M.Sc. Computer Science III Sem

Notes: Subject : Software Engineering

Unit 1: Introduction to Software Engineering

Computer software is the product that software engineers design and build. It includes the programs that execute within a computer of any size and architecture, documents that encompass hard-copy and virtual forms, and data that combine numbers and text but also includes representations of pictorial, video, and audio information. Computer software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities.

What is Software?: A textbook description of software might take the following form: 1. Software is a computer program that when executed provide desired function and performance. 2. Software is a data structure that enables the programs to adequately manipulate information.

1.1. Software Characteristics:

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.
2. Software doesn't "wear out." (Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.)
3. Although the industry is moving toward component-based assembly, most software continues to be custom built.

1.2. Software Applications

1. **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

2. **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.
3. **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., pointof-sale transaction processing).
4. **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.

5. **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).
6. **Personal computer software:** The personal computer software market has grown rapidly 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.
7. **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.
8. **Artificial intelligence software:** Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, al so called knowledgebased systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category

1.3. Software Myths : Software myths are categories into following categories such as Management myths, Practitioner's myths, Customers myths.

1.3.1. 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.

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

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

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

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

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

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

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

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

1.3.2. 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.

1.3.3. Practitioner's Myths:

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

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

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

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

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

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

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

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

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

1.4. Software Engineering: A Layered Technology

General engineering is the approach for analysis, design, construction, verification and maintenance of any technical entity. If this general engineering approach used for software development then it is called as software engineering. Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer is; “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”.

The IEEE has developed a more comprehensive definition of Software engineering: Software Engineering: (1) It is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Software engineering is a layered technology (Process, Methods, and Tools):

Software engineering is a layered technology. Referring to following figure 2.1, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management and similar philosophies stand in a continuous process improvement culture, and this culture ultimately leads to the development of increasingly more mature approaches to software engineering. The foundation that supports software engineering is a **quality focus**.

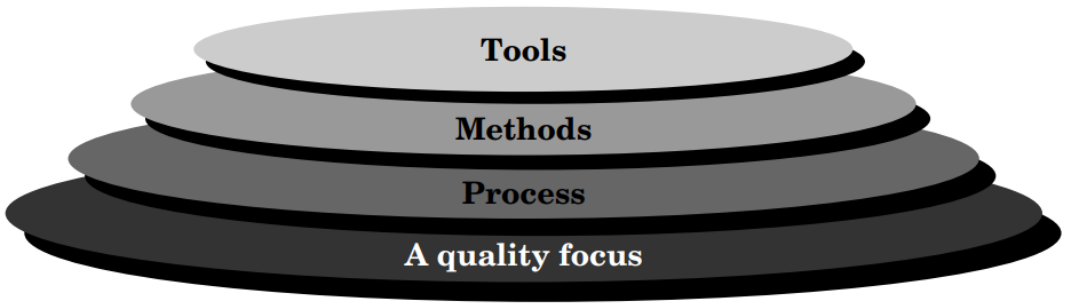


Figure 1.1. Software engineering layers

The foundation for software engineering is the **process layer**. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. **Process** defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology. 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 trust on a set of basic principles that covers 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.

1.5. A Generic View of Software Engineering

Engineering is the analysis, design, construction, verification, and management of technical 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?
- 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.

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

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.

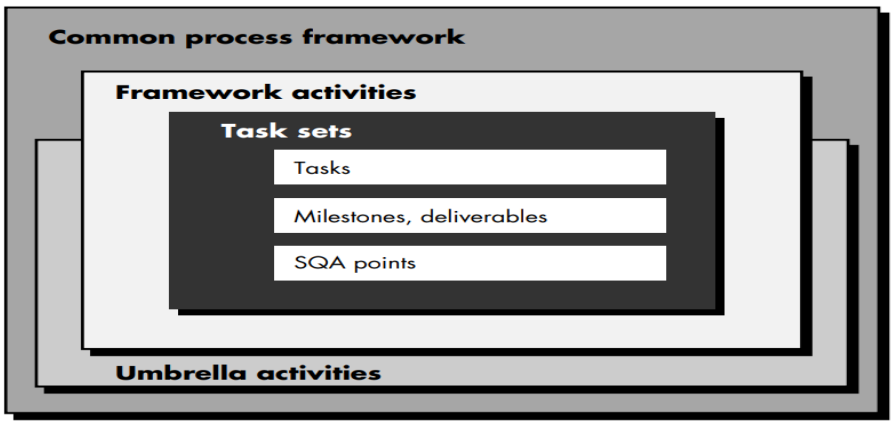
1. The **definition phase** focuses on **what**. That is, during definition, the software engineer attempts to identify what information is to be processed, what function and performance are desired, what system behavior can be expected, what interfaces are to be established, what design constraints exist, and what validation criteria are required to define a successful system. The key requirements of the system and the software are identified.
2. 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 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.
3. The **support phase** focuses on **change** associated with error correction, adaptations required 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:
    a. **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.
    b. **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.

c. **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.

d. **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.

1.6. The Software Process:



A software process can be characterized as shown in above 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 measurement2—overlay the process model.

Umbrella activities are independent of any one framework activity and occur throughout the process.

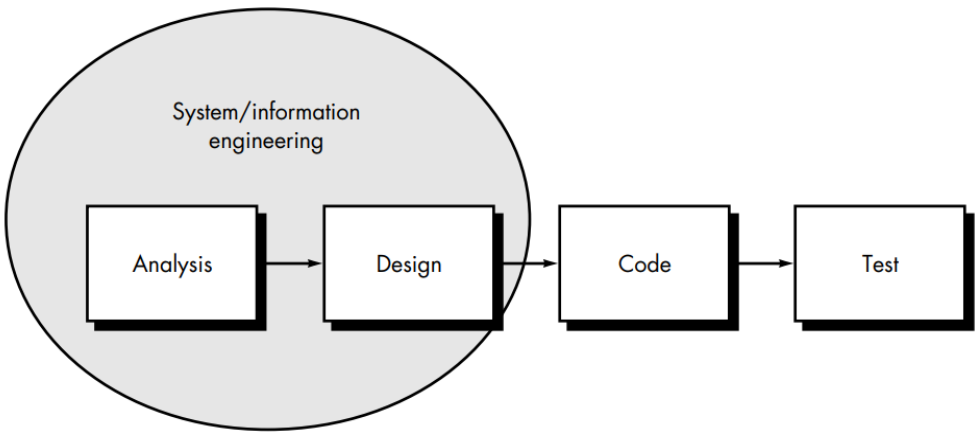
1.7. Software Process Models:

To solve actual problems in an industry, a software engineer or a team of engineers must incorporate a development strategy that includes the process, methods, and tools described in Section 2.1.1 and the generic phases discussed in Section 2.1.2. This strategy is often referred 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.

