III/IV B.Tech (Regular) DEGREE EXAMINATION

April, 2018 **Information Technology**

Sixth Semester **Software Engineering**

Time: Three hours Maximum: 60 Marks

Answer question No.1 compulsorily. (1*12=12 Marks)

Answer one question from each unit (4*12=48 Marks)

a) Discuss about Agile process.

Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

b) Explain any one characteristic of software.

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

c) What is CASE?

CASE stands for Computer Aided Software Engineering. It means, development and maintenance of software projects with help of various automated software tools.

d) What is Requirement analysis?

Requirements analysis, also called requirements engineering, is the process of determining user expectations for a new or modified product. These features, called requirements, must be quantifiable, relevant and detailed. In software engineering, such requirements are often called functional specifications.

e) Write *any two* planning practices in software engineering practice.

- Understand the scope of project
- Involve the customer in the planning activity
- Recognize that planning is iterative
- Estimate based on what you know
- Consider risk as you define the plan
- Be realistic
- Adjust granularity as you define the plan

f) Write *any two* Deployment practices in software engineering practice.

- Customer Expectations for the software must be managed.
- A complete delivery package should be assembled and tested.
- A support regime must be established before the software is delivered.
- Appropriate instructional materials must be provided to end-users.
- Buggy software should be fixed first, delivered later.

g) Define Design Engineering.

Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints. Software design may refer to either "all the activity involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems"

h) What is an Architectural design?

Software architecture refers to the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system. Each structure

comprises software elements, relations among them, and properties of both elements and relations.

i) **Define Component level design.**
The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design. Component-level design occurs after the data and architectural designs are established.

j) **What are project domains?**
A domain is a field of study that defines a set of common requirements, terminology, and functionality for any software program constructed to solve a problem in the area of computer programming, known as domain engineering.

k) **What is SQA plan?**
Software quality assurance (SQA) consists of a means of monitoring the software engineering processes and methods used to ensure quality

l) **What is system testing?**
System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

UNIT-I

2. a) Explain the software myths.

Three types of Myths and its reality explanation 3*2=6M

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

Myth: We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?
Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."
Myth: 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. "adding people to a late software project makes it “In the absence of meaningful standards, a new industry like software comes to depend instead on folklore.” Tom DeMarco later." At first, this statement may seem counterintuitive.
Myth: If I decide to outsource3 the software project to a third party, I can just relax and let that firm build it.
Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

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

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behaviour, 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. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established.

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."

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

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

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

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

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

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

b) What type of changes is made to legacy system if it exhibits poor quality?

Legacy system---2M Types of changes ---4M

In computing, a legacy system is an old method, technology, computer system, or application program, "of, relating to, or being a previous or outdated computer system." Often a pejorative term, referencing a system as "legacy" means that it paved the way for the standards that would follow it.

Types of changes

- Hardware - may be obsolete mainframe hardware.
- Support software - may rely on support software from suppliers who are no longer in business.
- Application software - may be written in obsolete programming languages.
- Application data - often incomplete and inconsistent.
- Business processes - may be constrained by software structure and functionality.
- Business policies and rules - may be implicit and embedded in the system software.

3. a) What is CMMI? Discuss how various maturity levels of CMMI can be measured.

Six levels each one mark 6*1=6M

Level 0 : Incomplete: The process are is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability.

Level 1 : Performed: All the specific goals has been achieved and work tasks required to produce defined products has been conducted.

Level 2 : Managed: All level 1 criteria have been achieved. All work associated with the process area conforms to an organizationally defined policy, all people are doing their jobs, stakeholders are actively involved in the process area, all work tasks and work products are monitored, controlled and reviewed.

Level 3 : **Defined**: All level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines and contributes work products, measures and other process improvement information to the organizational process assets."

Level 4 : **Quantitatively Managed**: All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as a criteria in managing the process."

Level 5 : **Optimized**: All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The staged CMMI model defines the same process areas, goals and practices as the continuous model. The primary difference is that the staged model defines five maturity levels rather than five capability levels.

b) Explain in detail the process patterns. Give few examples of it.

Process patterns ----4M example-----2M

PROCESS PATTERNS

The software process can be defined as a collection of patterns that define a set of activities, actions, work tasks, work products and/or related behaviours required to develop computer software.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a complete process (e.g., prototyping). In other situations, patterns can be used to describe an important framework activity (e.g., planning) or a task within framework activity (e.g., project- estimating). Ambler has proposed the following template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name that describes its functions within the software process (e.g., customer-communication).

