Q.2 a. Explain characteristics of software. (4)

Answer:

1) Software does not wear out

2) Software is not manufactured, but it is developed through the life cycle concept.

3) Reusability of components

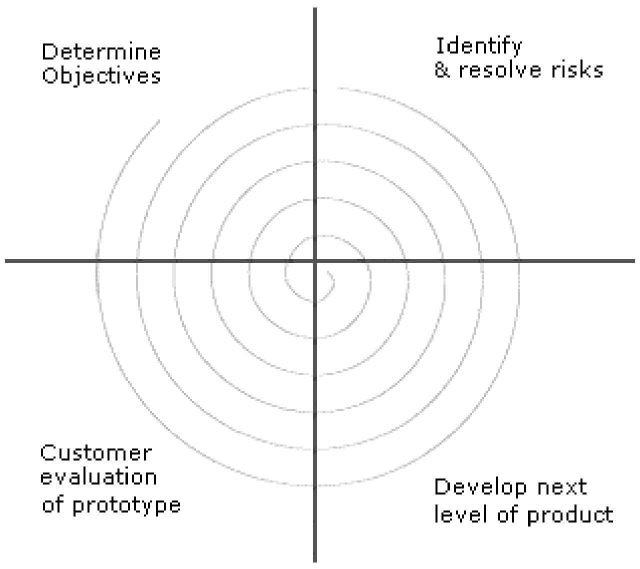
4) Software is flexible and can be amended to meet new requirements

5) Cost of its change at if not carried out initially is very high at later stage.

b. Explain the spiral model in detail. (6)

Answer:

The Spiral model of software development is shown in fig. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig.


Determine Objectives
Identify & resolve risks
Customer evaluation of prototype
Develop next level of product

The following activities are carried out during each phase of a spiral model.

First quadrant (Objective Setting)

During the first quadrant, it is n eeded to identify the objectives of the phase.
Examine the risks associated with these objectives.

Second Quadrant (Risk Assessment and Reduction)

• A detailed analysis is carried out for each identified project risk.

• Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

Third Quadrant (Development and Validation)

• Develop and validate the next level of the product after resolving the identified risks.

Fourth Quadrant (Review and Planning)

• Review the results achieved so far with the customer and plan the next iteration around the spiral.'

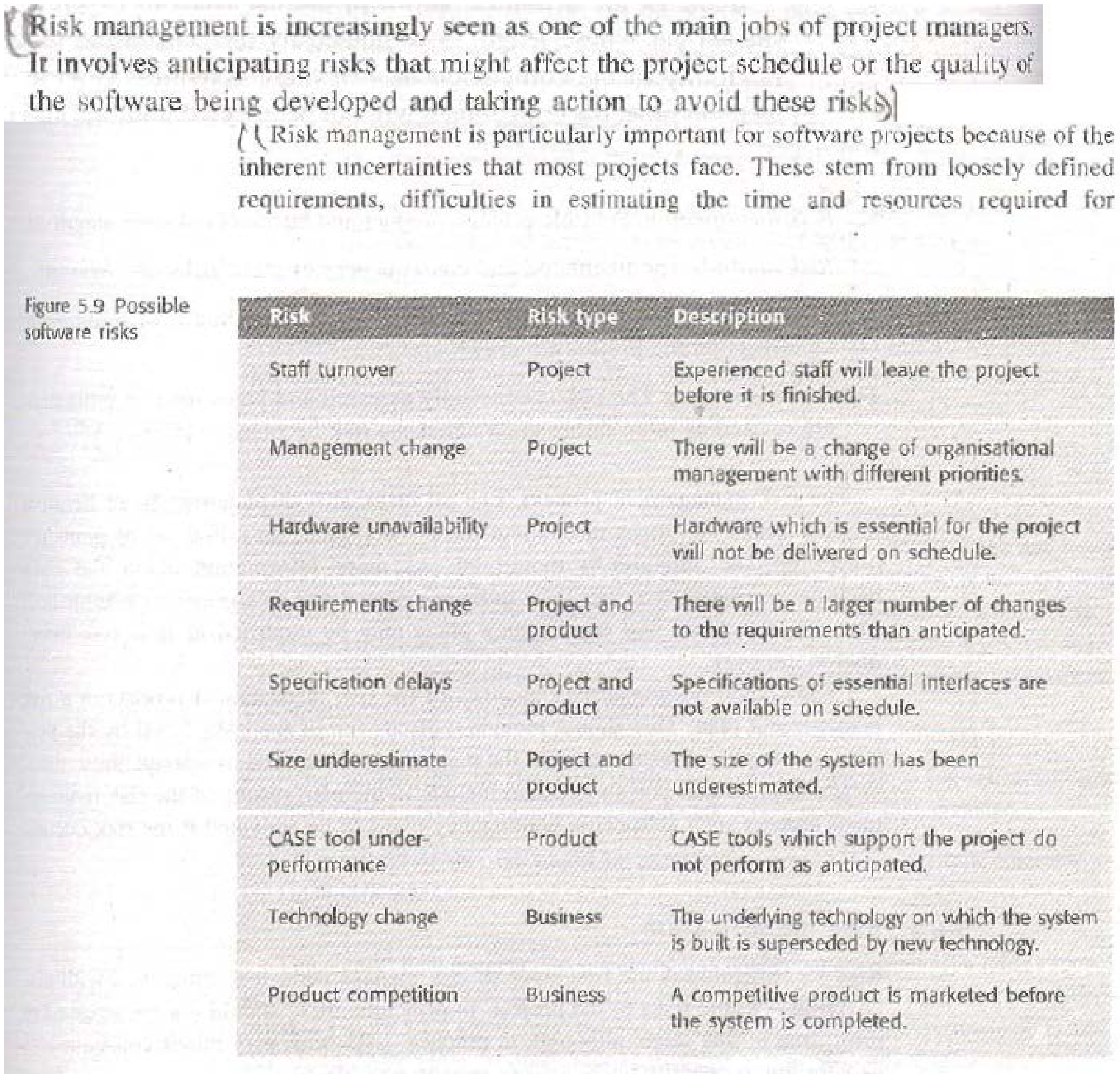
• Progressively more complete version of the software gets built with each iteration around the spiral

