18IT501

III/IV B.Tech (Regular) DEGREE EXAMINATION

Feb, 2021 **Information Technology**
Fifth Semester **Software Engineering**
Time: Three hours Maximum: 50 Marks
Answer question No.1 compulsorily. (1*10=10 Marks)
Answer any four from part-B (4*10=40 Marks)

a) Define Legacy Software.
The older programs which were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms are called as legacy software.

b) What is a process pattern?
Process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem

c) What are Spike Solutions?
If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design called a spike solution.

d) Define Refactoring.
Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves the internal structure.

e) What are Exciting Requirements?
The features that go beyond the customer's expectations and prove to be very satisfying when present are called exciting Requirements. For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

f) Define Modularity.
Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

g) How the UML Swim lanes are used in the activity?
The UML swim lane diagram is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.

h) Differentiate Verification and validation
Validation is the process of checking whether the specification captures the customer's needs, while verification is the process of checking that the software meets the specification.

i) What is Integration Testing?
Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

j) Differentiate Black box and White box Testing.
Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the software being tested is not known to the tester. White Box Testing is a software testing method in which the internal structure/ design/ implementation of the software being tested is known to the tester.

Part-B

UNIT-I

2. a) Define Software. Explain about the considerable characteristics differences between software and hardware.

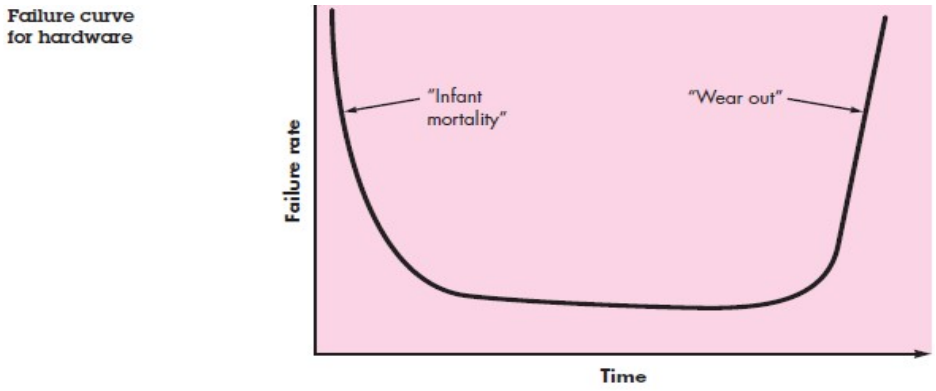
Software Definition---1M Characteristics----4M

Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Characteristics

Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

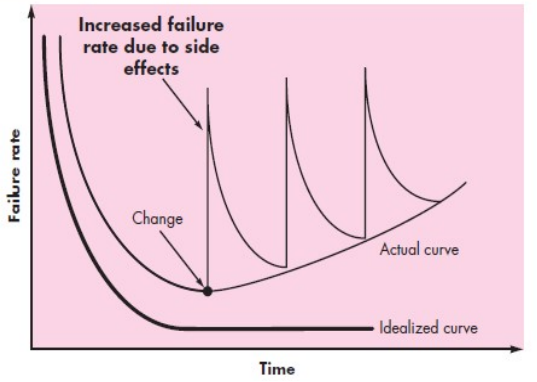


Software doesn't "wear out."

The figure depicts Failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life defects are corrected and the failure rate drops to a steady-state level for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out.*

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program.

FIGURE 1.2
Failure curves for software



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

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

b) What are the General Principles that focus on Software Engineering Practices as a whole?

Any Five Software Engineering Principles can be considered ---5M

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones.

The Third Principle: *Maintain the Vision*

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems.

The Fourth Principle: *What You Produce, Others Will Consume*

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able

to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing*. The audience for any product of software development is potentially large.

The Fifth Principle: *Be Open to the Future*

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*.

