Q.2 a. What is project plan? Discuss the purpose of each of the sections in a software project plan. (5)

Answer:

The project plan sets out the resources available to the project, the work breakdown and a schedule for carrying out the work. In some organisations, the project plan is a single document that includes the different types of plan. In other cases, the project plan is solely concerned with the development process. The details of the project plan vary depending on the type of project and organisation. However, most plans should include the following sections:

1. *Introduction* This briefly describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the project management.
2. *Project organisation* This describes the way in which the development team is organised, the people involved and their roles in the team.
3. *Risk analysis* This describes possible project risks, the likelihood of these risks arising and the risk reduction strategies that are proposed.
4. *Hardware and software resource requirements* This specifies the hardware and the support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.
5. *Work breakdown* This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.
6. *Project schedule* This shows the dependencies between activities, the estimated time required to reach each milestone and the allocation of people to activities.
7. *Monitoring and reporting mechanisms* This defines the management reports that should be produced, when these should be produced and the project monitoring mechanisms used.

b. Describe the different activities involved in the system design process. (4)

Answer:

System design as shown in figure 2(b) is concerned with how the system functionality is to be provided by the components of the system. The activities involved in this process are:

1. *Partition requirements* You analyse the requirements and organise them into related groups. There are usually several possible partitioning options, and you may suggest a number of alternatives at this stage of the process.
2. *Identify sub-systems* You should identify sub-systems that can individually or collectively meet the requirements. Groups of requirements are usually related to sub-systems, so this activity and requirements partitioning may be amalgamated. However, sub-system identification may also be influenced by other organizational or environmental factors.
3. *Assign requirements to sub-systems* You assign the requirements to subsystems. In principle, this should be straightforward if the requirements partitioning is used to drive the sub-system identification. In practice, there is never a clean match between requirements partitions and identified sub-systems. Limitations of externally purchased sub-systems may mean that you have to change the requirements to accommodate these constraints.

4. *Specify sub-system functionality* You should specify the specific functions provided by each sub-system. This may be seen as part of the system design phase or, if the sub-system is a software system, part of the requirements specification activity for that system. You should also try to identify relationships between sub-systems at this stage.
5. *Define sub-system interfaces* You define the interfaces that are provided and required by each sub-system. Once these interfaces have been agreed upon, it becomes possible to develop these sub-systems in parallel.

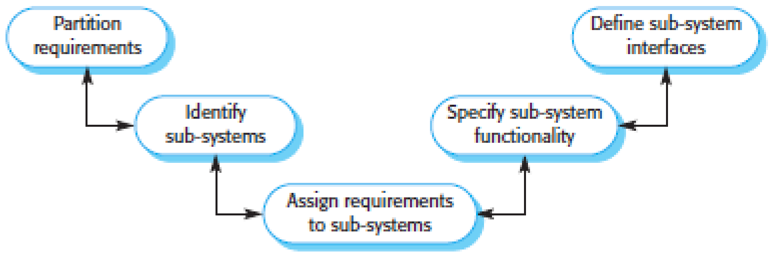

Partition requirements
Define sub-system interfaces
Identify sub-systems
Specify sub-system functionality
Assign requirements to sub-systems

Figure 2(b) The system design process

c. **With the help of suitable figure, discuss the four main phases in the requirements engineering process. (7)**

Answer:
Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation.
The requirements engineering process is shown in figure 2(c). This process leads to the production of a requirements document that is the specification for the system.
Requirements are usually presented at two levels of detail in this document. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.
There are four main phases in the requirements engineering process:
1. *Feasibility study* An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and whether it can be developed within existing budgetary constraints. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether to go ahead with a more detailed analysis.
2. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis and so on. This may involve the development of one or more system models and prototypes. These help the analyst understand the system to be specified.
3. *Requirements specification* The activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. *User requirements* are abstract statements of the system requirements for the customer and

end-user of the system; *system requirements* are a more detailed description of the functionality to be provided.
4. *Requirements validation* This activity checks the requirements for realism, consistency and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.
Of course, the activities in the requirements process are not simply carried out in a strict sequence. Requirements analysis continues during definition and specification, and new requirements come to light throughout the process. Therefore, the activities of analysis, definition and specification are interleaved. In agile methods such as extreme programming, requirements are developed incrementally according to user priorities, and the elicitation of requirements comes from users who are part of the development team.