1.7.1. Linear Sequential Model:

This model also as called classic life cycle or waterfall model, linear sequential model suggests a systematic and sequential approach to the 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.

FIGURE 2.4
The linear sequential model



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 necessary when software must interact with other

elements such as hardware, people, and databases. System engineering and analysis includes the requirements gathering at the system level with a small amount of top level design and analysis. Information engineering includes requirements gathering at the strategic business level and at the business area level.

Software requirements analysis: The requirements gathering process is mainly focused specifically on software. To understand the nature of the program to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behavior, performance, and interface. Requirements for 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 (blueprint) 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 i.e. code. The code generation step performs this task. If design is performed in a detailed manner, code generation can be done systematically.

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

Support: Software will certainly undergo change after it is delivered to the customer. Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral

device), or because the customer requires functional or performance enhancements. Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one.

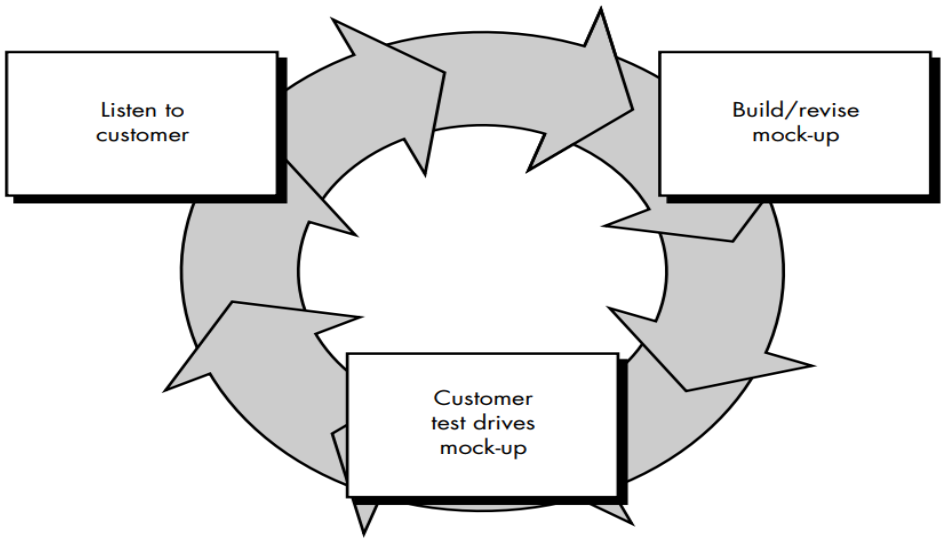
The linear sequential model is the oldest and the most widely used paradigm for software engineering. However, criticism of the paradigm has caused even active supporters to question its efficacy. Among the problems that are sometimes encountered when the linear sequential model is applied are:

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.

1.7. 2. The Prototyping Model:

The prototyping paradigm model 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.

FIGURE 2.5
The prototyping paradigm



Ideally, the prototype serves as a mechanism for identifying software requirements.

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. But, prototyping can also be problematic for the following reasons:

1. It is 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 and demands that "a few doses" be applied to make the prototype a working product.
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.

   Although problems can occur, prototyping can be an effective paradigm for software engineering.

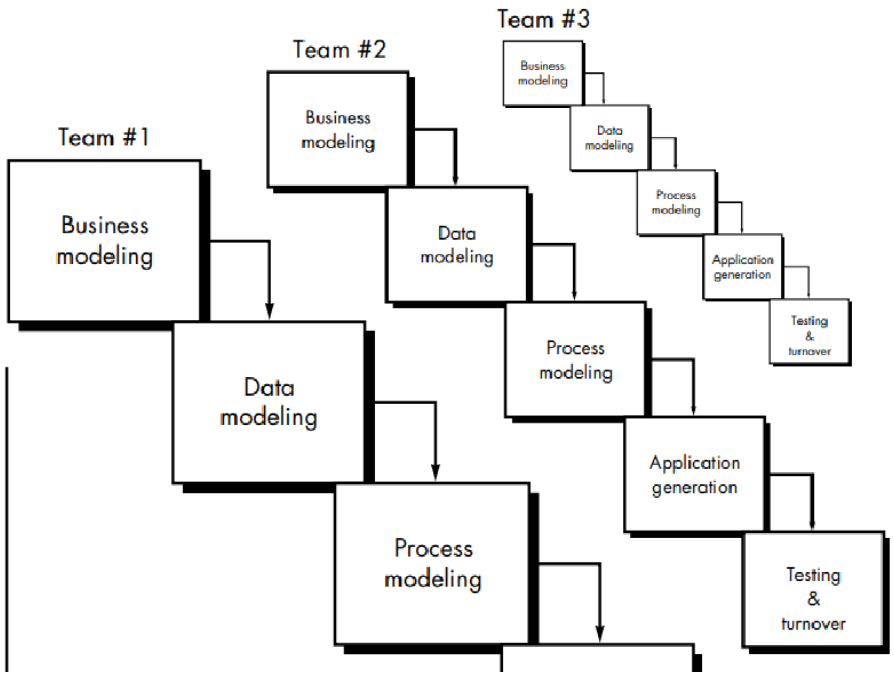
1.7.3 The Rad Model:

Rapid application development (RAD) is an incremental software development process model that focuses on short development cycle. The RAD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using the component-based construction. If requirements are well understood and project scope is defined, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days). The RAD approach includes following phases:

1. **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?
2. **Data modeling**: The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristics (called attributes) of each object are identified and the relationships between these objects defined.
3. **Process modeling:** The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.
4. **Application generation:** RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.
5. **Testing and turnover:** Since the RAD process focuses on 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.

FIGURE 2.6
The RAD model



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 short 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 changing 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.

1.7.4. The Incremental Model :

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative style like prototyping model. 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 reusable components. When an incremental model is used, the first increment is often a core product.