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

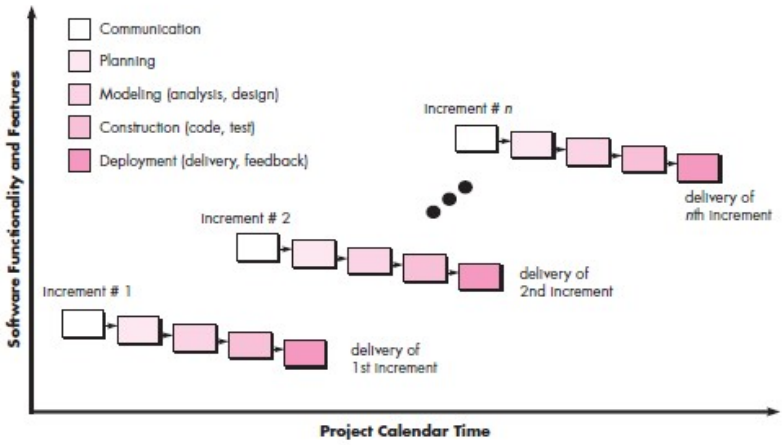
The Seventh principle: *Think!*

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience.

3. a) Explain with a neat diagram the Incremental Process Model

Diagram-1M Explanation---4M

FIGURE 2.5
The incremental model



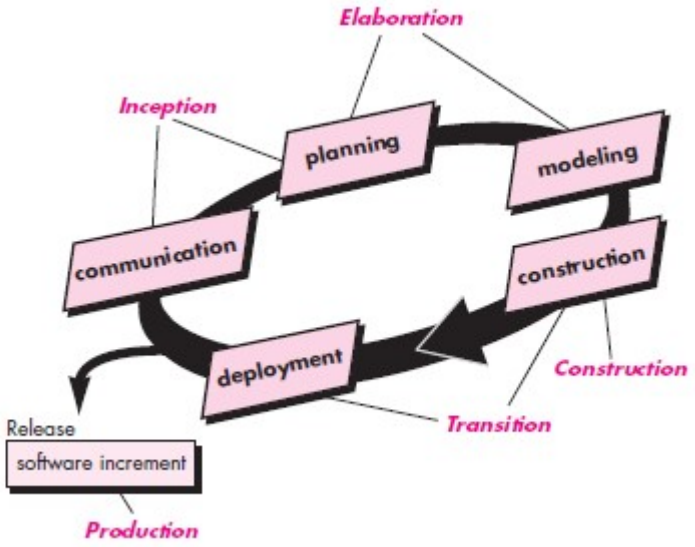
There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users

quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.
The *incremental* model combines elements of linear and parallel process flows discussed in Section 2.1. Referring to Figure 2.5, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software [McD93] in a manner that is similar to the increments produced by an evolutionary process flow.
For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.
The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

b) What are the phases of the Unified Process? Explain them.

Diagram—1M Explanation 4M

FIGURE 2.9
The Unified Process



The ***inception phase*** of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed.

Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires. Architecture at this point is nothing more than a tentative outline of major subsystems and the function and features that populate them. Later, the architecture will be refined and expanded into a set of models that will represent different views of the system. Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases that are to be applied as the software increment is developed.

The ***elaboration phase*** encompasses the communication and modeling activities of the generic process model (Figure 2.9). Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model. In some cases, elaboration creates an "executable architectural baseline" [Arl02] that represents a "first cut" executable system.

The ***construction phase*** of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users. To accomplish this, requirements and design models that were started during the elaboration phase are completed to reflect the final version of the software increment. All necessary and required features and functions for the software increment are then implemented in source code. As components are being implemented, unit tests are designed and executed for each.

The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing and user feedback reports both defects and necessary changes. In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release. At the conclusion of the transition phase, the software increment becomes a usable software release.

The ***production phase*** of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.

UNIT-II

4. a) Illustrate Adaptive Software Development as a technique for building complex systems.