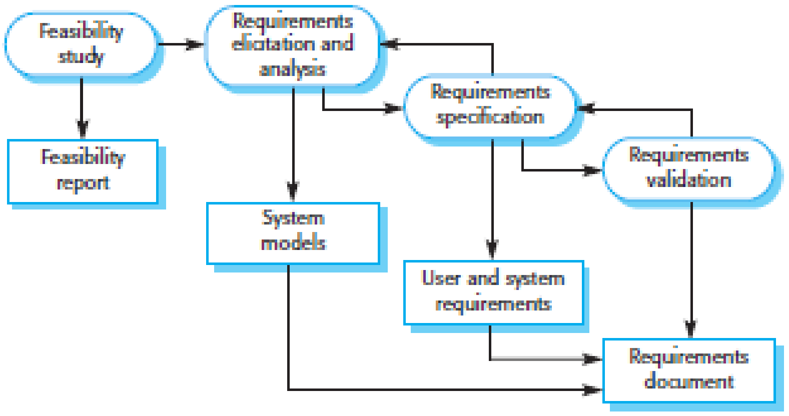


Figure 2(c) The requirements engineering process

Q.3 a. What is requirement validation? What are the different checks should be carried out on the requirements in the requirement document during the requirements validation process? What are the requirements validations techniques that can be used in conjunction or individually? (2+4+4)

Answer:
Requirements validation is concerned with showing that the requirements actually define the system that the customer wants. Requirements validation overlaps analysis in that it is concerned with finding problems with the requirements. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when they are discovered during development or after the system is in service. The cost of fixing a requirements problem by making a system change is much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed and then the system must be tested again.

<u>During the requirements validation process, checks should be carried out on the requirements in the requirements document. These checks include:</u>
1. *Validity checks* A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that

are required. Systems have diverse stakeholders with distinct needs, and any set of requirements is inevitably a compromise across the stakeholder community.
2. *Consistency checks* Requirements in the document should not conflict. That is, there should be no contradictory constraints or descriptions of the same system function.
3. *Completeness checks* The requirements document should include requirements, which define all functions, and constraints intended by the system user.
4. *Realism checks* Using knowledge of existing technology, the requirements should be checked to ensure that they could actually be implemented. These checks should also take account of the budget and schedule for the system development.
5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

<u>A number of requirements validation techniques can be used in conjunction or individually:</u>
1. *Requirements reviews* The requirements are analysed systematically by a team of reviewers. This process is discussed in the following section.
2. *Prototyping* In this approach to validation, an executable model of the system is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

b. Give example of the type of system models that you might create during the analysis process. (6)

Answer:
Examples of the types of system models that might be created during the analysis process are:

- *A data-flow model:* Data-flow models show how data is processed at different stages in the system.
- *A composition model:* A composition or aggregation model shows how entities in the system are composed of other entities.
- *An architectural model:* Architectural models show the principal sub-systems that make up a system.
- *A classification model:* Object class/inheritance diagrams show how entities have common characteristics.
- *A stimulus-response model:* A stimulus-response model, or state transition diagram, shows how the system reacts to internal and external events.

Q.4 a. In respect of the sub-system interface specification, give the structure of an object specification and discuss the different components of the body of the specification. (6)

Answer:
The structure of an object specification is shown in figure 4(a). The body of the specification has four components.
1. *An introduction* that declares the sort (the type name) of the entity being specified. A sort is the name of a set of objects with common characteristics. It is similar to a type in a programming language. The introduction may also include an 'imports' declaration, where the names of specifications defining other sorts are declared. Importing a specification makes these sorts available for use.
2. *A description* part, where the operations are described informally. This makes the formal specification easier to understand. The formal specification complements this description by providing an unambiguous syntax and semantics for the type operations.
3. *The signature* part defines the syntax of the interface to the object class or abstract data type. The names of the operations that are defined, the number and sorts of their parameters, and the sort of operation results are described in the signature.
4. *The axioms* part defines the semantics of the operations by defining a set of *axioms* that characterise the behaviour of the abstract data type. These axioms relate the operations used to construct entities of the defined sort with operations used to inspect its values.