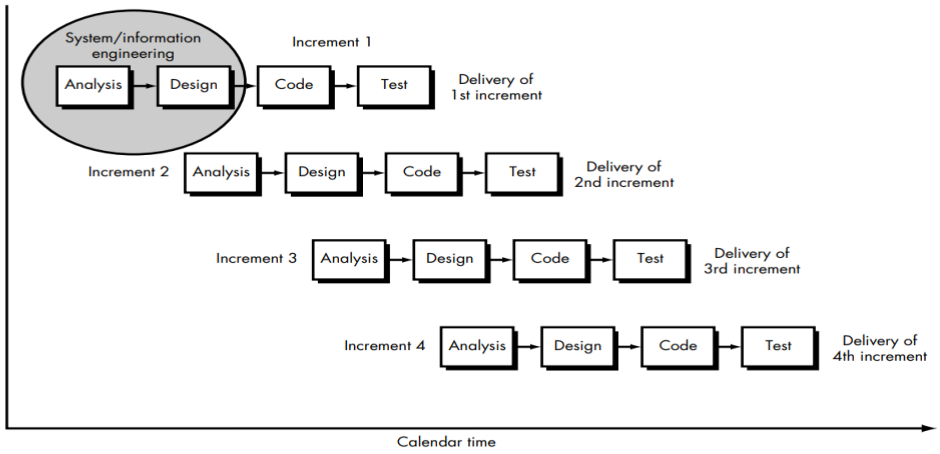


FIGURE 2.7
The incremental model

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

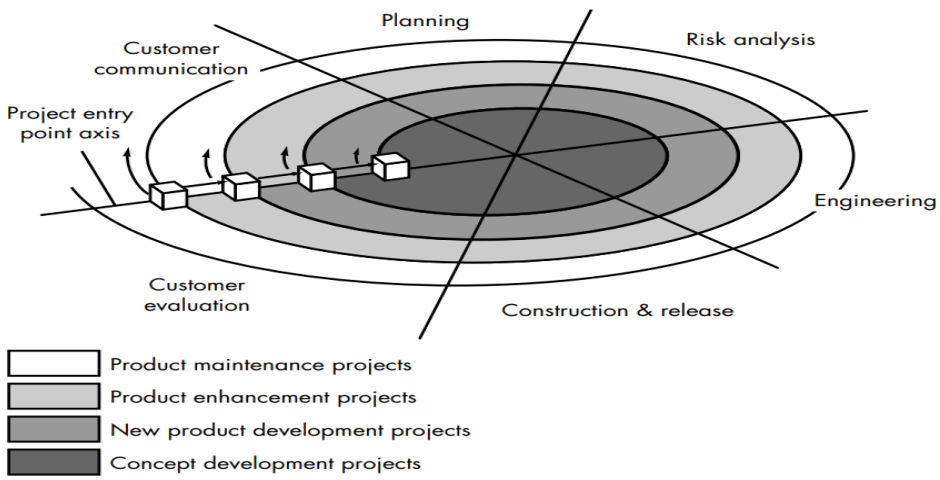
The incremental process model, like prototyping model and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are stripped down versions of the final product, but they 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.

1.7.5. Spiral Model :

The spiral model, originally proposed by Boehm and it 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 regions. Typically, there are between three and six task regions.

FIGURE 2.8
A typical spiral model



- Customer communication—tasks required to establish effective communication between developer and customer.
- Planning—tasks required to define resources, timelines, and other projectrelated 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) noted in are applied.

1.8. Fourth Generation Techniques (4GT):

The term fourth generation techniques (4GT) contain a broad array of software tools that have one thing in common: each helps the software engineer to specify some characteristic of software at a high level. The tool then automatically generates source code based on the developer's specification. The 4GT paradigm for software engineering focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problem to be solved in terms that the customer can understand.

Currently, a software development environment that supports the 4GT paradigm includes some or all of the following tools: nonprocedural languages for database query, report generation, data manipulation, screen interaction and definition, code generation; high-level graphics capability; spreadsheet capability, and automated generation of HTML and similar languages used for

Web-site creation using advanced software tools. Initially, many of the tools noted previously were available only for very specific application domains, but today 4GT environments have been extended to address most software application categories.

There is some merit in the claims of both sides and it is possible to summarize the current state of 4GT approaches:

1. The use of 4GT is a viable approach for many different application areas. Coupled with computer-aided software engineering tools and code generators, 4GT offers a credible solution to many software problems.
2. Data collected from companies that use 4GT indicate that the time required to produce software is greatly reduced for small and intermediate applications and that the amount of design and analysis for small applications is also reduced.
3. However, the use of 4GT for large software development efforts demands as much or more analysis, design, and testing (software engineering activities) to achieve substantial time savings that result from the elimination of coding.

=======End of Chapter 1========

Unit 2 / Chapter 2 : Software Process and Project Metrics

Software metrics is a broad range of measurements for computer software. Measurement can be applied to the software process with the purpose of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help to assess the quality of technical work products and to assist in strategic decision making as a project proceeds.

There are four (4) reasons for measuring the software processes, products, and resources that is **to characterize, to evaluate, to predict, or to improve.**

1. **We characterize** to gain understanding of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments.
2. **We evaluate** to determine status with respect to plans. We also evaluate to assess achievement of quality goals and to assess the impacts of technology and process improvements on products and processes.
3. **We predict** so that we can plan. Measuring for prediction involves gaining understandings of relationships among processes and products and building models of these relationships, so that the values we observe for some attributes can be used to predict others.
4. **We measure to improve** when we gather quantitative information to help us identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.

2.1. Measures, Metrics, and Indicators

In the software engineering context, a measure provides a quantitative indication of the extent (level), amount, dimension, capacity, or size of some attribute of a product or process. Measurement is the act of determining a measure. The IEEE defines metric as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute." A software engineer collects measures and develops metrics so that indicators will be obtained. An indicator is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the process to make things better.

2.2. Metrics In The Process And Project Domains

Metrics should be collected so that process and product indicators can be ascertained. Process indicators enable a software engineering organization to gain insight into the efficacy of an existing process (i.e., the paradigm, software engineering tasks, work products, and milestones). They enable managers and practitioners to assess what works and what doesn't. Process metrics are collected across all projects and over long periods of time. Their intent is to provide indicators that lead to long-term software process improvement.

Project indicators enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

2.2.1. Process Metrics and Software Process Improvement

The only normal way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement. But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance."

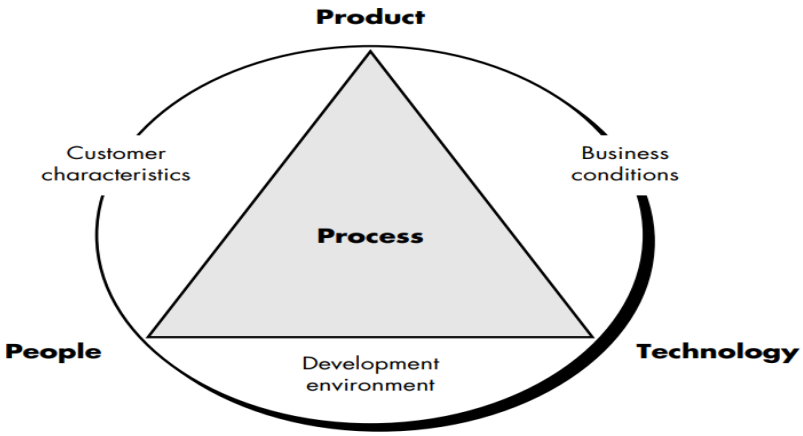


FIGURE 4.1 Determinants for software quality and organizational effectiveness (adapted from [PAU94])

The process sits at the center of a triangle connecting three factors that have a reflective influence on software quality and organizational performance. The

skill and motivation of people has been shown to be the single most influential factor in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e.,the software engineering methods) that populate the process also has an impact. In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., CASE tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication).

