard approach to software development.

**2.5 The Prototyping Model**

Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach. The prototyping paradigm (Figure 2.5) begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype.

 The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done. Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly. But what do we do with the prototype when it has served the purpose just described? Brooks provides an answer: In most projects, the first system built is barely usable. It may be too

M.Sc. Semester – I (Computer Science) 22

Paper – II (Software Engineering)

slow, too big, and awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved. When a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time.

The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers. The prototype can serve as "the first system." The one that Brooks recommends we throw away. But this may be an idealized view. It is true that both customers and developers like the prototyping paradigm. Users get a feel for the actual system and developers get to build something immediately. Yet, prototyping can also be problematic for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together “with chewing gum and baling wire,” unaware that in the rush to get it working no one has considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. 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 familiar 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. Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, the customer and developer must both agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part) and the actual software is engineered with an eye toward quality and maintainability

**2.6 The RAD Model**

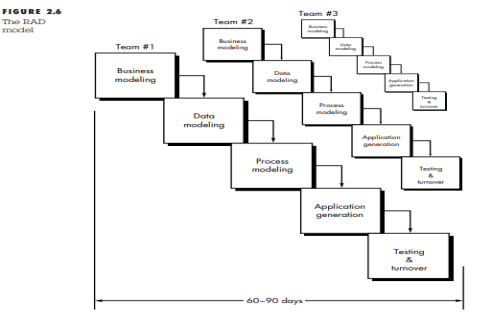
Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle. The RAD model is a “high-speed” adaptation of the linear sequential model in which rapid development is achieved by using

M.Sc. Semester – I (Computer Science) 23

Paper – II (Software Engineering)

component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a “fully functional system” within very short time periods (e.g., 60 to 90 days). Used primarily for information systems applications, the RAD approach encompasses the following phases:

**Business modeling:** The information flow among business functions is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it? . **Data modeling:** The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The char acteristics

(called attributes) of each object are identified and the relationships between these objects defined. The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created

for adding, modifying, deleting, or retrieving a data object.

**Application generation:** RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages the RAD process

M.Sc. Semester – I (Computer Science) 24

Paper – II (Software Engineering)

works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software. **Testing and turnover:** Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised. The RAD process model is illustrated in Figure 2.6. Obviously, the time constraints imposed on a RAD project demand “scalable scope”. If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described previously), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole. Like all process models, the RAD approach has drawbacks:

• For large but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

• RAD requires developers and customers who are committed to the rapid-fire activities necessary to get a system complete in a much abbreviated time frame. If commitment is lacking from either constituency, RAD projects will fail.

• Not all types of applications are appropriate for RAD. If a system cannot be properly modularized, building the components necessary for RAD will be problematic. If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.

• RAD is not appropriate when technical risks are high. This occurs when a new application makes heavy use of new technology or when the new software requires a high degree of interoperability with existing computer programs.

**2.7 Evolutionary Software Process Models**

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

M.Sc. Semester – I (Computer Science) 25

Paper – II (Software Engineering)

straight-line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The prototyping model (Section 2.5) is designed to assist the customer (or developer) in understanding requirements. In general, it is not designed to deliver a production system. The evolutionary nature of software is not considered in either of these classic software engineering paradigms.

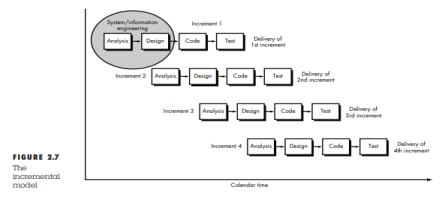
Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software. **2.7.1 The Incremental Model**

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. Referring to Figure 2.7, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable “increment” of the software.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm. When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

M.Sc. Semester – I (Computer Science) 26

Paper – II (Software Engineering)



The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user. Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay.

**2.7.2 The Spiral Model**

The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. A spiral model is divided into a number of framework activities, also called task

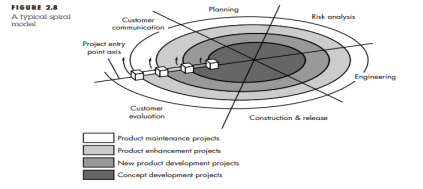
M.Sc. Semester – I (Computer Science) 27

Paper – II (Software Engineering)

regions. Typically, there are between three and six task regions. Figure 2.8 depicts a spiral model that contains six task regions:

• Customer communication—tasks required to establish effective communication between developer and customer.

