PAPER SOLUTION

SOFTWARE ENGINEERING

T.E. I.T. SEM: VI (REV)

WINTER -2010

1. A] Explain process and project metrics. 05

Ans.:

Explanation: 05 marks

Software process and product metrics are quantitative measures that enable software people to gain insight into the efficiency of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred.

Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

The *IEEE Standard Glossary of Software Engineering Terms* [IEE93] defines *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors for each).
A software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per person- hour expended on reviews.

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.

In some cases, the same software metrics can be used to determine project and then process indicators. In fact, measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domain.

B] What are the projects best suited for agile methodology & Why? 05
Ans:

C] Identify the risks in web based application development. 05

Ans: Explanation 05 marks

Web-based systems and applications (WebApps) deliver a complex array of content and functionality to a broad population of end-users. Web engineering is the process used to create high-quality WebApps.

Web applications must serve (and adapt to) the needs of more than the gardener. The following WebApp characteristics drive the process:

Immediacy.

Web-based applications have immediacy [NOR99] that is not found in any other type of software. That is, the time to market for a complete Web site can be a matter of a few days or weeks.3 Developers must use methods for planning, analysis, design, implementation, and testing that have been adapted to the compressed time schedules required for WebApp development.

Security.

Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end-users who may access the application. In order to protect sensitive content and provide secure modes of data transmission, strong security measures must be implemented throughout the infrastructure that supports a WebApp and within the application itself.

Aesthetics.

An undeniable part of the appeal of a WebApp is its look and feel. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

D] When do you prototype? What are the outcomes of prototyping? 05
Ans: Explanation 05 marks

The prototype can serve as "the first system." In most projects, the first system built is barely usable. It may be too slow, too big, and awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved

The prototyping paradigm begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype.

The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done. Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly.

Yet, prototyping can also be problematic for the following reasons:

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

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.

2. A] Explain the steps in requirement engineering. 10

Ans: Explanation of each step carries 02 marks

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system.

The requirements engineering process can be described in five distinct steps [SOM97]:
• requirements elicitation
• requirements analysis and negotiation
• requirements specification
• system modeling
• requirements validation
• requirements management

1. **Requirements Elicitation:**

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.

why requirements elicitation is difficult:
***Problems of scope*.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
• ***Problems of understanding***. The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untreatable.
• ***Problems of volatility.*** The requirements change over time. To help overcome these problems, system engineers must approach the requirements gathering activity in an organized manner.

Somerville and Sawyer [SOM97] suggest a set of detailed guidelines for requirements elicitation, which are summarized in the following steps:

• Assess the business and technical feasibility for the proposed system.
• Identify the people who will help specify requirements and understand their organizational bias.
• Define the technical environment (e.g., computing architecture, operating system, telecommunications needs) into which the system or product will be placed.

2. **Requirements Analysis and Negotiation:**

Once requirements have been gathered, the work products noted earlier form the basis for *requirements analysis.* Analysis categorizes requirements and organizes them into related subsets; explores each requirement in relationship to others; examines requirements for consistency, omissions, and ambiguity; and ranks requirements based on the needs of customers/users. As the requirements analysis activity commences, the following questions are asked and answered:
• Is each requirement consistent with the overall objective for the system/product?
• Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
• Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
• Is each requirement bounded and unambiguous? Does each requirement have ttribution? That is, is a source (generally, a specific individual) noted for each requirement?
• Do any requirements conflict with other requirements?

3. **Requirements Specification:**

In the context of computer-based systems (and software), the term *specification* means different things to different people. A specification can be a written document, a raphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
The *System Specification* is the final work product produced by the system and requirements engineer. It serves as the foundation for hardware engineering, software engineering, database engineering, and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element. The *System Specification* also describes the information (data and control) that is input to and output from the system.

4.Requirements Validation:

Requirements validation examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the formal technical review
The review team includes system engineers, customers, users, and other stakeholders who examine the system specification looking for errors in content or interpretation.

The following questions represent a small subset of those that might be asked:
• Are requirements stated clearly? Can they be misinterpreted?

• Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
• Is the requirement bounded in quantitative terms?
• What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
• Does the requirement violate any domain constraints?