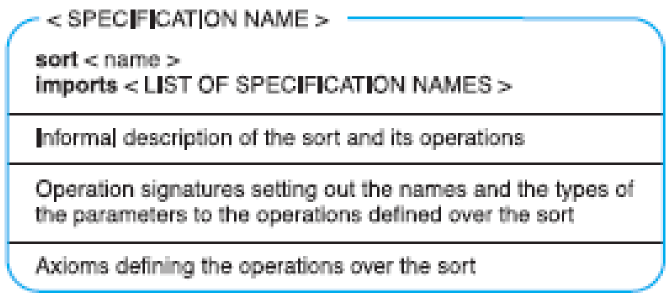

< SPECIFICATION NAME >
sort < name >
imports < LIST OF SPECIFICATION NAMES >
Informal description of the sort and its operations
Operation signatures setting out the names and the types of the parameters to the operations defined over the sort
Axioms defining the operations over the sort

Figure 4(a) The structure of an algebraic specification

b. **What is Pair Programming? What are the advantages of pair programming? (5)**

Answer:
Pair programming is a concept in which programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. Development does not actually involve the same pair of people working together. Rather, the idea is that pairs are created dynamically so that all team members may work with other members in a programming pair during the development process.

The use of pair programming has a number of advantages:

- It supports the idea of common ownership and responsibility for the system. This reflects Weinberg's ides of egoless programming where the software is owned by the team as whole and individuals are not held responsibility for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews are very successful in discovering a high percentage of software errors. However, they are time consuming to

organise and, typically, introduce delays into the development process. While pair programming is a less formal process that probably doesn't find so many errors, it is a much cheaper inspection process than formal program inspections.

- It helps support refactoring, which is a process of software improvement. A principle of extreme programming (XP) is that the software should be constantly refactored. That is, parts of the code should be rewritten to improve their clarity or structure. The difficulty of implementing this in a normal development environment is that this is effort that is expended for long-term benefit, and an individual who practices refactoring may be judged less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others gain immediately from the refactoring so they are likely to support the process.

c. Differentiate between evolutionary and throw-away prototyping. (5)

Answer:

Q.5 a. What is client-server model? What are the major components of this model? (5)

Answer:

The client–server architectural model is a system model where the system is organized as a set of services and associated servers and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other sub-systems. Examples of servers are print servers that offer printing services, file servers that offer file management services and a compile server, which offers programming language compilation services.
2. A set of clients that call on the services offered by servers. These are normally sub-systems in their own right. There may be several instances of a client program executing concurrently.
3. A network that allows the clients to access these services. This is not strictly necessary as both the clients and the servers could run on a single machine. In practice, however, most client–server systems are implemented as distributed systems.

Clients may have to know the names of the available servers and the services that they provide. However, servers need not know either the identity of clients or how many clients there are. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

b. Draw the structure of a CORBA-based distributed application based on OMG's vision of a distributed application and discuss the various components proposed in the distributed application. (6)

Answer:

The OMG's vision of a distributed application is shown in figure 5(b). This proposes that a distributed application should be made up of a number of components:

1. Application objects that are designed and implemented for this application.

2. Standard objects that are defined by the OMG for a specific domain. These domain object standards cover finance/insurance, electronic commerce, healthcare, and a number of other areas.
3. Fundamental CORBA services that provide basic distributed computing services such as directories and security management.
4. Horizontal CORBA facilities such as user interface facilities, system management facilities, and so on. The term *horizontal facilities* suggests that these facilities are common to many application domains and the facilities are therefore used in many different applications.

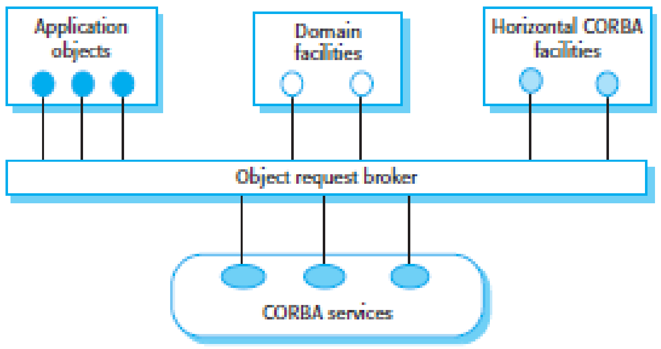


Figure 5(b) The structure of a CORBA-based distributed application

c. Write the advantages and disadvantages of broadcast model approach. (5)

Answer:
The **advantage** of the broadcast approach is that evolution is relatively simple. A new sub-system to handle particular classes of events can be integrated by registering its events with the event handler. Any sub-system can activate any other sub-system without knowing its name or location. The sub-systems can be implemented on distributed machines. This distribution is transparent to other sub-systems.
The **disadvantage** of this model is that sub-systems don't know if or when events will be handled. When a sub-system generates an event it does not know which other sub-systems have registered an interest in that event. It is quite possible for different sub-systems to register for the same events. This may cause conflicts when the results of handling the event are made available.