• Planning—tasks required to define resources, timelines, and other project related information. • Risk analysis—tasks required to assess both technical and management risks. • Engineering—tasks required to build one or more representations of the application. • Construction and release—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

• Customer evaluation—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage. Each of the regions is populated by a set of work tasks, called a task set, that are adapted to the characteristics of the project to be undertaken. For small projects, the number of work tasks and their formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality. In all cases, the umbrella activities (e.g., software configuration management and software quality assurance) are applied. As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the center. The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated 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 customer evaluation. In addition, the

M.Sc. Semester – I (Computer Science) 28

Paper – II (Software Engineering)

project manager adjusts the planned number of iterations required to complete the software. Unlike classical process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.

An alternative view of the spiral model can be considered by examining the project entry point axis, also shown in Figure 2.8. Each cube placed along the axis can be used to represent the starting point for different types of projects. A “concept development project” starts at the core of the spiral and will continue (multiple iterations occur along the spiral path that bounds the central shaded region) until concept development is complete. If the concept is to be developed into an actual product, the process proceeds through the next cube (new product development project entry point) and a “new development project” is initiated. The new product will evolve through a number of iterations around the spiral, following the path that bounds the region that has somewhat lighter shading than the core.

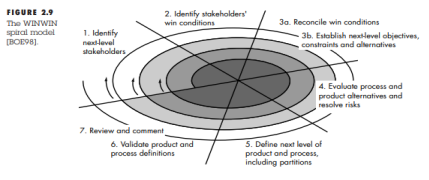
In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant, but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement). The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic. But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur. Finally, the model has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important paradigm can be determined with absolute certainty.

M.Sc. Semester – I (Computer Science) 29

Paper – II (Software Engineering)

**2.7.3 The WINWIN Spiral Model**

The spiral model discussed in Section 2.7.2 suggests a framework activity that addresses customer communication. The objective of this activity is to elicit project requirements from the customer. In an ideal context, the developer simply asks the customer what is required and the customer provides sufficient detail to proceed. Unfortunately, this rarely happens. In reality, the customer and the developer enter into a process of negotiation, where the customer may be asked to balance functionality, performance, and other product or system characteristics against cost and time to market. The best negotiations strive for a “win-win” result. That is, the customer wins by getting the system or product that satisfies the majority of the customer’s needs and the developer wins by working to realistic and achievable budgets and deadlines

Boehm’s WINWIN spiral model defines a set of negotiation activities at the beginning of each pass around the spiral. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem’s key “stakeholders.”

2. Determination of the stakeholders’ “win conditions.”

3. Negotiation of the stakeholders’ win conditions to reconcile them into a set of win-win conditions for all concerned (including the software project team). Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to software and system definition.

The WINWIN spiral model is illustrated in Figure 2.9. In addition to the emphasis placed on early negotiation, the WINWIN spiral model introduces three process milestones, called anchor points, that help establish the completion of one cycle around the spiral and provide

M.Sc. Semester – I (Computer Science) 30

Paper – II (Software Engineering)

decision milestones before the software project proceeds. In essence, the anchor points represent three different views of progress as the project traverses the spiral.

The first anchor point, life cycle objectives (LCO), defines a set of objectives for each major software engineering activity. For example, as part of LCO, a set of objectives establishes the definition of top-level system/product requirements. The second anchor point, life cycle architecture (LCA), establishes objectives that must be met as the system and software architecture is defined. For example, as part of LCA, the software project team must demonstrate that it has evaluated the applicability of off-the-shelf and reusable software components and considered their impact on architectural decisions. Initial operational capability (IOC) is the third anchor point and represents a set of objectives associated with the preparation of the software for installation/distribution, site preparation prior to installation, and assistance required by all parties that will use or support the software.

**2.7.4 The Concurrent Development Model**

The concurrent development model, sometimes called concurrent engineering, has been described in the following manner by Davis and Sitaram: Project managers who track project status in terms of the major phases [of the classic life cycle] have no idea of the status of their projects. These are examples of trying to track extremely complex sets of activities using overly simple models. Note that although . . . [a large] project is in the coding phase, there are personnel on the project involved in activities typically associated with many phases of development simultaneously.