5.Requirements Management:

Requirements management begins with identification. Each requirement is assigned a unique identifier that might take the form <requirement type><requirement #> where requirement type takes on values such as *F* = functional requirement, *D* = data requirement, *B* = behavioral requirement, *I* = interface requirement, and *P* = output requirement. Hence, a requirement identified as F09 indicates a functional requirement assigned requirement number 9.

Once requirements have been identified, traceability tables are developed.
Among many possible traceability tables are the following:
Features traceability table. Shows how requirements relate to important customer observable system/product features.
Source traceability table. Identifies the source of each requirement.
Dependency traceability table. Indicates how requirements are related to one another.
Subsystem traceability table. Categorizes requirements by the subsystem(s) that they govern.
Interface traceability table. Shows how requirements relate to both internal and external system interfaces.

B] What do you understand by process maturity? Mention the activities in CMM level 4 & 5. 10

Ans: Explanation of process maturity carries 02 marks
Explanation of each CMM Level carries 04 marks

The Software Engineering Institute (SEI) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity.

To determine an organization's current state of process maturity, the SEI uses an assessment that results in a five point grading scheme. The grading scheme determines compliance with a ***capability maturity model*** **(CMM)** that defines key activities required at different levels of process maturity.
The SEI approach provides a measure of the global effectiveness of a company's software engineering practices and establishes five process maturity levels that are defined in the following manner:

Level 1: Initial.
The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
Level 2: Repeatable.
Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
Level 3: Defined.
The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide software process.
Level 4: Managed.
Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.
Level 5: Optimizing.
Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

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

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

Each KPA is described by identifying the following characteristics:

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

Process maturity level 4:

- Software quality management
- Quantitative process management

Process maturity level 5:

- Process change management
- Technology change management
- Defect prevention

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

3. A] How are efforts estimated? In which phase of development cycle effort estimation
done? 10

B] What are the risks associated with delayed with projects? How do project managers
manage such risks? 10

4. A] State the 5 major tasks in SCM. How is version control done? 10

Ans: State SCM tasks 02 marks
Explanation of version control carries 08 marks

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

The five important SCM tasks are as follows:

1. *identification,*
2. *version control,*
3. *change control,*
4. *configuration auditing,* and
5. r*eporting.*

VERSION CONTROL:
Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process.
Clemm [CLE89] describes version control in the context of SCM:
Configuration management allows a user to specify alternative configurations of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified [and constructed] by describing the set of desired attributes.

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

One representation of the different versions of a system is the evolution graph presented in Figure 9.3. Each node on the graph is an aggregate object, that is, a complete version of the software. Each version of the software is a collection of SCIs (source code, documents, data), and each version may be composed of different variants.

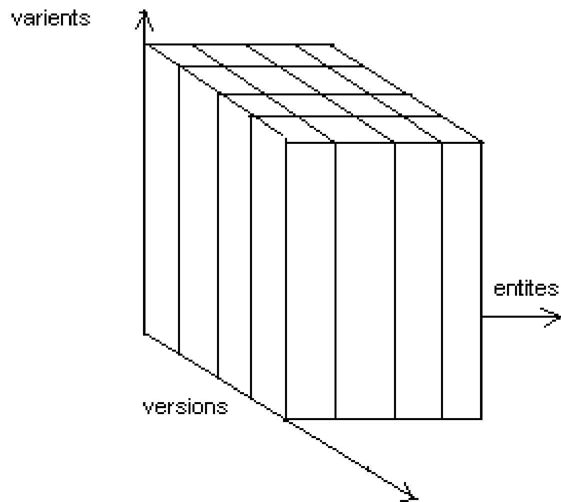

Fig. Object pool representation of components, variants, and versions

To illustrate this concept, consider a version of a simple program that is composed of entities 1, 2, 3, 4, and 5.3 Entity 4 is used only when the software is implemented using color displays. Entity 5 is implemented when monochrome displays are available. Therefore, two variants of the version can be defined: (1) entities 1, 2, 3, and 4; (2) entities 1, 2, 3, and 5.

To construct the appropriate *variant* of a given version of a program, each entity can be assigned an "attribute-tuple"—a list of features that will define whether the entity should be used when a particular variant of a software version is to be constructed.
One or more attributes is assigned for each variant. For example, a color attribute could be used to define which entity should be included when color displays are to be supported.