We measure the efficiency of a software process indirectly. That is, we derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end-users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, we might measure the effort and time spent performing the umbrella activities and the generic software engineering activities.

2.3. Project Metrics

Software process metrics are used for strategic purposes. But, Software project measures are calculated (strategic). That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.

The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control the progress.

As technical work commences, other project metrics begin to have significance.

Production rates represented in terms of pages of documentation, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from specification into design, technical metrics are collected to assess to assess design quality and to provide indicators that will influence the approach taken to code generation and testing.

The purpose of project metrics is twofold (dual). First is, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second is, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality. As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

2.4. Software Measurement

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly. Direct measures of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities"

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.

2.4.1. Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such as the one shown in Figure 4.4, can be created.

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
...	...	...	...	...	...	...	...

The table lists each software development project that has been completed over the past few years and corresponding measures for that project. Referring to the table entry (Figure 4.4) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha. Size-oriented metrics are not universally accepted as the best way to measure the process of software development.

2.4.2. Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since 'functionality' cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the function point. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity. Function points are computed by completing the table shown in Figure 4.5. Five information domain characteristics are determined and counts are provided in the appropriate table location. Information domain values are defined in the following manner:

FIGURE 4.5 Computing function points

			Weighting factor				
Measurement parameter	**Count**		**Simple**	**Average**	**Complex**		
Number of user inputs		×	3	4	6	=	
Number of user outputs		×	4	5	7	=	
Number of user inquiries		×	3	4	6	=	
Number of files		×	7	10	15	=	
Number of external interfaces		×	5	7	10	=	
Count total	→						

Number of user inputs: Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs: Each user output that provides application oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries: An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files: Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces: All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

$$FP = \text{count total} \ [0.65 + 0.01 \ \sum(Fi)]$$

Where count total is the sum of all FP entries obtained from Figure 4.5.

The Fi (i = 1 to 14) are "complexity adjustment values"

2.5. Reconciling Different Metrics Approaches

The relationship between lines of code and function points depend upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. The thesis of this work is that the amount of function to be provided by the application (program) can be estimated from the itemization of the major components8 of data to be used or provided by it. Furthermore, this estimate of function should be correlated to both the amount of LOC to be developed and the development effort needed. The following table provides rough estimates of the average number of lines of code required to build one function point in various programming languages:

Programming Language	LOC/FP (average)
Assembly language	320
C	128
COBOL	106
FORTRAN	106
Pascal	90
C++	64
Ada95	53
Visual Basic	32
Smalltalk	22
Powerbuilder (code generator)	16
SQL	12

A review of these data indicates that one LOC of C++ provides approximately 1.6 times the "functionality" (on average) as one LOC of FORTRAN. Furthermore, one LOC of a Visual Basic provides more than three times the functionality of a LOC for a conventional programming language. More detailed data on the relationship between FP and LOC are presented in [JON98] and can be used to "backfire" (i.e., to compute the number of function points when the number of delivered LOC are known) existing programs to determine the FP measure for each. LOC and FP measures are often used to derive productivity metrics. This invariably leads to a debate about the use of such data. Should the LOC/person-month (or FP/person-month) of one group be compared to similar data from another? Should managers appraise the performance of individuals by using these metrics? Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation, a historical baseline of information must be established.

2.6. Metrics For Software Quality

The quality of a system, application, or product is only as good as the requirements that describe the problem, the design that models the solution, the code that leads to an executable program, and the tests that exercise the software to uncover errors. A good software engineer uses measurement to assess the quality of the analysis and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-

time quality assessment, the engineer must use technical measures to evaluate quality in objective, rather than subjective ways. The project manager must also evaluate quality as the project progresses. Private metrics collected by individual software engineers are assimilated to provide project level results. Although many quality measures can be collected, the primary thrust at the project level is to measure errors and defects. Metrics derived from these measures provide an indication of the effectiveness of individual and group software quality assurance and control activities. Metrics such as work product (e.g., requirements or design) errors per function point, errors uncovered per review hour, and errors uncovered per testing hour provide insight into the efficacy of each of the activities implied by the metric. Error data can also be used to compute the defect removal efficiency (DRE) for each process framework activity.

2.6.1. An Overview of Factors That Affect Quality

Over 25 years ago, McCall and Cavano defined a set of quality factors that were a first step toward the development of metrics for software quality. These factors assess software from three distinct points of view: (1) product operation (using it), (2) product revision (changing it), and (3) product transition (modifying it to work in a different environment; i.e., "porting" it). In their work, the authors describe the relationship between these quality factors (what they call a framework) and other aspects of the software engineering process:

First, the framework provides a mechanism for the project manager to identify what qualities are important. These qualities are attributes of the software in addition to its functional correctness and performance which have life cycle implications. Such factors as maintainability and portability have been shown in recent years to have significant life cycle cost impact . . .

Secondly, the framework provides a means for quantitatively assessing how well the development is progressing relative to the quality goals established .

Thirdly, the framework provides for more interaction of QA personnel throughout the development effort . . .Lastly, . . . quality assurance personal can use indications of poor quality to help identify [better] standards to be enforced in the future.

2.6.2. Measuring Quality

Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team. Gilb suggests definitions and measures for each.

Correctness: A program must operate correctly or it provides little value to its users. Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the overall quality of a software product, defects are those problems reported by a user of the program after the program has been released for general use. For quality assessment purposes, defects are counted over a standard period of time, typically one year.

Maintainability: Software maintenance accounts for more effort than any other software engineering activity. Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change.

Integrity: Software integrity has become increasingly important in the age of hackers and firewalls. This attribute measures a system's ability to withstand attacks (both accidental and intentional) to its security. Attacks can be made on all three components of software: programs, data, and documents. To measure integrity, two additional attributes must be defined: threat and

Security: Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given

time. Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

integrity = summation [(1 – threat) (1 – security)]

where threat and security are summed over each type of attack

Usability: The catch phrase "user-friendliness" has become ubiquitous in discussions of software products. If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efficient in the use of the system, (3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system.

2.6.3. Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities. When considered for a project as a whole, DRE is defined in the following manner:

$$DRE = E/(E + D) \quad (4\text{-}4)$$

where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.

The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases (for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will

decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team to institute techniques for finding as many errors as possible before delivery.