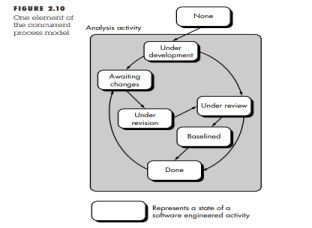
For example, personnel are writing requirements, designing, coding, testing, and integration testing [all at the same time]. Software engineering process models by Humphrey and Kellner have shown the concurrency that exists for activities occurring during any one phase. Kellner's more recent work uses state charts [a notation that represents the states of a process] to represent the concurrent relationship existent among activities associated with a specific event (e.g., a requirements change during late development), but fails to capture the richness of concurrency that exists across all software development and management activities in the project. Most software development process models are driven by time; the later it is, the later in the development process you are. [A concurrent process model] is driven by user needs, management decisions, and review results. The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states. For

M.Sc. Semester – I (Computer Science) 31

Paper – II (Software Engineering)

example, the engineering activity defined for the spiral model is accomplished by invoking the following tasks: prototyping and/or analysis modeling, requirements specification, and design. Figure 2.10 provides a schematic representation of one activity with the concurrent process model. The activity—analysis—may be in any one of the states noted at any given time. Similarly, other activities (e.g., design or customer communication) can be represented in an analogous manner. All activities exist concurrently but reside in different states.

For example, early in a project the customer communication activity (not shown in the figure) has completed its first iteration and exists in the awaiting changes state. The analysis activity (which existed in the none state while initial customer communication was completed) now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the analysis activity moves from the under development state into the awaiting changes state. The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities. For example,



during early stages of design, an inconsistency in the analysis model is uncovered. This generates the event analysis model correction which will trigger the analysis activity from the done state into the awaiting changes state. The concurrent process model is often used as the paradigm for the development of client/server applications. A client/server system is composed of a set of functional components. When applied to client/server, the concurrent process model defines activities in two dimensions: a system dimension and a component dimension. System level

M.Sc. Semester – I (Computer Science) 32

Paper – II (Software Engineering)

issues are addressed using three activities: design, assembly, and use. The component dimension is addressed with two activities:

**design and realization:** Concurrency is achieved in two ways:

(1) System and component activities occur simultaneously and can be modeled using the state oriented approach described previously.

(2) A typical client/server application is implemented with many components, each of which can be designed and realized concurrently.

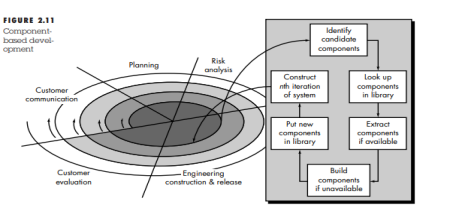
In reality, the concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities to a sequence of events, it defines a network of activities. Each activity on the network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

**2.8 Component-Based Development**

Object-oriented technologies provide the technical framework for a component-based process model for software engineering. The object oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithms used to manipulate the data. If properly designed and implemented, object-oriented classes are reusable across different applications and computer-based system architectures. The component-based development (CBD) model (Figure 2.11) incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model composes applications from prepackaged software components (called classes). The engineering activity begins with the identification of candidate classes. This is accomplished by examining the data to be manipulated by the application and the algorithms that will be applied to accomplish the manipulation.12 Corresponding data and algorithms are packaged into a class.

M.Sc. Semester – I (Computer Science) 33

Paper – II (Software Engineering)



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

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Based on studies of reusability, QSM Associates, Inc., reports component assembly leads to a 70 percent reduction in development cycle time; an 84 percent reduction in project cost, and a productivity index of 26.2, compared to an industry norm of 16.9. Although these results are a function of the robustness of the component library, there is little question that the component-based development model provides significant advantages for software engineers. The unified software development process is representative of a number of component-based development models that have been proposed in the industry. Using the Unified Modeling Language (UML), the unified process defines the components that will be used to build the system and the interfaces that will connect the components. Using a combination of iterative and incremental development, the unified process defines the function of the system by applying a scenario-based approach (from the user

M.Sc. Semester – I (Computer Science) 34

Paper – II (Software Engineering)

point of view). It then couples function with an architectural framework that identifies the form the software will take.