Another way to conceptualize the relationship between entities, variants and versions (revisions) is to represent them as an *object pool* [REI89]. Referring to Figure the relationship between configuration objects and entities, variants and versions can be represented in a three-dimensional space. An entity is composed of a collection of objects at the same revision level. A variant is a different collection of objects at the same revision level and therefore coexists in parallel with other variants.

A new version is defined when major changes are made to one or more objects.
A number of different automated approaches to version control have been proposed over the past decade.
The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

B] Describe the activities done during FTR, Configuration Audit and Status Reporting. **10**

Ans: FTR activities carries 04 marks.
Configuration audit & status Reporting activities carries 03 marks each.

FORMAL TECHNICAL REVIEWS:

A formal technical review is a software quality assurance activity performed by software engineers (and others).

The objectives of the FTR are

(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verify that the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards;
(4) to achieve software that is developed in a uniform manner; and
(5) to make projects more manageable.

In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

The FTR is actually a class of reviews that includes walkthroughs, inspections, round-robin reviews and other small group technical assessments of software. Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended. In the sections that follow, guidelines similar to those for a walkthrough [FRE90], [GIL93] are presented as a representative formal technical review.

The Review Meeting:

Regardless of the FTR format that is chosen, every review meeting should abide by the following constraints:

• Between three and five people (typically) should be involved in the review.
• Advance preparation should occur but should require no more than two hours of work for each person.
• The duration of the review meeting should be less than two hours.

Given these constraints, it should be obvious that an FTR focuses on a specific (and small) part of the overall software. For example, rather than attempting to review an entire design, walkthroughs are conducted for each component or small group of components. By narrowing focus, the FTR has a higher likelihood of uncovering errors. The focus of the FTR is on a work product (e.g., a portion of a requirements pecification, a detailed component design, a source code listing for a component). The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required. The project leader contacts a *review*

leader, who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation. Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work. Concurrently, the review leader also reviews the product and establishes an agenda for the review meeting, which is typically scheduled for the next day.

The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the *recorder;* that is, the individual who records (in writing) all important issues raised during the review. The FTR begins with an introduction of the agenda and a brief introduction by the producer. The producer then proceeds to "walk through" the work product, explaining the material, while reviewers raise issues based on their advance preparation. When valid problems or errors are discovered, the recorder notes each. 8.5.2 Review Reporting and Record Keeping During the FTR, a reviewer (the recorder) actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed.

A *review summary report* answers three questions:
1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The review summary report is a single page form (with possible attachments). It becomes part of the project historical record and may be distributed to the project leader and other interested parties.

The *review issues list* serves two purposes:
(1) to identify problem areas within the product and
(2) to serve as an action item checklist that guides the producer as corrections are made.
An issues list is normally attached to the summary report.
It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected. Unless this is done, it is possible that issues raised can "fall between the cracks." One approach is to assign the responsibility for follow up to the review leader.

Review Guidelines:

Guidelines for the conduct of formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed. A review that is uncontrolled can often be worse that no review at all.
The following represents a minimum set of guidelines for formal technical reviews:

1. *Review the product, not the producer.* An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment. Conducted improperly, the FTR can take on the aura of an inquisition. Errors should be

pointed out gently; the tone of the meeting should be loose and constructive; the intent should not be to embarrass or belittle. The review leader should conduct the review meeting to ensure that the proper tone and attitude are maintained and should immediately halt a review that has gotten out of control.

2. *Set an agenda and maintain it.* One of the key maladies of meetings of all types is *drift.* An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.

3. *Limit debate and rebuttal.* When an issue is raised by a reviewer, there may not be universal agreement on its impact. Rather than spending time debating the question, the issue should be recorded for further discussion off-line.

4. *Enunciate problem areas, but don't attempt to solve every problem noted.* A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.

5. *Take written notes.* It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.

6. *Limit the number of participants and insist upon advance preparation.* Two heads are better than one, but 14 are not necessarily better than 4. Keep the number of people involved to the necessary minimum. However, all review team members must prepare in advance. Written comments should be solicited by the review leader (providing an indication that the reviewer has reviewed the material).

7. *Develop a checklist for each product that is likely to be reviewed.* A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even test documents.

8. *Allocate resources and schedule time for FTRs.* For reviews to be effective, they should be scheduled as a task during the software engineering process. In addition, time should be scheduled for the inevitable modifications that will occur as the result of an FTR.