=====End of Chapter 2======

Unit 3/ Chapter 3: Software Design/User Interface Design

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

In the software engineering context, design focuses on four major areas of concern: **data, architecture, interfaces, and components**. Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software. The design task produces a data design, an architectural design, an interface design, and a component design.

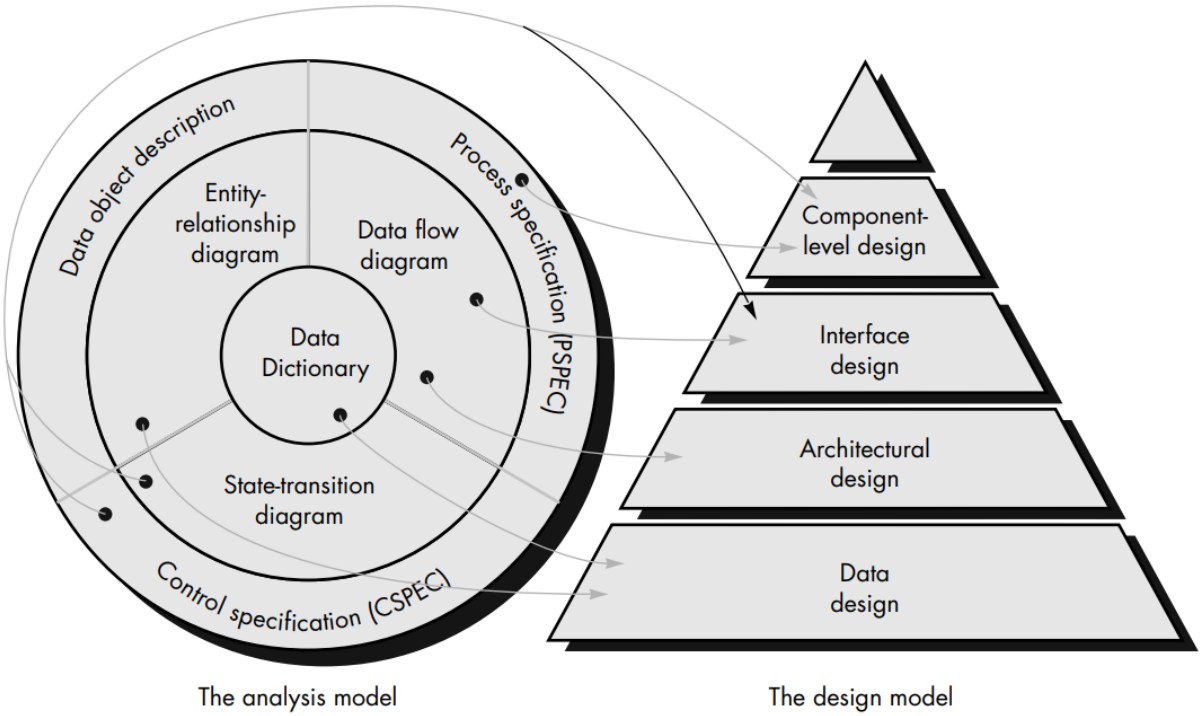
The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content shown in the data dictionary provide the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

The architectural design defines the relationship between major structural elements of the software, the “design patterns” that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied.

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

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

During the software design we make decisions that will ultimately affect the success of software construction. The importance of software design can be stated with a single word—quality. Design is the place where quality is look after in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support steps that follow. Without design, we risk building an unstable system—one that will fail when small changes are made;



3.1. The Software Design Process

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software.

Design and Software Quality

Throughout the design process, the quality of the developing design is assessed with a series of formal technical reviews or design walkthroughs. McGlaughlin author suggests three characteristics that help as a guide for the evaluation of a good design:

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

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

c. The design should provide a complete picture of the software, addressing the

data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. In order to evaluate the quality of a design representation, we must establish technical criteria for good design. The following are the guidelines for good software design:

1. A design should exhibit an architectural structure that (1) has been created using recognizable design patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. The design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and sub functions.
3. A design should contain distinct representations of data, architecture, interfaces, and components (modules).
4. A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

3.2. Software Design Principles

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the

software to be built. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design. The following are the design principles:

1. **The design process should not suffer from "tunnel vision."** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.
2. **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
3. **The design should not reinvent the wheel**. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. **The design should "minimize the intellectual distance"** between the software and the problem as it exists in the real world.
5. **The design should exhibit uniformity and integration**. A design is uniform if it appears that one person developed the entire thing.
6. **The design should be structured to accommodate change.** The design concepts discussed in the next section enable a design to achieve this principle.
7. **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Welldesigned software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

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

.

3.3. Software Design Concepts:

A set of fundamental software design concepts has evolved over the past four decades. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps the software engineer to answer the following questions:

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

Fundamental software design concepts provide the necessary framework for "getting it right."

3.3.1. Abstraction:

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

1. (“Abstraction is one of the fundamental ways that we as humans cope with complexity.”)
2. The psychological notion of "abstraction" permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details; use of abstraction also permits one to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure.

   .

   During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

   As we move through different levels of abstraction, we work to create procedural abstractions and data abstractions.

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

   A data abstraction is a named collection of data that describes a data object. we can define a data abstraction called **door**. Like any data object,

the data abstraction for door would include a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

3.3.2. Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. In each step of the refinement, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of any underlying computer or programming language. Every refinement step implies some design decisions.

3.3.3. Modularity:

Software architecture represents modularity; that is, software is divided into separately named and addressable components called as modules, that are integrated to satisfy problem requirements. This leads to a "divide and conquer" method that conclusion—it's easier to solve a complex problem when you break it into manageable pieces

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

3.3.4. Software Architecture:

Software architecture refers to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”. Software architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. One goal of software design is to derive an architectural execution of a system. This execution serves as a framework from which more detailed design activities are conducted. A set

of architectural patterns enable a software engineer to reuse design level concepts.

3.4. EFFECTIVE MODULAR DESIGN:

A modular design reduces complexity, enables the change and results in easier implementation by encouraging parallel development of different parts of a system.

3.4.1. Functional Independence

The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. Functional independence is a key to good design, and design is the key to software quality.