**3. Software requirements**

The requirements for a system are the descriptions of the services provided by the system and its operational constraints. These requirements reflect the needs of customers for a system that helps solve some problem such as controlling a device, placing an order or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE). In this chapter, I concentrate on the requirements themselves and how to describe them. The term requirement is not used in the software industry in a consistent way. In some cases, a requirement is simply a high-level, abstract statement of a service that the system should provide or a constraint on the system. At the other extreme, it is a detailed, formal definition of a system function. Davis (Davis, 1993) explains why these differences exist: If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined.

The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system. Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term user requirements to mean the high-level abstract requirements and system requirements to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide and the constraints under which it must operate. 2. System requirements set out the system’s functions, services and operational constraints in detail.

The system requirements document (sometimes called a functional specification) should be precise. It should define exactly what is to be implemented. It may be part of the contract

M.Sc. Semester – I (Computer Science) 35

Paper – II (Software Engineering)

between the system buyer and the software developers. Different levels of system specification are useful because they communicate information about the system to different types of readers. Figure 6.1 illustrates the distinction between user and system requirements. This example from a library system shows how a user requirement may be expanded into several system requirements. You can see from Figure 6.1 that the user requirement is more abstract, and the system requirements add detail, explaining the services and functions that should be provided by the system to be developed.

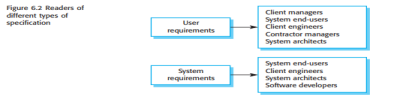
You need to write requirements at different levels of detail because different types of readers use them in different ways. Figure 6.2 shows the types of readers for the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

**3.1 Functional and non-functional requirements**

Software system requirements are often classified as functional requirements, nonfunctional requirements or domain requirements:

1. Functional requirements these are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. 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. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.

M.Sc. Semester – I (Computer Science) 36

Paper – II (Software Engineering)

4. Domain requirements these are requirements that come from the application domain of the system and that reflect characteristics and constraints of that domain. They may be functional or non-functional requirements In reality, the distinction between different types of requirements is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, say, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

**3.1.1 Functional requirements**

The functional requirements for a system describe what the system should do. These requirements 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. When expressed as user requirements, the requirements are usually described in a fairly abstract way. However, functional system requirements describe the system function in detail, its inputs and outputs, exceptions, and so on. Functional requirements for a software system may be expressed in a number of ways. For example, here are examples of functional requirements for a university library system called LIBSYS, used by students and faculty to order books and documents from other libraries.

1. The user shall be able to search either all of the initial set of databases or select a subset from it.

2. The system shall provide appropriate viewers for the user to read documents in the document store.

3. Every order shall be allocated a unique identifier (ORDER\_ID), which the user shall be able to copy to the account’s permanent storage area.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document, and they illustrate that functional requirements may be written at different levels of detail (contrast requirements 1 and 3). The LIBSYS system is a single interface to a range of article databases. It allows users to download copies of published articles in magazines, newspapers and scientific journals. I give a more detailed description of the requirements for the system on which LIBSYS is based in my book with Gerald Kotonya on requirements engineering (Kotonya

M.Sc. Semester – I (Computer Science) 37

Paper – II (Software Engineering)

and Sommerville, 1998). Imprecision in the requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement to simplify its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs. Consider the second example requirement for the library system that refers to ‘appropriate viewers’ provided by the system.

The library system can deliver documents in a range of formats; the intention of this requirement is that viewers for all of these formats should be available. However, the requirement is worded ambiguously; it does not make clear that viewers for each document format should be provided. A developer under schedule pressure might simply provide a text viewer and claim that the requirement had been met. 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 requirements should not have contradictory definitions.

In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness. One reason for this is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that different system stakeholders have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or, sometimes, after development is complete and the system is delivered to the customer.

**3.1.2 Non-functional requirements**

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific functions delivered by the system. They may relate to emergent system properties such as reliability, response time and store occupancy. Alternatively, they may define constraints on the system such as the capabilities of I/O devices and the data representations used in system interfaces. Non-functional requirements are rarely associated with individual system features. Rather, these requirements specify or constrain the emergent properties of the system. Therefore, they may specify system

M.Sc. Semester – I (Computer Science) 38

Paper – II (Software Engineering)

performance, security, availability, and other emergent properties. This means that they are often more critical than individual functional requirements.

**Fig. Types of Non-functional requirements**

System users can usually find ways to work around a system function that doesn’t really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if a real-time control system fails to meet its performance requirements, the control functions will not operate correctly. Non

functional requirements are not just concerned with the software system to be developed. Some non-functional requirements may constrain the process that should be used to develop the system.