Advantages of Spiral Model
• It is risk-driven model.
• It is very flexible.
• Less documentation is needed.
• It uses prototyping

Disadvantages of Spiral Model
• No strict standards for software development.
• No particular beginning or end of a particular phase.

c. What is the process of Risk Management? (6)

Answer:



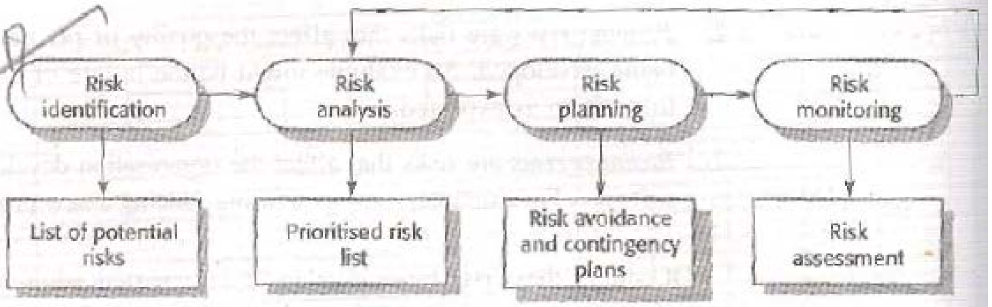
Risk management is increasingly seen as one of the main jobs of project managers. It involves anticipating risks that might affect the project schedule or the quality of the software being developed and taking action to avoid these risks.

Risk management is particularly important for software projects because of the inherent uncertainties that most projects face. These stem from loosely defined requirements, difficulties in estimating the time and resources required for

Figure 5.9 Possible software risks

Risk	Risk type	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organisational management with different priorities.
Hardware unavailability	Project	Hardware which is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool under-performance	Product	CASE tools which support the project do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Figure 5.10 The risk management process



software development, dependence on individual skills and requirements changes due to changes in customer needs.)

The process of risk management is illustrated in Figure 5.10. It involves several stages:

1. *Risk identification* Possible project, product and business risks are identified.
2. *Risk analysis* The likelihood and consequences of these risks are assessed.
3. *Risk planning* Plans to address the risk either by avoiding it or minimising its effects on the project are drawn up.
4. *Risk monitoring* The risk is constantly assessed and plans for risk mitigation are revised as more information about the risk becomes available.