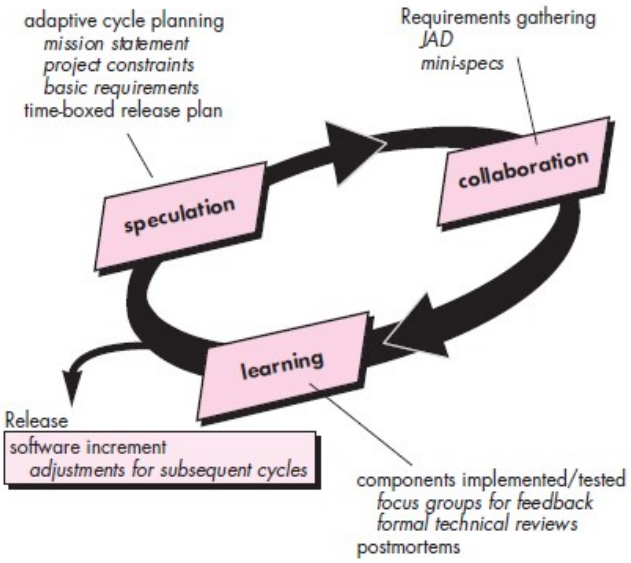
Diagram –1M Illustration-----4M

Adaptive Software Development (ASD) has been proposed by Jim High smith as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

High smith argues that an agile, adaptive development approach based on collaboration

is “as much a source of *order* in our complex interactions as discipline and engineering.” He defines an ASD “life cycle” (Figure 3.3) that incorporates three phases, speculation, collaboration, and learning.

FIGURE 3.3
Adaptive software development



During ***speculation***, the project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer’s mission statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project. No matter how complete and farsighted the cycle plan, it will invariably change. Based on information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality in which an ASD team is working.

Motivated people use ***collaboration*** in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on “***learning***” as much as it is on progress toward a completed cycle. In fact, argues that software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways: focus groups (Chapter 5), technical reviews and project post-mortems.

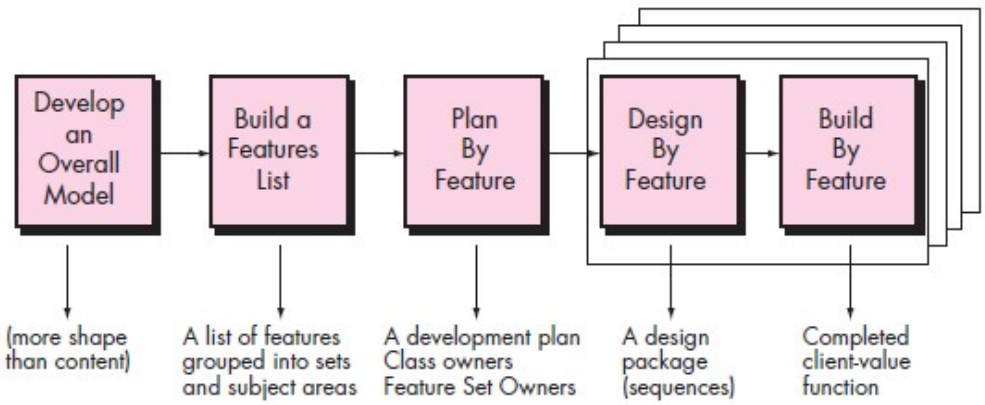
b) Write a Note on Feature Driven Development

Diagram—1M Explanation 4M

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues [Coa99] as a practical process model for object-oriented software engineering.
Stephen Palmer and John Felsing [Pal02] have extended and improved Coad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.
In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" [Coa99]. The emphasis on the definition of features provides the following benefits:

• Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
• Features can be organized into a hierarchical business-related grouping.
• Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
• Because features are small, their design and code representations are easier to inspect effectively.
• Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

FIGURE 3.5
Feature Driven Development [Coa99] (with permission)



For example: *Making a product sale* is a feature set that would encompass the features noted earlier and others.
The FDD approach defines five "collaborating" [Coa99] framework activities (in FDD these are called "processes") as shown in Figure 3.5. FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. As projects grow in size and complexity, ad hoc project management is often inadequate. It is essential for developers, their managers, and other stakeholders to understand.