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

Cohesion

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

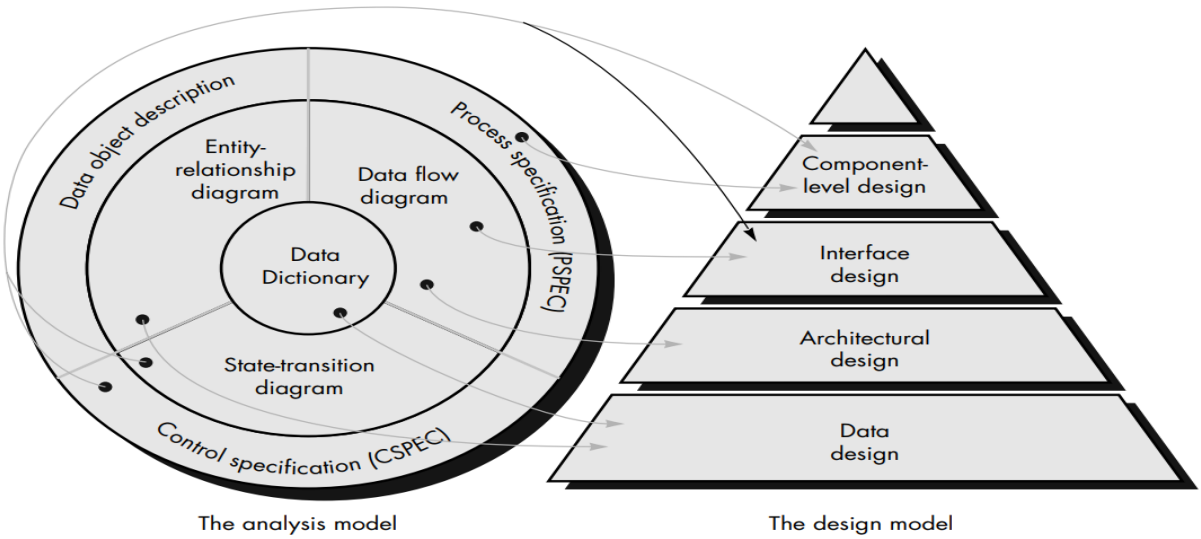
Coupling

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the

point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" [STE74], caused when errors occur at one location and propagate through a system.

3.5. The Software Design Model

The design principles and concepts discussed earlier establish a foundation for the creation of design model that includes representations of data, architecture, interfaces, and components. Like the analysis model before it, each of these design representations is tied to the others, and all can be traced back to software requirements.



The design model was represented as a pyramid. The symbolism of this shape is important. A pyramid is an extremely stable object with a wide base and a low center of gravity. Like the pyramid, we want to create a software design that is stable. By establishing a broad foundation using data design, a stable mid-region with architectural and interface design, and a sharp point by applying component-level design, we create a design model that is not easily “Sloped over” by the winds of change.

3.6. User Interface Design (GUI design):

The blueprint for a house is not complete without a representation of doors, windows, and utility connections for water, electricity, and telephone (not to mention cable TV). The "doors, windows, and utility connections" for computer software make up the interface design of a system.

Interface design focuses on three areas of concern: (1) the design of interfaces between software components, (2) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and (3) the design of the interface between a human (i.e., the user) and the computer. In this chapter we focus exclusively on the third interface design category—user interface design.

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

User interface design begins with the identification of users, tasks, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items.

3.6.1. Golden Rules of User Interface Design:

1. Place the user in control.

2. Reduce the user's memory load.

3. Make the interface consistent.

1. **Place the user in control.**

a) Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
b) Provide for flexible interaction.
c) Allow user interaction to be interruptible and undoable.
d) Streamline interaction as skill levels advance and allow the interaction to be customized.
e) Hide technical internals from the casual user
f) Design for direct interaction with objects that appear on the screen.

2. Reduce the user's memory load

The more a user has to remember, the more error-prone will be the interaction with the system.

a) **Reduce demand on short-term memory:** The interface should be designed to reduce the requirement to remember past actions.
b) **Establish meaningful defaults.** The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.
c) **Define shortcuts that are inbuilt.** When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).
d) **The visual layout of the interface should be based on a real world metaphor (Symbol).** For example, a bill payment system should use a check book and check register metaphor to guide the user through the bill paying process.
e) **Disclose information in a progressive fashion**. The interface should be organized hierarchically.

3. Make the interface consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to a design

standard that is maintained throughout all screen displays, (2) input mechanisms are controlled to a limited set that are used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented. Mandel author defines a set of design principles that help to make the interface consistent:

a) **Allow the user to put the current task into a meaningful context.** Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand.

b) **Maintain consistency across a family of applications**. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

c) **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.** Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

3.6.2. User Interface Design Process:

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

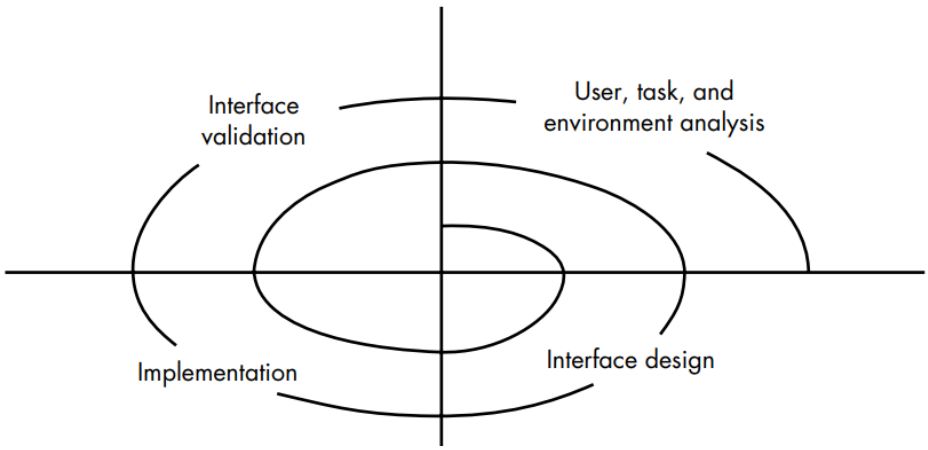
3.6.2.1. Interface Design Models:

Four different models come into play when a user interface is to be designed. The software engineer creates a design model, the software engineer establishes a user model, the end-user develops a mental image that is often called the

user's model or the system perception, and the implementers of the system create a system image. The role of interface designer is to resolve these differences and derive a consistent representation of the interface.

A design model of the entire system includes data, architectural, interface, and procedural representations of the software. The user model establishes the profile of end-users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality".

FIGURE 15.1
The user interface design process



3.6.2.2. The User Interface Design Process

The design process for user interfaces is iterative and can be represented using a spiral model. Referring to Figure 15.1, the user interface design process includes four distinct activities:

1. User, task, and environment analysis and modeling
2. Interface design
3. Interface construction
4. Interface validation

The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are

defined. For each user category, requirements are produced. In core part, the software engineer attempts to understand the system perception for each class of users. Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to fulfill the goals of the system are identified, described, and elaborated.