Q.3 a. Explain functional and nonfunctional requirement. (8)

Answer:

• Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

● Describe functionality or system services.

● Depend on the type of software, expected users and the type of system where the software is used.

● Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

Examples of functional requirements

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

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

● Every order shall be allocated a unique identifier which the user shall be able to copy to the account's permanent storage area

non-functional requirements

● These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.

● Process requirements may also be specified mandating a particular CASE system, programming language or development method.

● Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional classifications-

● Product requirements

• Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

● Organisational requirements

• Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

● External requirements

• Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirements examples-

● Product requirement 8.1 The user interface for LIBSYS shall be implemented as simple HTML without frames or Java applets.

● Organisational requirement 9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SPSTAN-95

. ● External requirement 7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

b. Explain data dictionary. (4)

Answer:

Data flow analysis (DFD) consists of a 4 tools. These are

: 1) data flow diagram

2) data dictionary

3)data structure diagram

4) structure chart

Data dictionary contain the information about the data of a system. I.e. the data about data or Meta data. A data dictionary is organized into 5 sections:

a) Data elements b) Data flows c) Data stores d) Process e) External elements

c. What is the role of system analyst? (4)

Answer:

The system analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

1 ƒWhat is the problem?

2 ƒ Why is it important to solve the problem?

3 What are the possible solutions to the problem?

4 What exactly are the data input to the system and what exactly are the data output by the system?

5 What are the likely complexities that might arise while solving the problem?

6 If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and the incompleteness. When the analyst detects any inconsistencies, anomalies or incompleteness in gathered requirements, he resolves them by carrying out further discussions with the end-users and the customers.

Q.4 a. What is software prototyping? What are its benefits? (6)

Answer:

Analysis should be conducted regardless of the software engineering paradigm that is applied. However, the form that analysis takes will vary. In some cases it is possible to apply operational analysis principles and derive a model of software from which a design can be developed. In other situations, requirements elicitation (via FAST, QFD, use-cases, or other "brainstorming" techniques [JOR89]) is conducted, analysis principles are applied, and a model of the software to be built, called a prototype, is constructed for customer and developer assessment. Finally, some circumstances require the construction of a prototype at the beginning of analysis, since the model is the only means through which requirements can be effectively derived. The modelthen evolves into production software

Selecting the Prototyping Approach-The prototyping paradigm can be either close-ended or open-ended. The close-ended
approach is often called throwaway prototyping. Using this approach, a prototype
serves solely as a rough demonstration of requirements. It is then discarded, and the
software is engineered using a different paradigm. An open-ended approach, called
evolutionary prototyping, uses the prototype as the first part of an analysis activity that
will be continued into design and construction. The prototype of the software is the
first evolution of the finished system.
Before a close-ended or open-ended approach can be chosen, it is necessary to
determine whether the system to be built is amenable to prototyping. A number of
prototyping candidacy factors [BOA84] can be defined: application area, application
complexity, customer characteristics, and project characteristics.8
In general, any application that creates dynamic visual displays, interacts heavily
with a user, or demands algorithms or combinatorial processing that must be developed
in an evolutionary fashion is a candidate for prototyping. However, these application
areas must be weighed against application complexity. If a candidate application
(one that has the characteristics noted) will require the development of tens of thousands
of lines of code before any demonstrable function can be performed, it is likely
may still be possible to prototype portions of the software.
Because the customer must interact with the prototype in later steps, it is essential
That
(1) customer resources be committed to the evaluation and refinement of the
prototype and
(2) the customer is capable of making requirements decisions in a
timely fashion.

Finally, the nature of the development project will have a strong bearingon the efficacy of prototyping.