5. a) Explain the Elements of Requirements Model.

Explanation ----5M

There are many different ways to look at the requirements for a computer-based system. Some software people argue that it's best to select one mode of representation (e.g., the use case) and apply it to the exclusion of all other modes. Other practitioners believe that it's worthwhile to use a number of different modes of representation to depict the requirements model. Different modes of representation force you to consider requirements from different viewpoints—an approach that has a higher probability of uncovering omissions, inconsistencies, and ambiguity.

Scenario-based elements. The system is described from the user's point of view using a scenario-based approach. For example, basic use cases (Section 5.4) and their corresponding use-case diagrams (Figure 5.2) evolve into more elaborate template-based use cases. Scenario-based elements of the requirements model are often the first part of the model that is developed. As such, they serve as input for the creation of other modeling elements. Figure 5.3 depicts a UML activity diagram17 for eliciting requirements and representing them using use cases. Three levels of elaboration are shown, culminating in a scenario-based representation.

FIGURE 5.3
UML activity diagrams for eliciting requirements

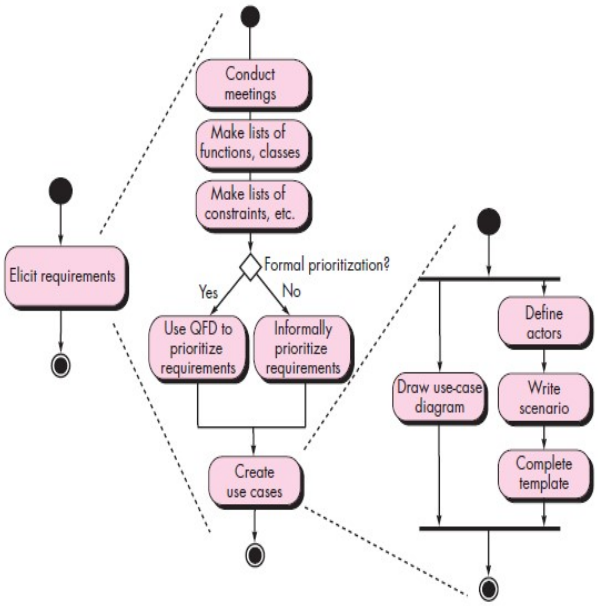
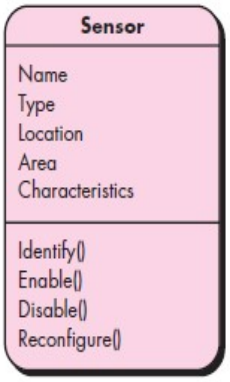


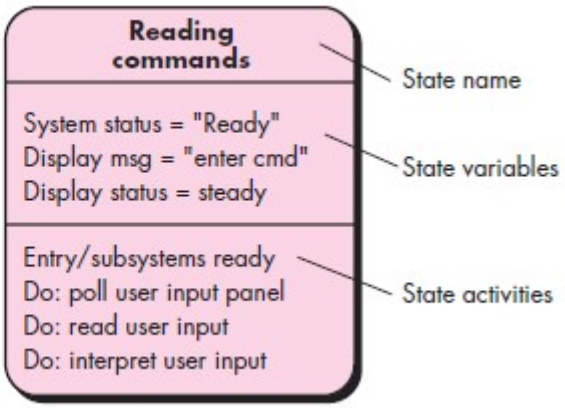
FIGURE 5.4
Class diagram for sensor



Class-based elements. Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of

things that have similar attributes andcommonbehaviors. For example, a UML class diagram can be used to depict a **Sensor** class for the *SafeHome* security function (Figure 5.4). Note that the diagram lists the attributes of sensors (e.g., name, type) and the operations (e.g., *identify, enable*) that can be applied to modify these attributes. In addition to class diagrams, other analysis modeling elements depict the manner in which classes collaborate with one another and the relationships and interactions between classes.