Intent. The objective of the pattern is described briefly. For example, the intent of customer-communication is "to establish a collaborative relationship with the customer in an effort to define project scope, business requirements, and other projects constraints".

Type. The pattern type is specified. Ambler suggests three types:

• Task pattern define a software engineering action or work task that is part of the process and relevant to successful software engineering practice (e.g., requirements gathering is a task pattern).

• Stage patterns represent a framework activity for the process. Since a framework activity encompasses multiple work tasks, a stage pattern incorporates multiple task patterns that are relevant to the stage. An example of a stage pattern might be communication. This pattern would incorporate the task pattern requirements gathering and others.

• Phase patterns define the sequence of framework activities is iterative in nature. An example of a phase pattern might be a spiral model or prototyping.

Initial context. The conditions under which the pattern applies are described. Prior to the initiation of the pattern, we ask (1) what organizational team created activities have already occurred (2) what is the entry state for the process? And (3) what software engineering information or project information already exists?

For example, the planning pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication: (2) successful completion of a number of task patterns (specified) for the customer-communication pattern has occurred: and (3) project scope, basic business requirements, and project constraints are known.

An Example Process Pattern:

The following abbreviated process pattern describes an approach that may be applicable when stakeholders have a general idea of what must be done, but are unsure of specific software requirements.

Pattern name. Prototyping.

Intent: The objective of the pattern is to build a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidity software requirements.

Type: Phase pattern.

Initial context: The following conditions must be met prior to the initiation of this pattern:(1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding problem to be solved has been identified by stakeholders: (4) an initial understanding of project scope, basic business requirements, and project constraints has been developed.

Problem: Requirements are hazy or nonexistent, yet there is clear recognition that there is a problem, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

Solutions: A description of the prototype that identifies basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process pattern.

Related Patterns: The following patterns are related to this pattern: customer communication; iterative design; iterative development, customer assessment; requirement extraction.

UNIT-II

4. a) Define Use case and explain their purpose.

Use case definition –2M Purpose-----4M

A **use case** is a methodology used in system analysis to identify, clarify, and organize system requirements. The use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal.

Purpose of Use case diagrams

The use case model is called as the requirements model; which also include a problem domain object model and user interface descriptions in this requirements model.

Use cases specify the functionality that the system will offer from the users' perspective. They are used to document the scope of the system and the developer's understanding of what it is that the users require.

Use cases are supported by *behavior specifications.* These specify the behavior of each use case either using UML diagrams, such as *collaboration diagrams* or *sequence diagrams* or in text form as *use case descriptions.*

Textual *use case descriptions* provide a description of the interaction between the users of the system called *actors,* and the high level functions within the system called use cases.

These descriptions can be in summary form or in a more detailed form in which the interaction between actor and use case is described in a step-by-step way. In all three representations, the use case describes the interaction as the user sees it, and is not a definition of the internal processes within the system, or some kind of program specification.

b) Analyse requirements elicitation and analysis process in detail.

Requirement elicitation--- 3M Analysis--3M

Requirement elicitation is the process of discovering, reviewing, documenting, and understanding the user's needs and constraints for the system.
Why do we need requirement elicitation?
There are some reasons according to Leffingwel, 2000:

1. Knowing what problems to be solved and recognizing system boundaries.
2. Identifying who are the stakeholders.
3. Recognizing the goal of system is the target to be achieved.

How?
You have to know the activities that involved in requirement elicitation is iterative process with feedback from another activities. The sum up of activities is requirement discovery, requirement classification and organization, requirement prioritization and negotiation, and requirement documentation.

Requirement analysis is the process of obtaining, classifying, and structuring the information carried by requirement engineer when trying to understand the whole parts of problem and relation. Another definition is the process of refining the user's needs and constraints.
Why?
There are some reasons below:
1. Processing the result of requirement elicitation to produce software requirement specification document

2. Developing detailed quality of attributes, where managers can make realistic project estimation and technical staff can continue their design, implementation, and testing.

3. Building understanding about characteristic of problem domain and requirements to find solution.

5. a) Discuss about modelling practices.

Analysis model----3M Design model ----3M

There are two classes of models are created:-
1) Analysis model
2) Design model

Analysis Model: – Represent the customer requirement, which includes algorithmic specifications of program or process. Analysis model includes three different domains a) Information domain b) Functional domain c) Behavioural domain.