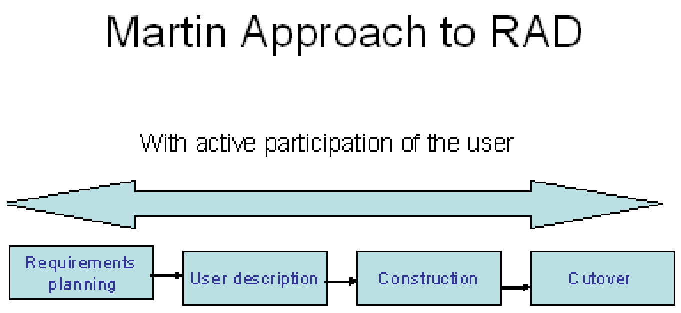
b. Explain in details RAD model. (10)

Answer:

RAD is proposed when requirements and solutions can be modularized as independent system or software components, each of which can be developed by different teams. User involvement is essential from requirement phase to deliver of the product. The process is stared with rapid prototype and is given to user for evolution. In that model user feedback is obtained and prototype is refined. SRS and design document are prepared with the association of users. RAD becomes faster if the software engineer uses the component's technology (CASE Tools) such that the components are really available for reuse. Since the development is distributed into component-development teams, the teams work in tandem and total development is completed in a short period (i.e., 60 to 90 days).

RAD Phases

• **Requirements planning phase** (a workshop utilizing structured discussion of business problems)
• **User description phase** – automated tools capture information from users
• **Construction phase** – productivity tools, such as code generators, screen generators, etc. inside a time-box. ("Do until done")
• **Cutover phase** -- installation of the system, user acceptance testing and user training



Advantage of RAD

• Dramatic time savings the systems development effort
• Can save time, money and human effort
• Tighter fit between user requirements and system specifications
• Works especially well where speed of development is important

Disadvantage of RAD

• More speed and lower cost may lead to lower overall system quality
• Danger of misalignment of system developed via RAD with the business due to missing information
• May have inconsistent internal designs within and across systems
• Possible violation of programming standards related to inconsistent naming conventions and inconsistent documentation

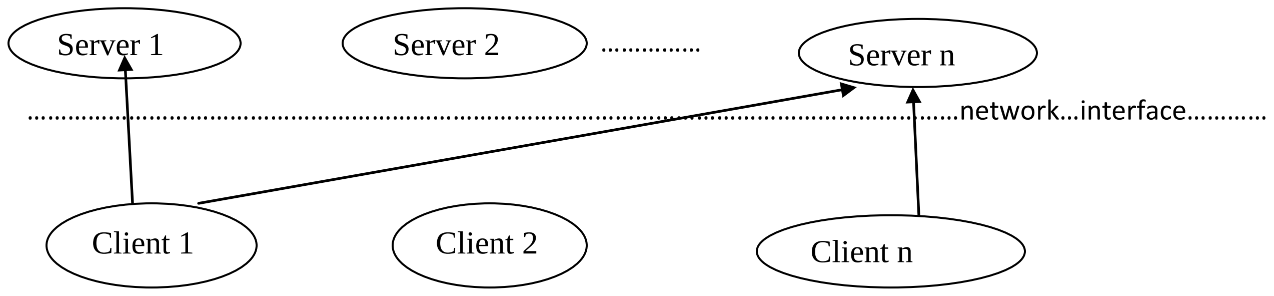
Q.5 a. Explain client server architecture. (10)

Answer:

The simplest way to connect clients and servers is a two-tier architecture as shown in fig. . In a two-tier architecture, any client can get service from any server by initiating a request over the network. With two tier client-server architectures, the user interface is usually located in the user's desktop and the services are usually supported by a server that is a powerful machine that can service many clients. Processing is split between the user interface and the database management server. There are a number of software vendors who provide tools to simplify development of applications for the two-tier client-server architecture.