FIGURE 5.5
UML state diagram notation



Behavioral elements. The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.

Flow-oriented elements. Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. Input may be a control signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system.

b) What are the questions to be asked and answered to validate requirements?

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments. A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or

product?
• Is each requirement testable, once implemented?
• Does the requirements model properly reflect the information, function, and behavior of the system to be built?
• Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
• Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

UNIT-III

6. a) List and Explain the analysis rules of thumb that should be followed when creating the analysis model.

Any five rules 5*1=5M

Analysis Rules of Thumb

Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

• *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.* "Don't get bogged down in details" [Arl02] that try to explain how the system will work.

• *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*

• *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.

• *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.

• *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.

• *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

b) What are the quality guide lines we have in order to evaluate Quality in the design process.

Software quality guidelines---5M

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.

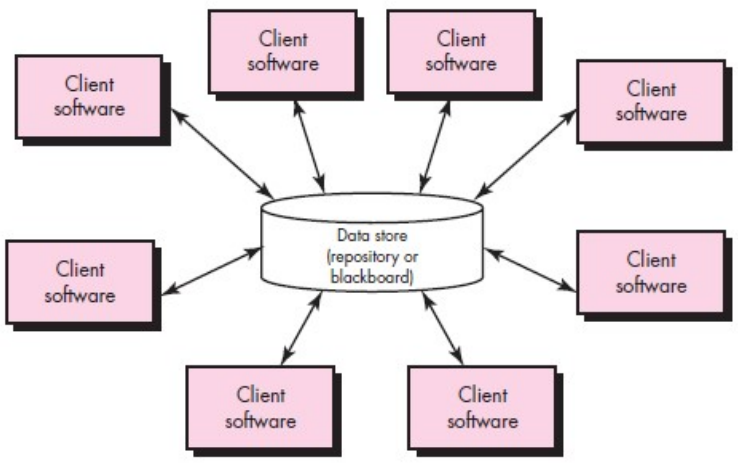
Software quality guidelines

- 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.

7. a) Explain the following Architectural Styles
Data Centered Architecture b) Data Flow Architecture
Data Centered Architecture------5M Data Flow Architecture-----5M

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client software accesses a central repository.
In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard"

FIGURE 9.1
Data-centered architecture



that sends notifications to client software when data of interest to the client changes.

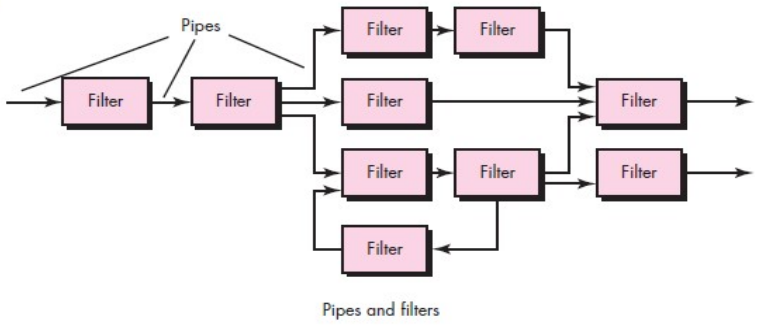
Data-centered architectures promote *integrability* [Bas03]. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed batch sequential.