Q.6 a. What do you mean by component composition? What are the different types of component composition? What are the different situations of incompatibility occurred when components are developed independently? (2+3+3)

Answer:
Component composition is the process of assembling components to create a system. If we assume a situation where reusable components are available, then most systems will be constructed by composing these reusable components with each other, with specially written components and with the component support infrastructure provided by the model framework. This infrastructure provides facilities to support component communication and horizontal services such as user interface services, transaction management,

concurrency and security. The ways in which components are integrated with this infrastructure are documented for each component model.
Composition is not a simple operation; there are a number of types as shown in figure 6(a):
1. *Sequential composition* This occurs when, in the composite component, the constituent components are executed in sequence. It corresponds to situation (a) in figure 6(a), where the provides interfaces of each component are composed. Some extra code is required to make the link between the components.
2. *Hierarchical composition* This occurs when one component calls directly on the services provided by another component. It corresponds to a situation where the provides interface of one component is composed with the requires interface of another component. This is situation (b) in figure 6(a).
3. *Additive composition* This occurs when the interfaces of two or more components are put together (added) to create a new component. The interfaces of the composite component are created by putting together all of the interfaces of the constituent components, with duplicate operations removed if necessary. This corresponds to situation (c) in figure 6(a).
You might use all the forms of component composition when creating a system. In all cases, you may have to write 'glue code' that links the components. For example, for sequential composition, the output of component A typically becomes the input to component B. You need intermediate statements that call component A, collect the result and then call component B with that result as a parameter.
When you write components especially for composition, you design the interfaces of these components so that they are compatible. You can therefore easily compose these components into a single unit. However, when components are developed independently for reuse, you will often be faced with interface incompatibilities where the interfaces of the components that you wish to compose are not the same. Three types of incompatibility can occur:
1. *Parameter incompatibility* The operations on each side of the interface have the same name but their parameter types or the number of parameters are different.
2. *Operation incompatibility* The names of the operations in the provides and requires interfaces are different.
3. *Operation incompleteness* The provides interface of a component is a subset of the requires interface of another component or vice versa.

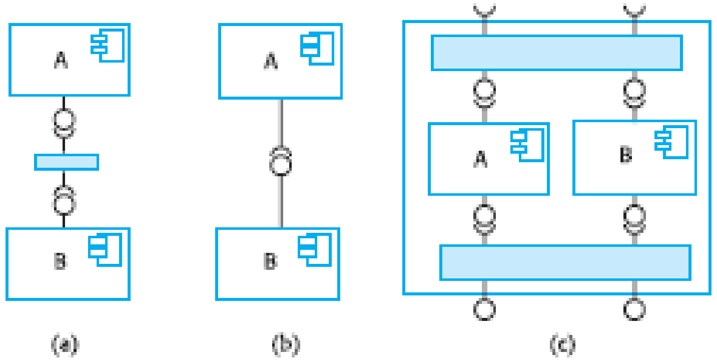


Figure 6(a) Types of component composition

b. What are the key factors that one should consider while planning for software reuse? (8)

Answer:

Following are the key factors that should be considered while planning for software reuse:

- ***The development schedule for the software:*** If the software has to be developed quickly, one should try to reuse off-the-shelf systems rather than individual components. These are large-grain reusable assets. Although the fit to requirements may be imperfect, this approach minimises the amount of development required.
- ***The expected software lifetime:*** If developing a long-lifetime system, one should focus on the maintainability of the system. In those circumstances, one should not just think about the immediate possibilities of reuse but also the long term implications. They will have to adapt the system to new requirements, which will probably mean making changes to components and how they are used. If one has do not access to the source code, you should probably avoid using components and systems from external suppliers; you cannot be sure that these suppliers will be able to continue supporting the reused software.
- ***The background, skills and experience of the development team:*** All reuse technologies are fairly complex and need quite a lot of time to understand and use them effectively. Therefore, if the development team has skills in a particular area, this is probably where you should focus.
- ***The criticality of the software and its non-functional requirements:*** For a critical system that has to be certified by an external regulator, one has to create a dependability case for the system. This is difficult if you don't have access to the source code of the software. It the software has stringent performance requirements, it may be impossible to use strategies such as reuse through program generators. These systems tend to generate relatively inefficient code.
- ***The application domain:*** In some application domains, such as manufacturing and medical information systems, there are several generic products that may be reused by configuring them to a local situation. If working in such domain, one should always consider these an option.
- ***The platform on which the system will run:*** Some components models, such as COM/Active X, are specific to Microsoft platforms. If you are developing on such a platform, this may be the most appropriate approach. Similarly, generic application systems may be platform-specific and may only be able to reuse these if your system is designed for the same platform.

Q.7 a. Define dependable process. What are the characteristics of dependable process? (2+4)

Answer:

Dependable software processes are processes that are geared to fault avoidance and fault detection. Dependable processes are well defined and repeatable, and include a spectrum of verification and validation activities. A well-defined process is a process that has been standardised and documented. A repeatable process is one that does not rely on individual interpretation and judgement. Irrespective of the people involved in the process, the organisation can be confident that the process will be successful. A dependable process should always include well-planned, comprehensive verification and validation activities whose aim is to ensure residual faults in the software are discovered before it is deployed.

The essential characteristics of dependable processes are as follows:

- Documentable: The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
- Standardised: A comprehensive set of software development standards that define how the software is to be produced and documented should be available.
- Auditable: The process should be understandable by people apart from process participants who can check that process standards are being followed and make suggestions for process improvement.
- Diverse: The process should include redundant and diverse verification and validation activities.
- Robust: The process should be able to recover from failures of individual process activities.

b. What do you mean by user integration? What are the different styles in which forms of interaction can be classified? Give one advantage, disadvantage and an example of each style. (2+4+4)

Answer:

User integration means issuing commands and associated data to the computer system. On early computers, the only way to do this was through a command-line interface, and a special-purpose language was used to communicate with the machine.

Shneiderman has classified the forms of interaction into five primary styles:

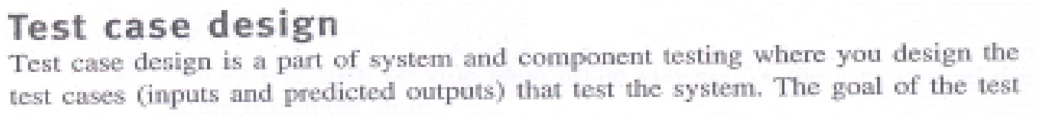
1. ***Direct manipulation:*** The user interacts directly with objects on the screen. Direct manipulation usually involves a pointing device (a mouse, a stylus, a trackball or, on touch screens, a finger) that indicates the object to be manipulated and the action, which specifies what should be done with that object. For example, to delete a file, you may click on an icon representing that file and drag it to a trash can icon.
2. ***Menu selection:*** The user selects a command from a list of possibilities (a menu). The user may also select another screen object by direct manipulation, and the command operates on that object. In this approach, to delete a file, you would select the file icon then select the delete command.
3. ***Form fill-in:*** The user fills in the fields of a form. Some fields may have associated menus, and the form may have action 'buttons' that, when pressed, cause some action to be initiated.
4. ***Command language:*** The user issues a special command and associated parameters to instruct the system what to do. To delete a file, you would type a delete command with the filename as a parameter.
5. ***Natural language:*** The user issues a command in natural language. This is usually a front end to a command language; the natural language is parsed and

translated to system commands. To delete a file, you might type 'delete the file named @@@'.