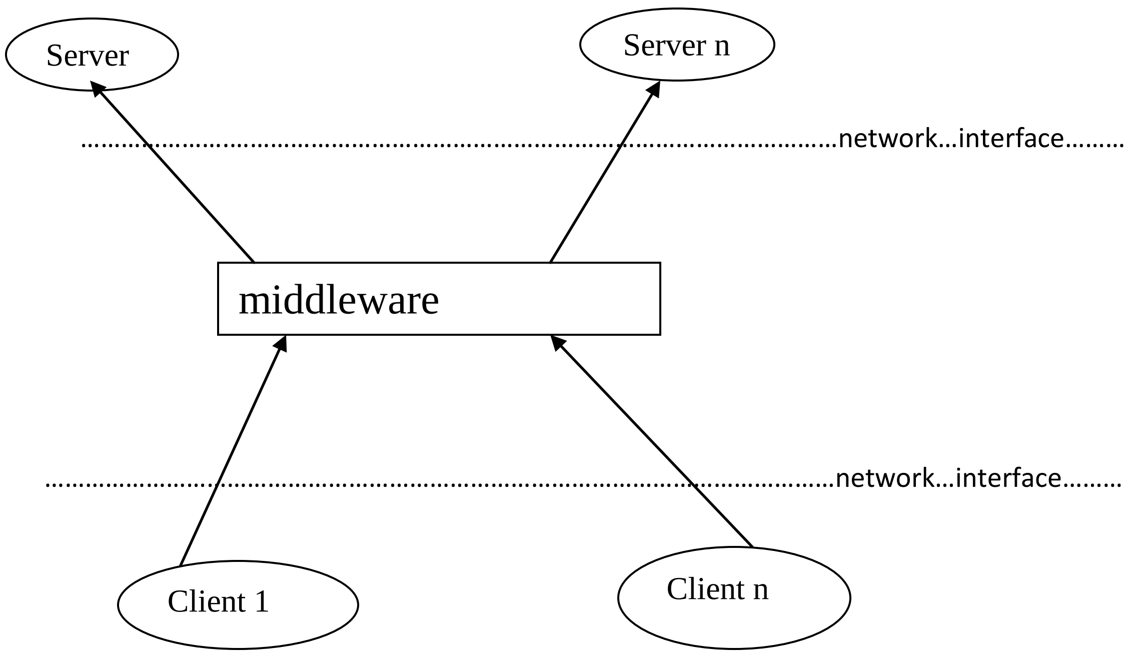
Limitations of two-tier client-server architecture-

A two tier architecture for client-server applications is an ideal solution but is not practical. The problem is that client and server components are manufactured by different vendors and the different vendors come up with different sets of interfaces and different implementation standards. That's the reason why clients and servers can often not talk to each other. A two tier architecture can work only in an open environment. In an open environment all components have standard interfaces. However, till date an open environment is still far from becoming practical.



Three-tier client-server architecture-

The three-tier architecture overcomes the important limitations of the two-tier architecture. In the three-tier architecture, a middleware was added between the user system interface client environment and the server environment as shown in fig. . The middleware keeps track of all server locations. It also translates client's requests into server understandable form. For example, if the middleware provides queuing, the client can deliver its request to the middleware and disengage because the middleware will access the data and return the answer to the client.


Server
Server n
...network...interface.........
middleware
...network...interface.........
Client 1
Client n

Three-tier client-server architecture

Functions of middleware-

The middleware performs many activities such as:

It knows the addresses of servers. So, based on client requests, it can locate the servers.

• It can translate between client and server formats of data and vice versa.

Popular middleware standards-

Two popular middleware standards are:

• CORBA (Common Object Request Broker Architecture)

• COM/DCOM

b. What is coupling? Explain its types. (6)

Answer:

***Coupling*: -** It is the measure of the degree of interdependence between modules. Coupling is highly between components if they depend heavily on one another, (e.g., there is a lot of communication between them).

Types of Coupling:-

1. Data coupling: communication between modules is accomplished through well-defined parameter lists consisting of data information items

2. Stamp coupling: Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.

3. Control coupling: a module controls the flow of control or the logic of another module. This is accomplished by passing control information items as arguments in the argument list.

4. Common coupling: modules share common or global data or file structures. This is the strongest form of coupling both modules depend on the details of the common structure

5. Content coupling: A module is allowed to access or modify the contents of another, e.g. modify its local or private data items. This the strongest form of coupling

Cohesion :-It is a measure of the degree to which the elements of a module are functionally related. Cohesion is weak if elements are bundled simply because the perform similar or related functions. Cohesion is weak if elements are bundled simply because they perform similar or related functions. Cohesion is strong if all parts are needed for the functioning of other parts (e..Important design objective is to maximize module cohesionand minimize module coupling

Q.6 a. Explain inheritance and polymorphism in object oriented design. (8)

Answer:

Inheritance is one of the key differentiators between conventional and OO systems.
A subclass **Y** inherits all of the attributes and operations associated with its superclass, **X**. This means that all data structures and algorithms originally designed and implemented for **X** are immediately available for **Y**—no further work need be done. Reuse has been accomplished directly.
Any change to the data or operations contained within a superclass is immediately inherited by all subclasses that have inherited from the superclass.2 Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system.