Analysis Modelling principles:-

1) The information domain of problem must be clearly represented: – Analysis model uses "data flow diagram "to show information domain which includes following detail input flow into system, output flow in system, data collection by data store to collect data in system.
2) The function of software must be defined clearly:-function are process those transform input flow to output flow so specification must be clearly defined.
3) Behavior of system must be defined clearly:-Analysis model uses state transition diagram to represent the behavior of system clearly.
4) The clear hierarchy among information function and behavior must be shown: – information, function and behavior of system must be represented by using proper hierarchy which leads to easy design.
5) Analysis should be clear enough to convert it into design model: – it analysis of requirement is clear and simple then it will be easy for design.

Design Model:-

Design model provides concentrate specification for the construction of software. Design model includes a) Architectural design b) User interface design c) component level details.

Design Modelling principles:-

1) Design should be traceable from analysis model: – using elements if analysis model is constructed.
2) Consider a architecture of system to be built: – Architectural design shows following Architectural design style are as follow a) data concentrate Architecture b) data flow Architecture c) main / subprogram Architecture.
3) Data should be important rather than design of functions: – data design shows relationship between different data object shown by entity relationship between different data object shown by entity relationship diagram so data modeling is important.
4) Internal as well as External Interface must be designed: – For proper flow of data among system or external environment to system it is important to have all internal and external interface proper design is required.
5) User interface design must satisfy all needs of user:-Design must be simple and convenient to end user side.
6) Cohesion measure functionally strength of module so for proper design all function in single module should be highly cohesive so that why Component level design should be functionally independent.
7) Components required loosely coupled to the external environment: – The Component should be loosely coupled to one another and to external environment in the modeling system.
8) Design module should be easy to understand:-As simple modules are easy to test, debug and modify so design should be simple.
9) Accept that design modeling is iterative: – Good design should be capable to the absorb all changes.

b) Explain building the analysis model in requirements engineering.

Elements ----4M Rules of thumb----2M

Elements of the analysis model

1. Scenario based element

- This type of element represents the system user point of view.
- Scenario based elements are use case diagram, user stories.

2. Class based elements

- The object of this type of element manipulated by the system.
- It defines the object,attributes and relationship.
- The collaboration is occurring between the classes.
- Class based elements are the class diagram, collaboration diagram.

3. Behavioral elements

- Behavioral elements represent state of the system and how it is changed by the external events.
- The behavioral elements are sequenced diagram, state diagram.

4. Flow oriented elements

- An information flows through a computer-based system it gets transformed.
- It shows how the data objects are transformed while they flow between the various system functions.
- The flow elements are data flow diagram, control flow diagram.

Analysis Rules of Thumb

The rules of thumb that must be followed while creating the analysis model.

The rules are as follows:

- The model focuses on the requirements in the business domain. The level of abstraction must be high i.e there is no need to give details.
- Every element in the model helps in understanding the software requirement and focus on the information, function and behaviour of the system.
- The consideration of infrastructure and non-functional model delayed in the design. **For example,** the database is required for a system, but the classes, functions and behaviour of the database are not initially required. If these are initially considered then there is a delay in the designing.
- Throughout the system minimum coupling is required. The inter connections between the modules is known as 'coupling'.
- The analysis model gives value to all the people related to model.
- The model should be simple as possible. Because simple model always helps in easy understanding of the requirement.

UNIT-III

6. a) Analyse design concepts in design engineering.

Any Six concepts 6*1=6M

Design concepts

The set of fundamental software design concepts are as follows:

1. Abstraction

- A solution is stated in large terms using the language of the problem environment at the highest level abstraction.
- The lower level of abstraction provides a more detail description of the solution.
- A sequence of instruction that contain a specific and limited function refers in a procedural abstraction.
- A collection of data that describes a data object is a data abstraction.

2. Architecture

- The complete structure of the software is known as software architecture.
- Structure provides conceptual integrity for a system in a number of ways.
- The architecture is the structure of program modules where they interact with each other in a specialized way.
- The components use the structure of data.
- The aim of the software design is to obtain an architectural framework of a system.
- The more detailed design activities are conducted from the framework.

3. Patterns

A design pattern describes a design structure and that structure solves a particular design problem in a specified content.

4. Modularity

- A software is separately divided into name and addressable components. Sometime they are called as modules which integrate to satisfy the problem requirements.
- Modularity is the single attribute of a software that permits a program to be managed easily.

5. Information hiding

- Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules not requiring that information.

6. Functional independence

- The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.
- The functional independence is accessed using two criteria i.e Cohesion and coupling.

7. Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.

8. Refactoring

- It is a reorganization technique which simplifies the design of components without changing its function behaviour.
- Refactoring is the process of changing the software system in a way that it does not change the external behaviour of the code still improves its internal structure.

b) What is design process in design engineering?

Software quality guidelines---3M Quality attributes----3M

Design process

- The main aim of design engineering is to generate a model which shows firmness, delight and commodity.
- Software design is an iterative process through which requirements are translated into the blueprint for building the software.

Software quality guidelines

- A design is generated using the recognizable architectural styles and compose a good design characteristic of components and it is implemented in evolutionary manner for testing.
- A design of the software must be modular i.e the software must be logically partitioned into elements.
- In design, the representation of data , architecture, interface and components should be distinct.
- A design must carry appropriate data structure and recognizable data patterns.
- Design components must show the independent functional characteristic.
- A design creates an interface that reduces the complexity of connections between the components.
- A design must be derived using the repeatable method.
- The notations should be use in design which can effectively communicates its meaning.

Quality attributes

The attributes of design name as 'FURPS' are as follows:

Functionality:
It evaluates the feature set and capabilities of the program.
Usability:
It is accessed by considering the factors such as human factor, overall aesthetics, consistency and documentation.
Reliability:
It is evaluated by measuring parameters like frequency and security of failure, output result accuracy, the mean-time-to-failure(MTTF), recovery from failure and the the program predictability.

Performance:
It is measured by considering processing speed, response time, resource consumption, throughput and efficiency.
Supportability:

- It combines the ability to extend the program, adaptability, serviceability. These three term defines the maintainability.
- Testability, compatibility and configurability are the terms using which a system can be easily installed and found the problem easily.
- Supportability also consists of more attributes such as compatibility, extensibility, fault tolerance, modularity, reusability, robustness, security, portability, scalability.

7. a) Discuss about designing class based components.

Design guidelines---2M cohesion---2M coupling---2M

Component Level Design Guidelines
As component level design proceeds, there are some guidelines other than basic design guidelines. These guidelines are applied to components and their interfaces, their dependencies and inheritance.
Components
- Naming conventions should be established for components.
- These are specified as part of architectural model and then refined and elaborated as part of component level design.
- Architectural component names should be drawn from the problem domain.
- It should have meaning to all stakeholders.

Interfaces
- Provides information about communication and collaboration.
- Lollipop approach to represent interface should be used in combination with UML box and dashed arrow.
- Interface should flow from left side of the component box for consistency.
- Only relevant interfaces to component should be shown.

Dependencies And Inheritance
- Model dependencies from left to right.
- Model inheritance from bottom to top.
- Component interdependencies should be represented via interfaces.

Cohesion
Cohesion measures the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules. The tighter the elements are bound to each other, the higher will be the cohesion of a module. In practice, designers should avoid a low level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa.
Various types of cohesion are listed below.

- **Functional cohesion:** In this, the elements within the modules are concerned with the execution of a single function.
- **Sequential cohesion:** In this, the elements within the modules are involved in activities in such a way that the output from one activity becomes the input for the next activity.

- **Communicational cohesion:** In this, the elements within the modules perform different functions, yet each function references the same input or output information.
- **Procedural cohesion:** In this, the elements within the modules are involved in different and possibly unrelated activities.
- **Temporal cohesion:** In this, the elements within the modules contain unrelated activities that can be carried out at the same time.
- **Logical cohesion:** In this, the elements within the modules perform similar activities, which are executed from outside the module.
- **Coincidental cohesion:** In this, the elements within the modules perform activities with no meaningful relationship to one another.

Coupling

Coupling measures the degree of interdependence among the modules. Several factors like interface complexity, type of data that pass across the interface, type of communication, number of interfaces per module, etc. influence the strength of coupling between two modules. For better interface and well-structured system, the modules should be loosely coupled in order to minimize the 'ripple effect' in which modifications in one module results in errors in other modules. Module coupling is categorized into the following types.

- **Control coupling:** Two modules are said to be 'control coupled' if they communicate (pass a piece of information intended to control the internal logic) using at least one 'control flag'. The control flag is a variable whose value is used by the dependent modules to make decisions.
- **Content coupling:** Two modules are said to be 'content coupled' if one module modifies data of some other module or one module is under the control of another module or one module branches into the middle of another module.
- **Common coupling:** Two modules are said to be 'common coupled' if *they* both reference a common data block.

b) Explain the golden rules.