This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

FIGURE 9.2
Data-flow architecture



UNIT-IV

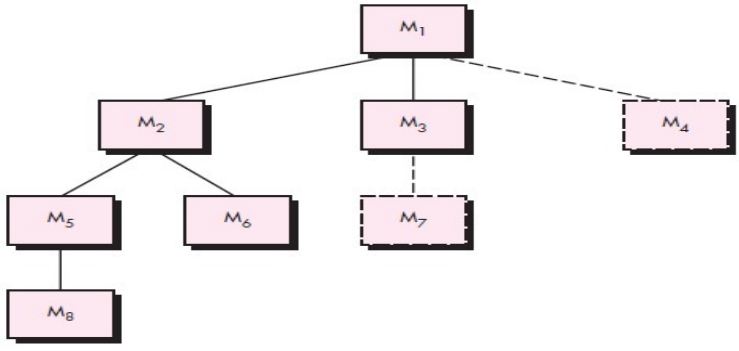
8. a) Explain the Integration Test Strategy for Conventional Software

Integration Test Strategy----5M

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

Top-down integration. *Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module

FIGURE 17.5
Top-down integration



main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. Referring to Figure 17.5, *depth-first integration* integrates all components on amajor control path of the program structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.

For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally.

From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

Bottom-up integration. *Bottom-up integration testing,* as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

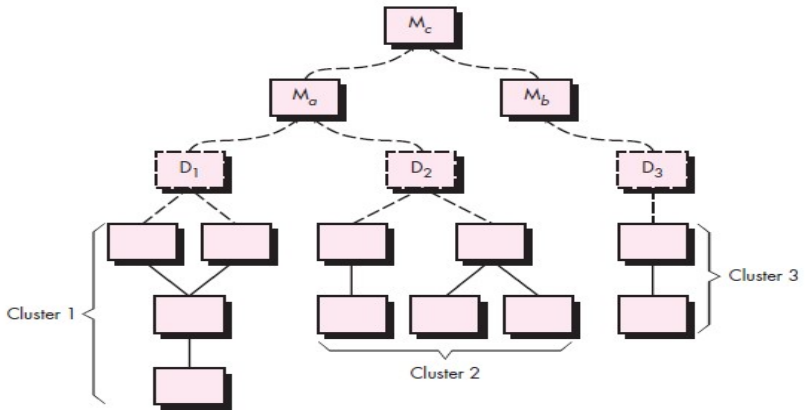
1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.

2. A *driver* (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

FIGURE 17.6
Bottom-up integration



Integration follows the pattern illustrated in Figure 17.6. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M*a*. Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a*. Similarly, driver D3 for cluster 3 is removed prior to integration with module M*b*. Both M*a* and M*b* will ultimately be integrated with component M*c*, and so forth.

b) Write a note on the debugging process.

Diagram---1M Process---4M

Debugging is not testing but often occurs as a consequence of testing.5 Referring to Figure 17.7, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the non corresponding data are a symptom of an underlying cause as yet hidden. The

debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes: (1) the cause will be found and corrected or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

A few characteristics of bugs provide some clues:

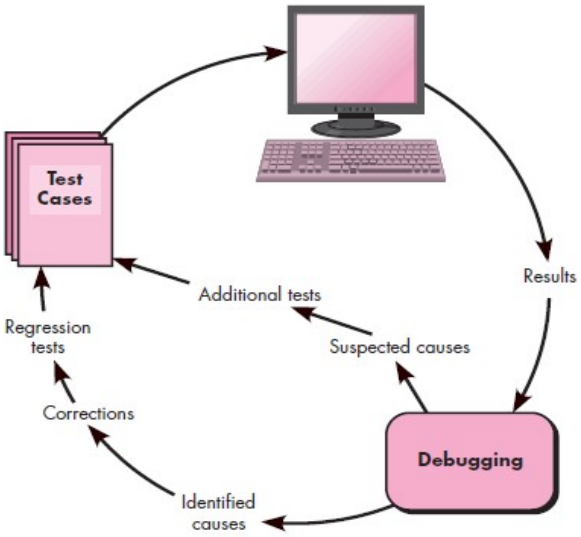
1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components exacerbate this situation.

2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).