Examples of process requirements include a specification of the 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, or because of external factors such as safety regulations or privacy legislation. Figure 6.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of

M.Sc. Semester – I (Computer Science) 39

Paper – II (Software Engineering)

the software (product requirements), the organization developing the software (organizational requirements) or from external sources.

The types of non-functional requirements are:

1. Product requirements - These requirements specify product behavior. Examples include 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; and usability requirements.

2. Organizational requirements - These requirements are derived from policies and procedures in the customer’s and developer’s organization. Examples include process standards that must be used; implementation requirements such as the programming language or design method used; and delivery requirements that specify when the product and its documentation are to be delivered.

3. External requirements - This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include interoperability 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; and ethical requirements. Ethical requirements are requirements placed on a system to ensure that it will be acceptable to its users and the general public.

Figure 6.4 shows examples of product, organizational and external requirements taken from the library system LIBSYS whose user requirements were discussed in Section 6.1.1. The product requirement restricts the freedom of the LIBSYS designers in the implementation of the system user interface. It says nothing about the functionality of LIBSYS and clearly identifies a system constraint rather than a function. This requirement has been included because it simplifies the problem of ensuring the system works with different browsers. The organizational requirement specifies that the system must be developed according to a company standard process defined as XYZCo-SP-STAN-95. The external requirement is derived from the need for the system to conform to privacy legislation. It specifies that library staff should not be allowed access to data, such as the addresses of system users, which they do not need to do their job.

A common problem with non-functional requirements is that they can be difficult to verify. Users or customers often state these requirements as general goals such as ease of use,

M.Sc. Semester – I (Computer Science) 40

Paper – II (Software Engineering)

the ability of the system to recover from failure or rapid user response. These vague goals cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. As an illustration of this problem, consider Figure 6.5.

This shows a system goal relating to the usability of a traffic control system and is typical of how a user might express usability requirements. I have rewritten it to show how the goal can be expressed as a ‘testable’ non-functional requirement. While it is impossible to objectively verify the system goal, you can design system tests to count the errors made by controllers using a system simulator. Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested.

Figure 6.6 shows a number of possible metrics that you can use to specify non functional system properties. You can measure these characteristics when the system is being tested to check whether or not the system has met its non-functional requirements. In practice, however, customers for a system may find it practically impossible to translate their goals into quantitative requirements. For some goals, such as maintainability, there are no metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don’t understand what some number defining the required reliability (say) means in terms of their everyday experience with computer systems.

Furthermore, the cost of objectively verifying quantitative nonfunctional requirements may be very high, and the customers paying for the system may not think these costs are justified. Therefore, requirements documents often include statements of goals mixed with requirements. These goals may be useful to developers because they give indications of customer priorities. However, you should always tell customers that they are open to misinterpretation and cannot be objectively verified. Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, it may be a requirement that the maximum memory used by a system should be no more than 4 Mbytes. Memory constraints are common for embedded systems where space or weight is limited and the number of ROM chips storing the system software must be minimised. Another requirement might be that the system should be written using Ada, a programming

M.Sc. Semester – I (Computer Science) 41

Paper – II (Software Engineering)



language for critical, real-time software development. However, it may not be possible to compile an Ada program with the required functionality into less that 4 Mbytes. There therefore has to be a trade-off between these requirements: an alternative development language or increased memory added to the system. It is helpful if you can differentiate functional and non-functional requirements in the requirements document. In practice, this is difficult to do. If the non-functional requirements are stated separately from the functional requirements, it is sometimes difficult to see the relationships between them. If they are stated with the functional requirements, you may find it difficult to separate functional and nonfunctional considerations and to identify requirements that relate to the system as a whole.

However, you should explicitly highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements. Non-functional requirements such as safety and security requirements are particularly important for critical systems.