Three Rules each 2M 3*2=6M

Golden Rules

User Interface means how a software looks on a user's screen. It is obvious that people will have less interest in using a software with bad user interface. So, user interface of a software must be good or excellent. In order to do that, software engineers must follow some rules for user interface design called golden rules. There are three golden rules and theyare:

Place the User in Control:

User wants to control the computer but does not want to be controlled by the computer. So, user interface constraints specified by the designer must simplify the mode of user's interaction with the computer. User interface must not frustrate the user. Design principles that places the user in control are:

Interaction modes must be defined in a way that will never force a user to do unnecessary or undesired actions : User must not be forced to do operations which he/she feels unnecessary. For example, user must not be forced to do spell check operation for editing a word, letter etc,.

Provide flexible interaction : Different users choose different types of interactions like mouse, keyboard commands, voice etc,. User must be given the chance to choose. For example, drawing complex shapes via keyboard commands frustrates users but doing the same via voice is simple.User interaction must be interruptible and undoable by the user : User should be able to interrupt even when in a sequence of actions but without losing any progress. Also, any user action must be undoable. Define interaction based on skill levels and allow the interaction to be customized : Users may be normal people or computer experts. It is better to allow some macros to be created by expert users to replace the same set of instructions used many times.

Reduce the User's Memory Load

A software must not force a user to memorize anything. On the other hand, it should provide recall feature to provide data when it is needed by storing it in its memory. Design principles that reduces the user's memory load are:

Reduce the need for short term memory : Some complex tasks need users to memorize some recent data. User Interface must be designed such that visual cues are provided to the user which helps him/her in reusing previous results etc,.

Defaults must be meaningful : Suppose there are some properties like volume etc,. The default values of such properties must be meaningful i.e. generally used. But, user must be able to change the properties if he/she wants to. Also, user must be able to reset the properties to their default values.

Define intuitive shortcuts : There are many mnemonics like Ctrl + C to copy, Ctrl + V to paste, Ctrl + X to cut etc,. These shortcuts must be designed such that they are easy to remember i.e. Ctrl + C to copy is easy to remember as C is for Copy etc,.

Make the Interface Consistent

User Interface must display or acquire information in a consistent fashion. This includes organisation of visual information according to design rules common to all types of screens, defining and implementing mechanisms for navigation from task to task consistently etc,. Design principles that makes the interface consistent are:

User must be able to put the current task into a meaningful context : The interface must be in a way that enables the users to know the context of the current task. So, there should be some indicators which say from which task did the user come here and to which task he/she can go from here.

Maintain consistency across a family of applications : This rule suggests that a set of software applications should implement same design rules if possible so that interaction gets easier with time.

If previous models have created user expectations, do not make changes unless there is a good reason : Suppose a special feature is provided in a previous version of the software application and its popular among its users, do not try to remove it or change it. Hence, the golden rules for user interface design.

UNIT-IV

8. a) Explain metrics for maintenance?

Software maturity index formula---2M Explanation---4M

All of the software metrics introduced in this chapter can be used for the development of new software and the maintenance of existing software. However, metrics designed explicitly for maintenance activities have been proposed.

IEEE Std. 982.1-1988 [IEE94] suggests a ***software maturity index*** (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

MT = the number of modules in the current release

Fc = the number of modules in the current release that have been changed

Fa = the number of modules in the current release that have been added

Fd = the number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

SMI = [*MT* _ (*Fa* + *Fc* + *Fd*)]/*MT*

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI and empirical models for maintenance effort can be developed.

b) Analyse object oriented testing methods.

Any three test methods 3*2=6M

The Test Case Design Implications of OO Concepts

The OO class is the target for test case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing. As Binder notes, "Testing requires reporting on the concrete and abstract state of an object." Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

Inheritance also leads to additional challenges for the test case designer. We have already noted that each new context of usage requires retesting, even though reuse has been achieved. In addition, multiple inheritance complicates testing further by increasing the number of contexts for which testing is required. If subclasses instantiated from a superclass are used within the same problem domain, it is likely that the set of test cases derived for the superclass can be used when testing the subclass. However, if the subclass is used in an

entirely different context, the super class test cases will have little applicability and a new set of tests must be designed.

Applicability of Conventional Test Case Design Methods

The white-box testing methods can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested. However, the concise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods. Use-cases can provide useful input in the design of black-box and state-based tests .