4. The symptom may be caused by human error that is not easily traced.

FIGURE 17.7 The debugging process



5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

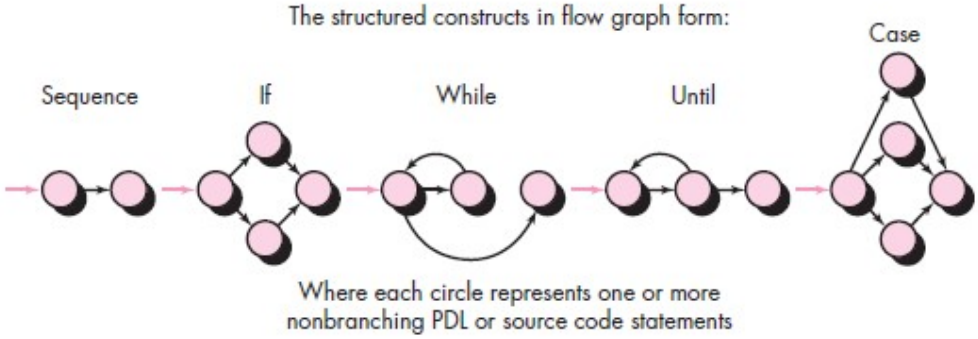
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

9.a) Explain Flow graph notation in Basis Path Testing.

Flow Graph Notation Explanation----4M Diagram---1M

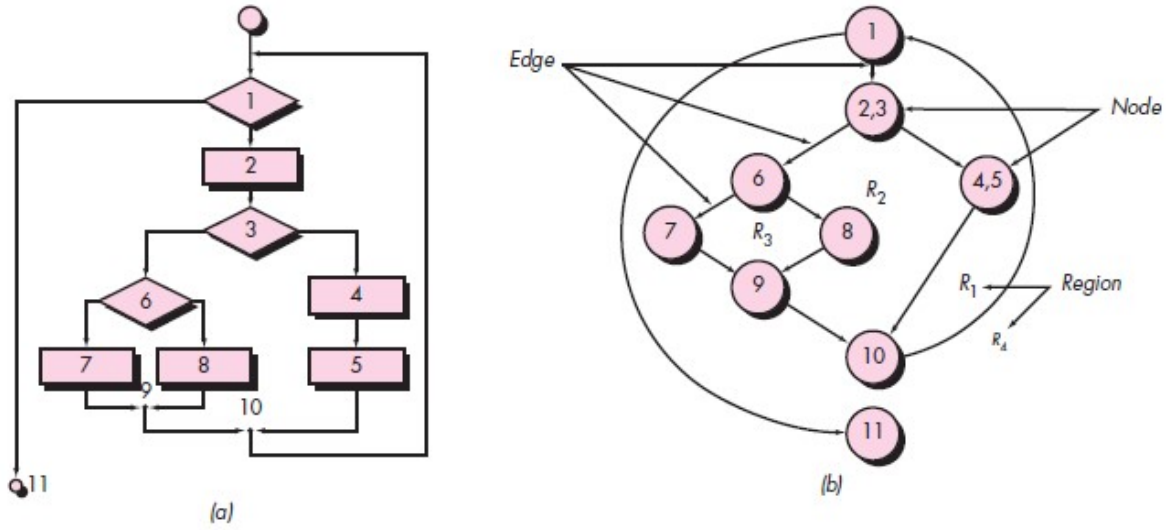
Before we consider the basis path method, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced.3 The flow graph depicts logical control flow using the notation illustrated in Figure 18.1. Each structured construct (Chapter 10) has a corresponding flow graph symbol.

FIGURE 18.1 Flow graph notation



To illustrate the use of a flow graph, consider the procedural design representation in Figure 18.2a. Here, a flowchart is used to depict program control structure. Figure 18.2b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 18.2b, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links,* represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region.

FIGURE 18.2 (a) Flowchart and (b) flow graph



b) Explain about Equivalence partitioning in Black box Testing.

Explanation about Equivalence partitioning----5M

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed.

Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is a specific numeric value, a range of values, a set

of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.

4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class is exercised at once.

Scheme prepared by Signature of the HOD

Paper Evaluators:

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