**3.1.3 Domain requirements**

Requirements in their own right constrain existing functional requirements or set out how particular computations must be carried out. Because these requirements are specialized,

M.Sc. Semester – I (Computer Science) 42

Paper – II (Software Engineering)

software engineers often find it difficult to understand how they are related to other system requirements. Domain requirements are important because they often reflect fundamentals of the application domain. If these requirements are not satisfied, it may be impossible to make the system work satisfactorily. The LIBSYS system includes a number of domain requirements:

1. There shall be a standard user interface to all databases that shall be based on the Z39.50 standard.

2. Because of copyright restrictions, some documents must be deleted immediately on arrival.

Depending on the user’s requirements, these documents will either be printed locally on the system server for manual forwarding to the user or routed to a network printer. The first requirement is a design constraint. It specifies that the user interface to the database must be implemented according to a specific library standard. The developers therefore have to find out about that standard before starting the interface design. The second requirement has been introduced because of copyright laws that apply to material used in libraries. It specifies that the system must include an automatic delete-on-print facility for some classes of document. This means that users of the library system cannot have their own electronic copy of the document.

To illustrate domain requirements that specify how a computation is carried out, consider Figure 6.7, taken from the requirements specification for an automated train protection system. This system automatically stops a train if it goes through a red signal. This requirement states how the train deceleration is computed by the system. It uses domain

specific terminology. To understand it, you need some understanding of the operation of railway systems and train characteristics. The requirement for the train system illustrates a major problem with domain requirements. They are written in the language of the application domain (mathematical equations in this case), and it is often difficult for software engineers to understand them. Domain experts may leave information out of a requirement simply because it is so obvious to them. However, it may not be obvious to the developers of the system, and they may therefore implement the requirement in the wrong way.

**3.2 User requirements**

M.Sc. Semester – I (Computer Science) 43

Paper – II (Software Engineering)

The user requirements for a system should describe the functional and nonfunctional 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. Consequently, if you are writing user requirements, you should not use software jargon, structured notations or formal notations, or describe the requirement by describing the system implementation. You should write user requirements in simple language, with simple tables and forms and intuitive diagrams. However, various problems can arise when requirements are written in natural language sentences in a text document:

1. Lack of clarity - It is sometimes difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.

2. Requirements confusion Functional requirements, non-functional requirements, system goals and design information may not be clearly distinguished.

3. Requirements amalgamation several different requirements may be expressed together as a single requirement.

As an illustration of some of these problems, consider one of the requirements for the library shown in Figure 6.8. This 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 would have been better left to the system requirements specification. It is good practice to separate user requirements from more detailed system requirements in a requirements document. Otherwise, non-technical readers of the user requirements may be overwhelmed by details that are really only relevant for technicians. Figure 6.9 illustrates this confusion. This example is taken from an actual requirements document for a CASE tool for editing software design models. The user may specify that a grid should be displayed so that entities may be accurately positioned in a diagram. The first sentence mixes up three kinds of requirements.

1. A conceptual, functional requirement states that the editing system should provide a grid. It presents a rationale for this.

2. A non-functional requirement giving detailed information about the grid units (centimeters or inches).

M.Sc. Semester – I (Computer Science) 44

Paper – II (Software Engineering)

3. A non-functional user interface requirement that defines how the grid is switched on and off by the user.

The requirement in Figure 6.9 also gives some but not all initialization information. It defines that the grid is initially off. However, it does not define its units when turned on. It provides some detailed information—namely, that the user may toggle between units—but not the spacing between grid lines. 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. I have rewritten the editor grid requirement (Figure 6.10) to focus only on the essential system features. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included and is particularly useful when requirements are changed. For example, the rationale in Figure 6.10 recognizes that an active grid where positioned objects automatically ‘snap’ to a grid line can be useful. However, this has been deliberately rejected in favor of manual positioning. If a change to this is proposed at some later stage, it will be clear that the use of a passive grid was deliberate rather than an implementation decision. To minimize misunderstandings when writing user requirements, I recommend that you follow some simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. The format I use shows the initial requirement in boldface, including a statement of rationale with each user requirement and a reference to the more detailed system requirement specification. You may also include information on who proposed the requirement (the requirement source) so that you know whom to consult if the requirement has to be changed.

2. Use language consistently. You should always distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using ‘shall’. Desirable requirements are not essential and are written using ‘should’.

3. Use text highlighting (bold, italic or colour) to pick out key parts of the requirement. 4. Avoid, as far as possible, the use of computer jargon. Inevitably, however, detailed technical terms will creep into the user requirements.

M.Sc. Semester – I (Computer Science) 45

Paper – II (Software Engineering)

The Robertsons (Robertson and Robertson, 1999), in their book that covers the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, supporting materials, and so on. This extends the format that I have used in Figure 6.10, and it can be used for both user and system requirements.