It is important to note that, at each level of the class hierarchy, new attributes and operations may be added to those that have been inherited from higher levels in the hierarchy. In fact, whenever a new class is to be created, the software engineer has a number of options

• The class can be designed and built from scratch. That is, inheritance is not used.
• The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required.

• The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
• Characteristics of an existing class can be overridden and private versions of attributes or operations are implemented for the new class

Polymorphism is a characteristic that greatly reduces the effort required to extend an existing OO system. To understand polymorphism, consider a conventional application that must draw four different types of graphs: line graphs, pie charts, histograms, andKiviat diagrams. Ideally, once data are collected for a particular type of graph, the graph should draw itself. To accomplish this in a conventional application (and maintain module cohesion), it would be necessary to develop drawing modules for each type of graph.
Although this design is reasonably straightforward, adding new graph types could be tricky. A new drawing module would have to be created for each graph type and then the control logic would have to be updated for each graph.
To solve this problem, all of the graphs become subclasses of a general class called **graph**. Using a concept called overloading , each subclass defines an operation calleddraw. An object can send a draw message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own draw operation to create the appropriate graph

b. Explain design pattern and application system reuse for software use. (8)

Answer:

● A design pattern is a way of reusing abstract knowledge about a problem and its solution.
● A pattern is a description of the problem and the essence of its solution.

● It should be sufficiently abstract to be reused in different settings.

● Patterns often rely on object characteristics such as inheritance and polymorphism.

Pattern elements-

● Name

 • A meaningful pattern identifier.

● Solution descript tion.

● Problem description.

• Not a concrete design but a template for a design solution that can be instantiated in different

● Consequencesways.

• The results and trade-offs of applying the pattern.

The Observer pattern

● Name

• Observer.

● Description

• Separates the display of object state from the object itself.

● Problem description

• Used when multiple displays of state are needed.

● Solution description

• See slide with UML description.

● Consequences

• Optimisations to enhance display performance are impractical.

Application system reuse

● Involves the reuse of entire application systems either by configuring a system for an environment or by integrating two or more systems to create a new application.

● Two approaches covered here:

• COTS product integration;

• Product line development.

COTS product reuse

COTS product reuse-

● COTS - Commercial Off-The-Shelf systems.

● COTS systems are usually complete application systems that offer an API (Application Programming Interface).

● Building large systems by integrating COTS systems is now a viable development strategy for some types of system such as Ecommerce systems.

● The key benefit is faster application development and, usually, lower development costs.

COTS design choices-

● Which COTS products offer the most appropriate functionality?

• There may be several similar products that may be used.

● How will data be exchanged

• Individual products use their own data structures and formats.

● What features of the product will actually be used?

• Most products have more functionality than is needed. You should try to deny access to unused functionality.

Product line development

● Architectures must be structured in such a way to separate different sub-systems and to allow them to be modified.

● The architecture should also separate entities and their descriptions and the higher levels in the system access entities through descriptions rather than directly

Q.7 a. Explain user interface design process in detail. (8)

Answer:

The design process for user interfaces is iterative and can be represented using a spiral model similar

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

The spiral implies that each of these tasks will occur more than
once, with each pass around the spiral representing additional elaboration of requirements
and the resultant design. In most cases, the implementation activity involves
prototyping—the only practical way to validate what has been designed.
The initial analysis activity focuses on the profile of the users who will interact
with the system. Skill level, business understanding, and general receptiveness to the
new system are recorded; and different user categories are defined. For each user
category, requirements are elicited. In essence, the software engineer attempts to
understand the system perception for each class of users

The analysis of the user environment focuses on the physical work environment.
Among the questions to be asked are
• Where will the interface be located physically?
• Will the user be sitting, standing, or performing other tasks unrelated to the interface?
• Does the interface hardware accommodate space, light, or noise constraints?
• Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis activity is used to create an analysis
model for the interface. Using this model as a basis, the design activity commences.
The goal of interface design is to define a set of interface objects and actions (and
their screen representations) that enable a user to perform all defined tasks in a manner
that meets every usability goal defined for the system.