9. *Conduct meaningful training for all reviewers.* To be effective all review participants should receive some formal training. The training should stress both process-related issues and the human psychological side of reviews. Freedman and Weinberg [FRE90] estimate a one-month learning curve for every 20 people who are to participate effectively in reviews.

10. *Review your early reviews.* Debriefing can be beneficial in uncovering problems with the review process itself. The very first product to be reviewed should be the review guidelines themselves.

5. A] Explain cohesion & coupling and purpose of modular design. 10

Ans: Explaination of purpose of modular design 04 marks

Explanation of cohesion & coupling carries 03 marks each.

PURPOSE OF MODULAR DESIGN:

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

1.Functional Independence:

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding. In 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.
To summarize, functional independence is a key to good design, and design is the key to software quality.

2. 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.
Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable.
The scale for cohesion is nonlinear. low levels of cohesion should be avoided when modules are designed.

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

The module is called when computed data exceed prespecified bounds.
It performs the following tasks:
(1) computes supplementary data based on original computed data,
(2) produces an error report (with graphical content) on the user's workstation,
(3) performs follow-up calculations requested by the user,
(4) updates a database, and
(5) enables menu selection for subsequent processing.

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

3.Coupling:

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

low coupling (called *data coupling*) is exhibited in this portion of structure. A variation of data coupling, called *stamp coupling,* is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules *b* and *a*.

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

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

B] Relate the data flow and control flow diagram with an example case study of your choice. 10

6. A] What are the advantages of Test-driven development? 05

6. B] State software quality factors. **05**

Ans: Explanation of software quality factors 05 marks

Following are the software quality factors:

Quality:

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

Quality Assurance:

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

Cost of Quality:

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities. Cost of quality studies are conducted to provide a baseline for the current cost of quality, identify opportunities for reducing the cost of quality, and provide a normalized basis of comparison. The basis of normalization is almost always dollars. Once we have normalized quality costs on a dollar basis, we have the necessary data to evaluate where the opportunities lie to improve our processes. Furthermore, we can evaluate the effect of changes in dollar-based terms.

software quality:

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

C] Describe the different techniques in white box testing. 10

Ans: Explanations of each technique carries 05 marks.

WHITE-BOX TESTING:

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

The different White Box Testings are as follows:

A] BASIS PATH TESTING:

The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

1. Flow Graph Notation:

Before the basis path method can be introduced, a simple notation for the representation
of control flow, called a *flow graph* (or *program graph*) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 17.1. Each structured construct has a corresponding flow graph symbol.

2 .Cyclomatic Complexity:

Cyclomatic complexity is a software metric that provides a quantitative measure of the
logical complexity of a program.
When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with

an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An *independent path* is any path through the program that introduces at least one
new set of processing statements or a new condition.

3. Deriving Test Cases:

The basis path testing method can be applied to a procedural design or to source
code. In this section, we present basis path testing as a series of steps. The procedure *average,* depicted in PDL in Figure 17.4, will be used as an example to illustrate each step in the test case design method. Note that *average,* although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

4. Graph Matrices:

A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

B] CONTROL STRUCTURE TESTING:

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

1. Condition Testing:

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational
expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E1 <relational-operator> *E2*

where *E1* and *E2* are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥. A *compound condition* is composed of two
or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬).
A condition without relational expressions is referred to as a *Boolean expression.*

2. Data Flow Testing:

The *data flow testing* method selects test paths of a program according to the locations
of definitions and uses of variables in the program.

3. Loop Testing:

Loops are the cornerstone for the vast majority of all algorithms implemented in oftware.
And yet, we often pay them little heed while conducting software tests.
Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

7. Write short notes on any two: 2X10

A] Security Engineering.

7. B] Project scheduling and tracking.

Software project scheduling:

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks.

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

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.
Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.
Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.
Effort validation. Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been scheduled at any given time. For example, consider a project that has three assigned staff members (e.g., 3 person-days are available per day of assigned effort5). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person days of effort. More effort has been allocated than there are people to do the work.
Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.
Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a module) or a part of a work product. Work products are often combined in deliverables.
Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality (Chapter 8) and has been approved.
Each of these principles is applied as the project schedule evolves.

SCHEDULING:

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification to software projects.