**3.3 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. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and consistent specification of the whole system. Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is impossible, in practice, to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different sub-systems that make up the system, this architectural definition is essential if you want to reuse software components when implementing the system.

2. In most cases, systems must interoperate with other existing systems. These constrain the design, and these constraints impose requirements on the new system.

3. The use of a specific architecture to satisfy non-functional requirements (such as N-version programming to achieve reliability, may be necessary.

An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified be used. Natural language is often used to write system requirements specifications as well as user requirements. However, because system requirements are more detailed than user requirements, natural language specifications can be confusing and hard to understand:

1. Natural language understanding relies on the specification readers and writers using the same words for the same concept. This leads to misunderstandings because of the ambiguity

M.Sc. Semester – I (Computer Science) 46

Paper – II (Software Engineering)

of natural language. Jackson (Jackson, 1995) gives an excellent example of this when he discusses signs displayed by an escalator. These said ‘Shoes must be worn’ and ‘Dogs must be carried’. I leave it to you to work out the conflicting interpretations of these phrases. 2. A natural language requirements specification is over flexible. You can say the same thing in completely different ways. It is up to the reader to find out when requirements are the same and when they are distinct.

3. There is no easy way to modularize natural language requirements. It may be difficult to find all related requirements. To discover the consequence of a change, you may have to look at every requirement rather than at just a group of related requirements. Because of these problems, requirements specifications written in natural language are prone to misunderstandings. These are often not discovered until later phases of the software process and may then be very expensive to resolve.

 It is essential to write user requirements in a language that non-specialists can understand. However, you can write system requirements in more specialized notations

(Figure 6.11). These include stylised, structured natural language, graphical models of the requirements such as use-cases to formal mathematical specifications.

**3.3.1 Structured language specifications**

M.Sc. Semester – I (Computer Science) 47

Paper – II (Software Engineering)

Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way. The advantage of this approach is that it maintains most of the expressiveness and understandability of natural language but ensures that some degree of uniformity is imposed on the specification. Structured language notations limit the terminology that can be used and use templates to specify system requirements. They may incorporate control constructs derived from programming languages and graphical highlighting to partition the specification. An early project that used structured natural language for specifying system requirements is described by Heninger (Heninger, 1980). Special-purpose forms were designed to describe the input, output and functions of an aircraft software system. The system requirements were specified using these forms.

 To use a form-based approach to specifying system requirements, you must define one or more standard forms or templates to express the requirements. The specification may be structured around the objects manipulated by the system, the functions performed by the

system or the events processed by the system. An example of such a form-based specification is shown in Figure 6.12. The insulin pump bases its computations of the user’s insulin

M.Sc. Semester – I (Computer Science) 48

Paper – II (Software Engineering)

requirement on the rate of change of blood sugar levels. These rates of change computed using the current and previous readings. You can download a complete version of the specification for the insulin pump from the book’s web pages. When a standard form is used for specifying functional requirements, the following information should be included:

1. Description of the function or entity being specified

2. Description of its inputs and where these come from

3. Description of its outputs and where these go to

4. Indication of what other entities are used (the requires part)

5. Description of the action to be taken

6. If a functional approach is used, a pre-condition setting out what must be true before the function is called and a post-condition specifying what is true after the function is called 7. Description of the side effects (if any) of the operation.

Using formatted specifications removes some of the problems of natural language specification. Variability in the specification is reduced and requirements are organized more effectively. However, it is difficult to write requirements in an unambiguous way, particularly when complex computations are required. You can see this in the description shown in Figure 6.12, where it isn’t made clear what happens if the pre-condition is not satisfied. To address this problem, you can add extra information to natural language requirements using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system and how sequences of actions are performed. Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these.

Figure 6.13 is a revised description of the computation of the insulin dose. Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions. Figure 6.14 illustrates the sequence of actions when a user wishes to withdraw cash from an automated teller machine (ATM). You should read a sequence diagram from top to bottom to see the order of the actions that take place. In Figure 6.14, there are three basic sub-sequences:

1. Validate card - The user’s card is validated by checking the card number and user’s PIN.

M.Sc. Semester – I (Computer Science) 49

Paper – II (Software Engineering)

2. Handle request - The user’s request is handled by the system. For a withdrawal, the database must be queried to check the user’s balance and to debit the amount withdrawn. Notice the exception here if the requestor does not have enough money in their account. 3. Complete transaction the user’s card is returned and, when it is removed, the cash and receipt are delivered.