The implementation activity normally begins with the creation of a prototype that
enables usage scenarios to be evaluated. As the iterative design process continues,
a user interface tool kit may be used to complete the construction of
the interface.
Validation focuses on
(1) the ability of the interface to implement every user task
correctly, to accommodate all task variations, and to achieve all general user requirements;
(2) the degree to which the interface is easy to use and easy to learn; and
(3)the users' acceptance of the interface as a useful tool in their work.
As we have already noted, the activities described in this section occur iteratively.
Therefore, there is no need to attempt to specify every detail (for the analysis or design
model) on the first pass. Subsequent passes through the process elaborate task detail,
design information, and the operational features of the interface.

b. Explain CBSE process (8)

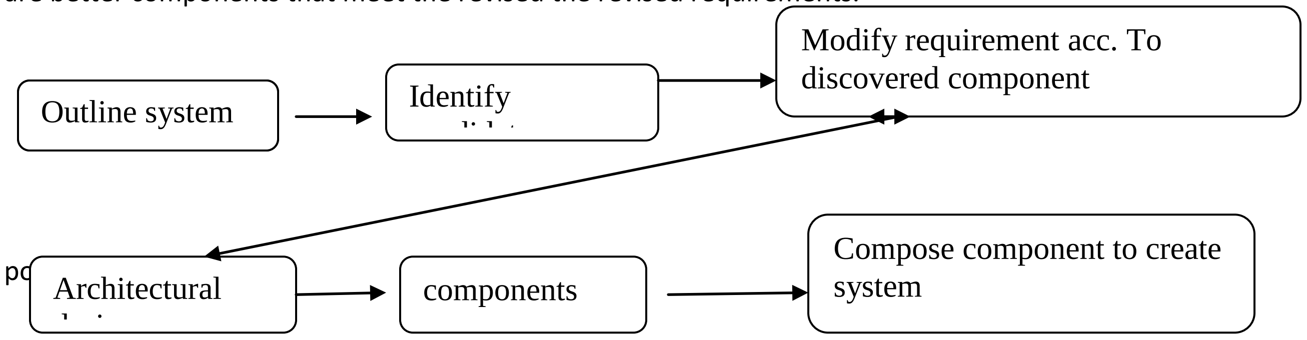
Answer:

When reusing components, it When reusing components, it is essential to make trade-offs between offs between ideal requirements and the services actually provided by available components. Components
. • This involves: This involves:

• Developing outline requirements; Developing outline requirements;

• Searching for components then modifying requirements according tdifying requirements according to available o available functionality. functionality.

• Searching again to find if there are better components that meet Searching again to find if there are better components that meet the revised the revised requirements.


Outline system
Identify
Modify requirement acc. To discovered component
po
Architectural
components
Compose component to create system

• CBSE is a reuse-based approach to defining and implementing loosely based approach to defining and implementing loosely coupled components into systems. coupled components into systems.

• A component is a software unit whose functionality and dependencies are ies are completely defined by its interfaces. completely defined by its interfaces.

• A component model defines a set of standards that component providers and composers should follow. and composers should follow.

• During the CBSE process, the processes of requirements engineeriesses of requirements engineering and system design are interleaved. system design are interleaved.

• Component composition is the process of Component composition is the process of 'wiring' components together to components together to create a system. create a system.

• When composing reusable components, you normally have to write ts, you normally have to write adaptors to reconcile different component interfaces. adaptors to reconcile different component interfaces.

• When choosing compositions, you have to consider required functiWhen choosing compositions, you have to consider required functionality, onality, non-functional requirements and system function

Q.8 a. What is white box testing? Explain Basis path testing. (10)

Answer:

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 attheir boundaries and within their operational bounds, and
(4) exercise internal datastructures to ensure their validity.
A reasonable question might be posed at this juncture: "Why spend time and energy
worrying about (and testing) logical minutiae when we might better expend effoensuring that program requirements have been met?" Stated another way, why don't
we spend all of our energy on black-box tests? The answer lies in the nature of software
defectsrt •Logic errors and incorrect assumptions are inversely proportional to the probability
that a program path will be executed. Errors tend to creep into our work
when we design and implement function, conditions, or control that are out
of the mainstream. Everyday processing tends to be well understood (and
well scrutinized), while "special case" processing tends to fall into the cracks.
• We often believe that a logical path is not likely to be executed when, in fact, it
may be executed on a regular basis. The logical flow of a program is sometimes
counterintuitive, meaning that our unconscious assumptions about
flow of control and data may lead us to make design errors that are uncovered
only once path testing commences.
• Typographical errors are random. When a program is translated into programming
language source code, it is likely that some typing errors will occur.
Many will be uncovered by syntax and type checking mechanisms, but others
may go undetected until testing begins. It is as likely that a typo will exist on
an obscure logical path as on a mainstream path.