The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system. The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn; and (3) the users' acceptance of the interface as a useful tool in their work

3.6.2.3. Interface Design Activities:

Once task analysis has been completed, all tasks (or objects and actions) required by the end-user have been identified in detail and the interface design activity commences. The first interface design steps can be accomplished using the following approach:

1. Establish the goals and purposes for each task.
2. Map each goal and intention to a sequence of specific actions.
3. Specify the action sequence of tasks and subtasks, also called a user scenario, as it will be executed at the interface level.
4. Indicate the state of the system; that is, what does the interface look like at the time that a user scenario is performed?
5. Define control mechanisms; that is, the objects and actions available to the

user to alter the system state.

6. Show how control mechanisms affect the state of the system.

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

=======End of Chapter 3=======

Unit 4 / Chapter 4 : Software Risk Analysis

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can outbreak a software project. Risks are a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a possibility plan should the problem actually occur. For this reason, understanding the risks and taking proactive measures to avoid or manage them is a key element of good software project management.

4.1. Reactive Vs Proactive Risk Strategies

There are two approaches for dealing with software risk such as reactive and proactive risk strategies. "Indiana Jones School of risk management". Said that Never worrying about problems until they happened, Indy would react in some heroic way. Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. .

More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire fighting mode. When this fails, "crisis management" takes over and the project is in real risk.

The more intelligent strategy for risk management is to be proactive. A proactive strategy begins long before technical work is started. The potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software development team creates a plan for managing risk. The primary objective of a team is to avoid risk, but because not all risks can be avoided, the team works to develop an emergency plan that will enable it to respond in a controlled and effective manner.

In this chapter, we will discuss a proactive strategy for risk management.

Software risks has two characteristics :

1) **Uncertainty—**the risk may or may not happen; that is, there are no 100% probable risks.
2) **Loss—**if the risk becomes a reality, unwanted consequences or losses will occur

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan, if project risks become real, then the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.

Business risks threaten the viability (practicality) of the software to be built. Business risks often threaten the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one

really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell, (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

It is extremely important to note that simple categorization won't always work. Some risks are simply unpredictable in advance.

4.2. Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them.

There are two distinct types of risks for each of the categories, generic risks and product-specific risks. Generic risks are a potential threat to every software project. Product-specific risks can be identified only by those with a clear understanding of the technology, people, and environment that is specific to the project at hand.

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

- **Product size**—risks associated with the overall size of the software to be built or modified.
- **Business impact**—risks associated with constraints imposed by management or the marketplace.
- **Customer characteristics**—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

- **Process definition**—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- **Development environment**—risks associated with the availability and quality of the tools to be used to build the product.
- **Technology to be built**—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- **Staff size and experience**—risks associated with the overall technical and project experience of the software engineers who will do the work.

4.3. Risk Projection:

Risk projection, also called risk estimation, attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. The project planner, along with other managers and technical staff, performs four risk projection activities: (1) establish a scale that reflects the assumed probability of a risk, (2) define/outline the consequences of the risk, (3) estimate the impact of the risk on the project and the product, and (4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

4.3.1. Developing a Risk Table:

A risk table provides a project manager with a simple technique for risk projection. A sample risk table is illustrated in Figure 6.2.

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

A project team begins by listing all risks in the first column of the table. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. The impact of each risk is assessed.

Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks comes to the top of the table, and low-probability risks drop to the bottom. The project manager studies the resultant sorted table and defines a cutoff line. The cutoff line (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are re-evaluated to accomplish second-order prioritization. This accomplishes first-order risk prioritization. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow. All risks that lie above the cutoff line must be managed. The column labeled RMMM contains a pointer into a

Risk Mitigation, Monitoring and Management Plan or alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff.

4.3.2. Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware (a technical risk) will prevent early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many customers are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, a project manager might want the “bad news” to occur as soon as possible, but in some cases, the longer the delay, the better.

The following steps are recommended to determine the overall consequences of a risk:

1. Determine the average probability of occurrence value for each risk component.
2. Determine the impact for each component based on the criteria shown.
3. Complete the risk table and analyze the results as described in the preceding

4.4. Risk Refinement:

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage. One way to do this is to represent the risk in condition-transition-consequence (CTC) format . That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 6.4.2, we can write:

Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

4.5. Risk Mitigation, Monitoring, And Management (RMMM)

The main purpose of risk analysis activity is to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:

- Risk avoidance
- Risk monitoring
- Risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for risk mitigation. For example, assume that high staff turnover is noted as a project risk, r1. Based on past history and management perception, the likelihood, l1, of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact, x1, is projected at level 2. That is, high turnover will have a critical impact on project cost and schedule. To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under our control before the project starts.
- Once the project commences, assume turnover will occur and develop echniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.

- Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, risk monitoring activities starts. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less possible. In the case of high staff turnover, the following factors can be monitored:

• General attitude of team members based on project pressures.

• The degree to which the team has jelled.

• Interpersonal relationships among team members.

• Potential problems with compensation and benefits.

• The availability of jobs within the company and outside it

In addition to monitoring these factors, the project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of documentation standards and mechanisms to be sure that documents are developed in a timely manner occur." The project manager should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, the project manager may temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling

newcomers who must be added to the team to “get up to speed.” Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode.” This might include video-based knowledge capture, the development of “commentary documents,” and/or meeting with other team members who will remain on the project.

4.5.1. The RMMM Plan:

A risk management strategy can be included in the software project plan or the risk management steps can be organized into a separate Risk Mitigation, Monitoring and Management Plan. The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk.

======End of chapter 4 =======

Unit 5 / Chapter 5. Software Testing Techniques

Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors but how? That’s where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic of software components, and (2) exercise the input and output

domains of the program to uncover errors in program function, behavior and performance. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation. It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing.

5.1. Testing Objectives

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

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

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

3. A successful test is one that uncovers an as-yet-undiscovered error.

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

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.

5.2. Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggests a set of testing principles :

• **All tests should be traceable to customer requirements.** As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.

• **Tests should be planned long before testing begins.** Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been set.

• **The Pareto principle applies to software testing.** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

• **Testing should begin "in the small" and progress toward testing "in the large."** The first tests planned and executed generally focus on individual components. As testing progresses, focus hits in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

• **Exhaustive (complete/comprehensive) testing is not possible.** The number of path permutations for even a moderately sized program is exceptionally large For this reason, it is impossible to execute every combination of paths during testing.

• **To be most effective, testing should be conducted by an independent third party.** By most effective, we mean testing that has the highest probability of finding errors.

5.3. Testability:

Software testability is simply how easily a software program can be tested. Since testing is so deeply difficult, it pays to know what can be done to streamline it. Sometimes programmers are willing to do things that will help the testing process and a checklist of possible design points, features, etc., can be useful in negotiating with them. There are certainly metrics that could be used to measure testability in most of its aspects. Sometimes, testability is used to

mean how adequately a particular set of tests will cover the product. It's also used by the military to mean how easily a tool
can be checked and repaired in the field.