Program evaluation and review technique (PERT) and *critical path method* (CPM)

[MOD83] are two project scheduling methods that can be applied to software development.
Both techniques are driven by information already developed in earlier project planning activities:

- Estimates of effort
- A decomposition of the product function
- The selection of the appropriate process model and task set
- Decomposition of tasks

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.
Both PERT and CPM provide quantitative tools that allow the software planner to
(1) determine the *critical path*—the chain of tasks that determines the duration of the project;
(2) establish “most likely” time estimates for individual tasks by applying statistical models; and
(3) calculate “boundary times” that define a time "window" for a particular task.

Boundary time calculations can be very useful in software project scheduling. Slippage in the design of one function, for example, can retard further development of other functions. Riggs [RIG81] describes important boundary times that may be discerned from a PERT or CPM network:
(1) the earliest time that a task can begin when all preceding tasks are completed in the shortest possible time,
(2) the latest time for task initiation before the minimum project completion time is delayed,
(3) the earliest finish—the sum of the earliest start and the task duration,
(4) the latest finish the latest start time added to task duration, and
(5) the *total float*—the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule. Boundary time calculations lead to a determination of critical path and provide the manager with a quantitative method for evaluating progress as tasks are completed.
Both PERT and CPM have been implemented in a wide variety of automated tools that are available for the personal computer [THE93]. Such tools are easy to use and make the scheduling methods described previously available to every software project manager.

1. **Timeline Charts:**

When creating a software project schedule, the planner begins with a set of tasks (the work breakdown structure). If automated tools are used, the work breakdown is input as a task network or task outline. Effort, duration, and start date are then input for each task. In addition, tasks may be assigned to specific individuals.
As a consequence of this input, a *timeline chart,* also called a *Gantt chart,* is generated.

A timeline chart can be developed for the entire project. Alternatively, separate charts can be developed for each project function or for each individual working on the project.
It depicts a part of a software project schedule that emphasizes the concept scoping task for a new word-processing (WP) software product. All project tasks (for concept scoping) are listed in the left-hand column. The horizontal bars indicate the duration of each task. When multiple bars occur at the same time on the calendar, task concurrency is implied. The diamonds indicate milestones.
Once the information necessary for the generation of a timeline chart has been input, the majority of software project scheduling tools produce *project tables*—a tabular listing of all project tasks, their planned and actual start- and end-dates, and a variety of related information. Used in conjunction with the timeline chart, project tables enable the project manager to track progress.

2. Tracking the Schedule:

The project schedule provides a road map for a software project manager. If it has been properly developed, the project schedule defines the tasks and milestones that must be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

• Conducting periodic project status meetings in which each team member reports progress and problems.
• Evaluating the results of all reviews conducted throughout the software engineering process.
• Determining whether formal project milestones been accomplished by the scheduled date.
• Comparing actual start-date to planned start-date for each project task listed in the resource table.
• Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
• Using earned value analysis (Section 7.8) to assess progress quantitatively.

In reality, all of these tracking techniques are used by experienced project managers.
Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff. If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems occur, the project manager must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed,10 additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.
When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called *time-boxing* [ZAH95]. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore, an incremental software paradigm is chosen and a schedule is derived for each incremental delivery.

The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A “box” is put around each task. When a task hits the boundary of its time box (plus or minus 10 percent), work stops and the next task begins.

The initial reaction to the time-boxing approach is often negative: “If the work isn’t finished, how can we proceed?” The answer lies in the way work is accomplished.
By the time the time-box boundary is encountered, it is likely that 90 percent of the task has been completed.11 The remaining 10 percent, although important, can (1) be delayed until the next increment or (2) be completed later if required. Rather than becoming “stuck” on a task, the project proceeds toward the delivery date.

C] System testing.

Ans: Explanation of system testing carries 02 marks
Explanation of each type carries 02 marks

SYSTEM TESTING:

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests that are worthwhile for software-based systems.

1.Recovery Testing:
A system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

2 Security Testing:
Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer [BEI84]: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

3 Stress Testing:

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

For example,
(1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
(2) input data rates may be increased by an order of magnitude to determine how input functions will respond,
(3) test cases that require maximum memory or other resources are executed,
(4) test cases that may cause thrashing in a virtual operating system are designed,
(5) test cases that may cause excessive hunting for disk-resident data are created.

A variation of stress testing is a technique called *sensitivity testing.* In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation.

Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

4. Performance Testing:

Performance testing is designed to test the run-time performance of software within the context of an integrated system.

Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion.

External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.