Basis path testing-

Control Flow Graph (CFG) - A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph (as shown in fig. 10.3). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements. Fig. 10.3 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid's

Path-A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths

Linearly independent path- A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

Control flow graph-In order to understand the path coverage-based testing strategy, it is very much necessary to understand the control flow graph (CFG) of a program. Control flow graph (CFG) of a program has been discussed earlier.

Linearly independent pathp- The path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths. Linearly independent paths have been discussed earlier.

Cyclomatic complexity- For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for. There are three different ways to compute the cyclomatic complexity. The answers computed by the three methods are guaranteed to agree. Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as: V(G) = E – N + 2 where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph. For the CFG of example shown in fig. 10.4, E=7 and N=6. Therefore, the cyclomatic complexity = 7-6+2 = 3. Method 2: An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows: V(G) = Total number of bounded areas + 1 In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used.

b. Explain following testing methods: (3×2)
(i) Alpha and Beta testing.
(ii) Stress testing.

Answer:

• Alpha Testing. Alpha testing refers to the system testing carried out by the test team within the developing organization.

• Beta testing. Beta testing is the system testing performed by a select group of friendly customers.

Stress Testing- Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity. For example, suppose an operating system is supposed to support 15 multiprogrammed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts. Stress testing is especially important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours. For example, if the non-functional requirement specification states that the response time should not be more than 20 secs per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

Q.9 a. Explain function of SCM. (6)

Answer:

• **Identification** -identifies those items whose configuration needs to be controlled, usually consisting of hardware, software, and documentation.

• **Change Control** - establishes procedures for proposing or requesting changes, evaluating those changes for desirability, obtaining authorization for changes, publishing and tracking changes, and implementing changes. This function also identifies the people and organizations who have authority to make changes at various levels.

• **Status Accounting** -is the documentation function of CM. Its primary purpose is to maintain formal records of established configurations and make regular reports of configuration status. These records should accurately describe the product, and are used to verify the configuration of the system for testing, delivery, and other activities.

• **Auditing** -Effective CM requires regular evaluation of the configuration. This is done through the auditing function, where the physical and functional configurations are compared to the documented configuration. The purpose of auditing is to maintain the integrity of the baseline and release configurations for all controlled products

b. Write short notes on: (5×2)
(i) SQA.
(ii) COCOMO model.

Answer:

• It is a planned and systematic pattern of all actions necessary to provide adequate confidence that the time or product conforms to established technical requirements.”

• Purpose of SQAP is to specify all the works products that need to be produced during the project, activities that need to performed for checking the quality of each of the work product

• It is interested in the quality of not only the final product but also an intermediate product.

Verification:-is the process of determine whether or not product of a given phase of software development full fill the specification established during the previous phase.

Validation:-is the process of evaluating software at the end of software development to ensure compliance with the software requirement. testing is common method of validation Software V&V is a system engineering process employing rigorous methodologies for evaluating the correctness and quality of the software product throughout the software life cycle.

COCOMO is one of the most widely used software estimation models in the world

· It was developed by Barry Boehm in 1981

· COCOMO predicts the effort and schedule for a software product development based on inputs relating to the size of the software and a number of cost drivers that affect productivity

COCOMO has three different models that reflect the complexity:

1. Basic Model

2. Intermediate Model

3. Detailed Model The Development Modes: Project Characteristics Basic Model Basic model aim at estimating, in a quick and rough fashion, most of the small to medium sized software projects.These models of software development are considered in this model: organic, semi -detected and embedded. KLOC (thousands of lines of code) is a traditional measure of how large a computer program is or how long or how many people it will take to write it

Organic Mode

· developed in a familiar, stable environment,

· similar to the previously developed projects

· relatively small and requires little innovation

Semidetached Mode

· intermediate between Organic and Embedded

Embedded Mode

· tight, inflexible constraints and interface requirements

TEXT BOOK

I. Software Engineering, Ian Sommerville, 7th edition, Pearson Education, 2004