Fault-based Testing

Fault-based testing is used to determine or uncover a set of plausible faults. In other words, the focus of tester in this testing is to detect the presence of possible faults. Fault-based testing starts by examining the analysis and design models of OO software as these models may provide an idea of problems in the implementation of software. With the knowledge of system under test and experience in the application domain, tester designs test cases where each test case targets to uncover some particular faults.

The effectiveness of this testing depends highly on tester experience in application domain and the system under test. This is because if he fails to perceive real faults in the system to be plausible, testing may leave many faults undetected. However, examining analysis and design models may enable tester to detect large number of errors with less effort. As testing only proves the existence and not the absence of errors, this testing approach is considered to be an effective method and hence is often used when security or safety of a system is to be tested.

Scenario-based Testing

Scenario-based testing is used to detect errors that are caused due to incorrect specifications and improper interactions among various segments of the software. Incorrect interactions often lead to incorrect outputs that can cause malfunctioning of some segments of the software. The use of scenarios in testing is a common way of describing how a user might accomplish a task or achieve a goal within a specific context or environment. Note that these scenarios are more context- and user specific instead of being product-specific. Generally, the structure of a scenario includes the following points.

- A condition under which the scenario runs.
- A goal to achieve, which can also be a name of the scenario.
- A set of steps of actions.
- An end condition at which the goal is achieved.
- A possible set of extensions written as scenario fragments.

Scenario- based testing combines all the classes that support a use-case (scenarios are subset of use-cases) and executes a test case to test them. Execution of all the test cases ensures that all methods in all the classes are executed at least once during testing. However, testing all the objects (present in the classes combined together) collectively is difficult. Thus, rather than testing all objects collectively, they are tested using either top-down or bottom-up integration approach.

9. a) Explain metrics for source code in product metrics.

Software metrics formula---2M Explanation---4M

Halstead's theory of software science [HAL77] is one of "the best known and most thoroughly studied . . . composite measures of (software) complexity" [CUR80]. Software science proposed the first analytical "laws" for computer software.

Software science assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. These follow:

$n1$ = the number of distinct operators that appear in a program.

$n2$ = the number of distinct operands that appear in a program.

$N1$ = the total number of operator occurrences.

$N2$ = the total number of operand occurrences.

Halstead uses these primitive measures to develop expressions for the overall *program length, potential minimum volume* for an algorithm, the *actual volume* (number of bits required to specify a program), the *program level* (a measure of software complexity), the *language level* (a constant for a given language), and other features such as development effort, development time, and even the projected number of faults in the software.

Halstead shows that length N can be estimated

$$\boldsymbol{N = n1 \log 2\ n1 + n2 \log 2\ n2}$$

and program volume may be defined

$$\boldsymbol{V = N \log 2\ (n1 + n2)}$$

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, L must always be less than 1.

In terms of primitive measures, the volume ratio may be expressed as

$$\boldsymbol{L = 2/n1\ \_\ n2/N2}$$

Halstead's work is amenable to experimental verification and a large body of research has been conducted to investigate software science.

b) Discuss test strategies for object oriented software.

Unit Testing in the OO Context---3M Integration Testing in the OO Context---3M

The classical strategy for testing computer software begins with "testing in the small" and works outward toward "testing in the large." Stated in the jargon of software testing, we begin with unit testing, then progress toward integration testing, and culminate with validation and system testing. In conventional applications, unit testing focuses on the smallest compilable program unit—the subprogram (e.g., module, subroutine, procedure, component). Once each of these units has been tested individually, it is integrated into a program structure while a series of regression tests are run to uncover errors due to interfacing between the modules and side effects caused by the addition of new units. Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as *methods* or *services*) that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class or object. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.

We can no longer test a single operation in isolation but rather as part of a class. To illustrate, consider a class hierarchy in which an operation *X* is defined for the super class and is inherited by a number of subclasses. Each subclass uses operation *X,* but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation *X* is used varies in subtle ways, it is necessary to test operation *X* in the context of each of the subclasses. This means that testing operation *X* in a vacuum is ineffective in the object-oriented context.

Integration Testing in the OO Context

Because object-oriented software does not have a hierarchical control structure, conventional top-down and bottom-up integration strategies have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class".

There are two different strategies for integration testing of OO systems.The first, *thread-based testing,* integrates the set of classes required to respond to one input or event for the system. Each *thread* is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed. Unlike conventional integration, the use of drivers and stubs as replacement operations is to be avoided, when possible.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

Scheme prepared by Signature of the HOD

Paper Evaluators:

S.No	Name of the College	Name of the Examiner	Signature