5.4. Operability : The better it works, the more efficiently it can be tested."

• The system has few bugs (bugs add analysis and reporting overhead to the test process).

• No bugs block the execution of tests.

• The product evolves in functional stages (allows simultaneous development and testing).

5.5. Observability. "What you see is what you test."

• Distinct output is generated for each input.

• System states and variables are visible or queriable during execution.

• Past system states and variables are visible or queriable (e.g., transaction logs).

• All factors affecting the output are visible.

• Incorrect output is easily identified.

• Internal errors are automatically detected through self-testing mechanisms.

• Internal errors are automatically reported.

• Source code is accessible.

Kaner, Falk, and Nguyen suggest the following attributes of a “good” test:

1. A good test has a high probability of finding an error.
2. A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test.
3. A good test should be “best of breed”
4. A good test should be neither too simple nor too complex.

5.6. White-Box Testing:

White-box testing, sometimes called glass-box testing, is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that :

(1) Guarantee that all independent paths within a module have been exercised at least once,

(2) Exercise all logical decisions on their true and false sides,

(3) Execute all loops at their boundaries and within their operational bounds

(4) Exercise internal data structures to ensure their validity.

5.7. Black-Box Testing:

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing. Because black-box testing purposely neglects control structure, attention is focused on the information domain.

5.8. Differences between Black Box vs White Box Testing:

1. **Black Box Testing** is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester

2. **White Box Testing** is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.

Black Box Testing	White Box Testing
It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
It is mostly done by software testers.	It is mostly done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred as outer or external software testing.	It is the inner or the internal software testing.
It is functional test of the software.	It is structural test of the software.
This testing can be initiated on the basis of requirement specifications document.	This type of testing of software is started after detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behavior testing of the software.	It is the logic testing of the software.
It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
It is also called closed testing.	It is also called as clear box testing.
It is least time consuming.	It is most time consuming.

It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example: search something on google by using keywords	**Example:** by input to check and verify loops
Types of Black Box Testing: • A. Functional Testing • B. Non-functional testing • C. Regression Testing	**Types of White Box Testing:** • A. Path Testing • B. Loop Testing • C. Condition testing

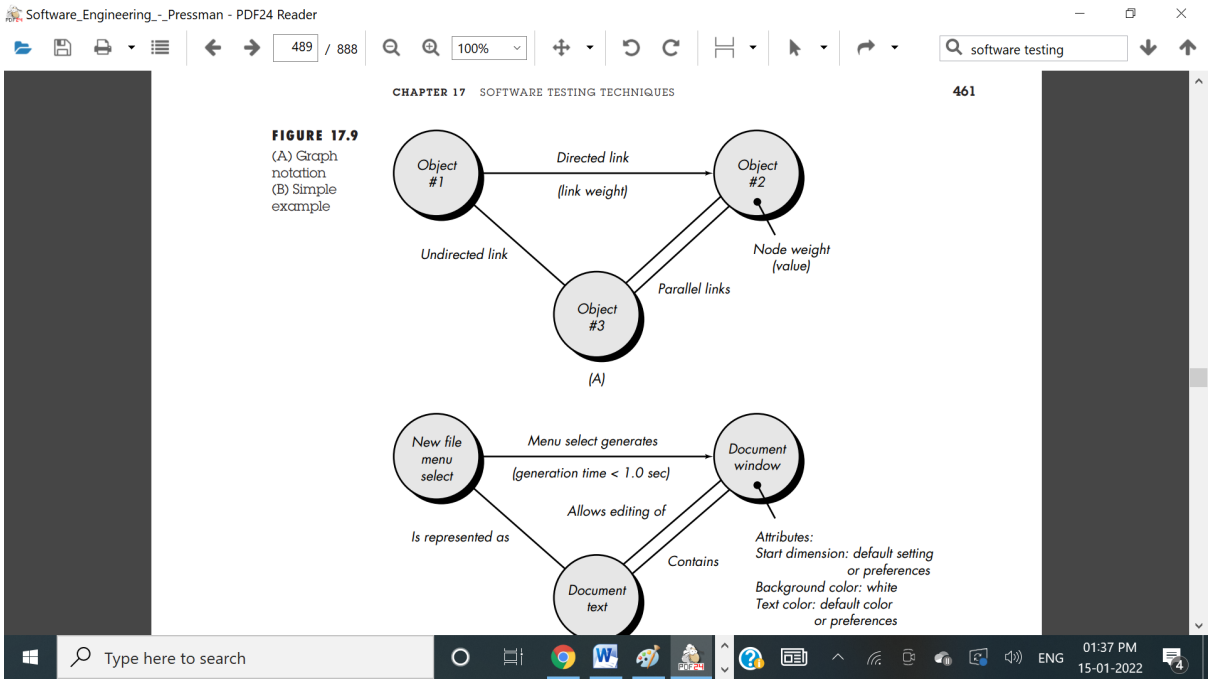
5.9. Graph-Based Testing Methods

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

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

To accomplish these steps, the software engineer begins by creating a graph—a

collection of nodes that represent objects; links that represent the relationships between objects; node weights that describe the properties of a node (e.g., a specific data value or state behavior); and link weights that describe some characteristic of a link.



5.9. Equivalence Partitioning:

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed. Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [BEI95]. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is

either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

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

5.11. Boundary Value Analysis:

For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that boundary value analysis (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values. Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

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

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering

analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4 . If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

5.12. Software Testing Strategies

A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in the successful construction of software. The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation. Software testing strategies provide the software developer with a template for testing and all have the following generic characteristics:

• Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.

• Different testing techniques are appropriate at different points in time.

• Testing is conducted by the developer of the software and an independent test group.

• Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

5.12.1. Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm [BOE81] states this another way:

Verification : "Are we building the product right?"

Validation : "Are we building the right product?"

Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing.

5.13. Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

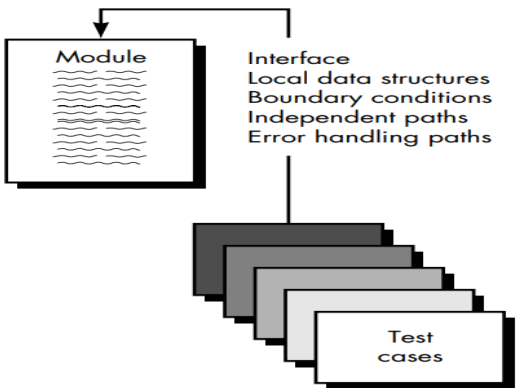


Fig: Unit Testing