**3.4 Interface specification**

Almost all software systems must operate with existing systems that have already been implemented and installed in an environment. If the new system and the 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 (perhaps as an appendix) in the requirements document. There are three types of interface that may have to be defined:

M.Sc. Semester – I (Computer Science) 50

Paper – II (Software Engineering)

1. Procedural interfaces where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces (APIs).

2. Data structures that are passed from one sub-system to another. Graphical data models are the best notations for this type of description. If necessary, program descriptions in Java or C++ can be generated automatically from these descriptions.

3. Representations of data (such as the ordering of bits) that have been established for an existing sub-system.

These interfaces are most common in embedded, real-time system. Some programming languages such as Ada (although not Java) support this level of specification. However, the best way to describe these is probably to use a diagram of the structure with annotations explaining the function of each group of bits. They are rarely used in practice for interface specification although, in my view, they are ideally suited for this purpose. A programming language such as Java can be used to describe the syntax of the interface. However, this has to be supplemented by further description explaining the semantics of each of the defined operations.

Figure 6.15 is an example of a procedural interface definition defined in Java. In this case, the interface is the procedural interface offered by a print server. This manages a queue of requests to print files on different printers. Users may examine the queue associated with a printer and may remove their print jobs from that queue. They may also switch jobs from one printer to another. The specification in Figure 6.15 is an abstract model of the print server that does not reveal any interface details. The functionality of the interface operations can be defined using structured natural language or tabular description.

**6.5 The software requirements document**

The software requirements document (sometimes called the software requirements specification or SRS) is the official statement 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. In some cases, the user and system requirements may be integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

M.Sc. Semester – I (Computer Science) 51

Paper – II (Software Engineering)

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. Figure 6.16, taken from my book with Gerald Kotonya on requirements engineering (Kotonya and Sommerville, 1998) illustrates possible users of the document and how they use it.

The diversity of possible users means that the requirements 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. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements. The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used.

When the system will be developed by an external contractor, critical system specifications need to be precise and very detailed. When there is more flexibility in the requirements and where an in-house, iterative development process is used, the requirements document can be much less detailed and any ambiguities resolved during development of the system. A number of large organizations, such as the US Department of Defense and the IEEE, have defined standards for requirements documents. Davis (Davis, 1993) discusses some of these standards and compares their contents. The most widely known

M.Sc. Semester – I (Computer Science) 52

Paper – II (Software Engineering)

standard is IEEE/ANSI 830-1998 (IEEE, 1998). This 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 and 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 cover functional, non-functional and interface requirements. This is obviously the most substantial part of the document but because of the wide variability in organizational practice, it is not appropriate to define a standard structure for this section. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.

4. Appendices

5. Index Although the IEEE standard is not ideal, it contains a great deal of good advice on how to write requirements and how to avoid problems. It is too general to be an organizational standard in its own right. It is a general framework that can be tailored and adapted to define a standard geared to the needs of a particular organisation.

Figure 6.17 illustrates a possible organization for a requirements document that is based on the IEEE standard. However, This was first proposed by Heninger (Heninger, 1980) and, helps the maintainers of the system and may allow designers to include support for future system features. Of course, the information that is included in a requirements document must depend on the type of software being developed and the approach to development that is used. If an evolutionary approach is adopted for a software product (say),

M.Sc. Semester – I (Computer Science) 53

Paper – II (Software Engineering)

the requirements document will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, non-functional system requirements. In this case, the designers and programmers use their judgment to decide how to meet the outline user requirements for the system. By contrast, when the software is part of a large system engineering project that includes interacting hardware and software systems, it is often essential to define the requirements to a fine level of detail.

For long documents, it is particularly important to include a comprehensive table of contents and document index so that readers can find the information that they need. Requirements documents are essential when an outside contractor is developing the software system. However, agile development methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort that is largely wasted. Rather than a formal document, approaches such as extreme programming (Beck, 1999) propose that user requirements should be collected incrementally and written on cards. The user then prioritizes requirements for implementation in the next increment of the system. For business systems where requirements are unstable, I think that this approach is a good one. However, I would argue that it is still useful to write a short supporting document that defines the business and dependability requirements for the system. It is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

---------------------------------------------------------------------------------------------------------------------

M.Sc. Semester – I (Computer Science) 54