Interaction Style	**Advantage**	**Disadvantage**	**Application examples**
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement Only suitable where there is a visual metaphor for tasks and objects.	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for inexperienced users Can become complex if many menu options	Most general purpose systems
Form fill-in	Simple data entry	Takes up a lot of screen space Causes problems where user options do match the form fields	Stock control Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems Command and control systems
Natural language	Accessible to causal users Easily extended	Requires more typing Natural language understanding systems are unreliable	Information retrieval systems

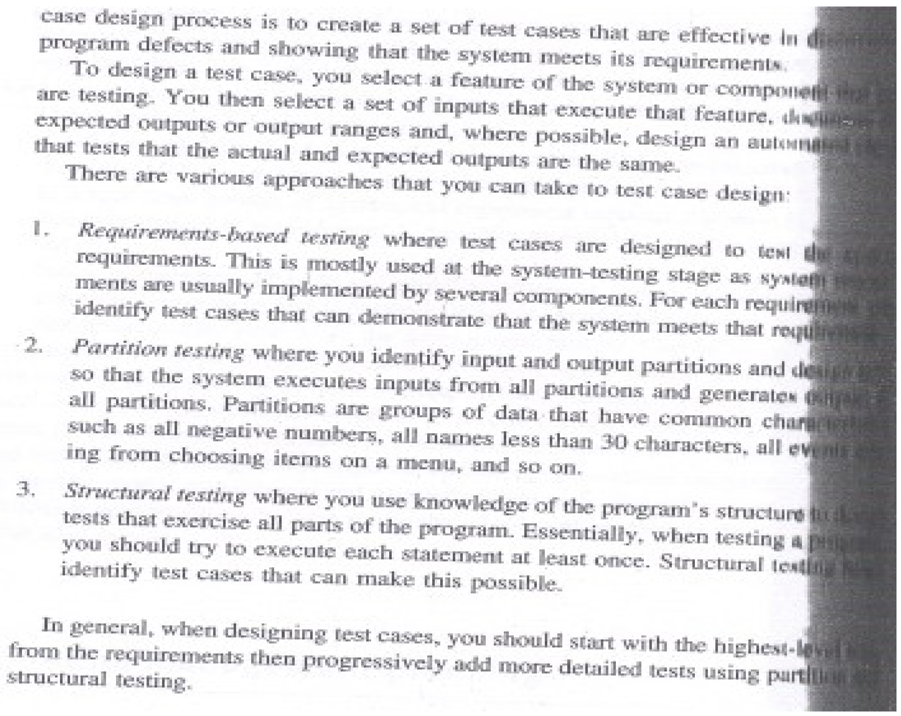
Q.8 a. What do you mean by test case design? What approaches can be taken for test case design? (5)

Answer:



Test case design

Test case design is a part of system and component testing where you design the test cases (inputs and predicted outputs) that test the system. The goal of the test



case design process is to create a set of test cases that are effective in
program defects and showing that the system meets its requirements.

To design a test case, you select a feature of the system or compon
are testing. You then select a set of inputs that execute that feature,
expected outputs or output ranges and, where possible, design an autom
that tests that the actual and expected outputs are the same.

There are various approaches that you can take to test case design:

1. *Requirements-based testing* where test cases are designed to test
requirements. This is mostly used at the system-testing stage as system
ments are usually implemented by several components. For each require
identify test cases that can demonstrate that the system meets that requ

2. *Partition testing* where you identify input and output partitions and
so that the system executes inputs from all partitions and generates
all partitions. Partitions are groups of data that have common chara
such as all negative numbers, all names less than 30 characters, all ev
ing from choosing items on a menu, and so on.

3. *Structural testing* where you use knowledge of the program's structure
tests that exercise all parts of the program. Essentially, when testing a
you should try to execute each statement at least once. Structural test
identify test cases that can make this possible.

In general, when designing test cases, you should start with the highest-l
from the requirements then progressively add more detailed tests using parti
structural testing.

b. What is software testing workbench? With the help of suitable figure, discuss some of the tools that might be included in such a testing workbench? (6)

Answer:

A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data. Figure 8(b) shows some of the tools that might be included in such a testing workbench:

1. *Test manager:* Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.

2. *Test data generator:* Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.

3. *Oracle:* Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-to-back testing involves running the oracle and the program to be tested in parallel.

4. *File comparator:* Compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.

5. *Report generator:* Provides report definition and generation facilities for test results.

6. *Dynamic analyser:* Adds code to a program to count the number of times each statement has been executed. After testing, an execution profile is generated showing how often each program statement has been executed.

7. *Simulator:* Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute. User interface simulators are script-

driven programs that simulate multiple simultaneous user interactions. Using simulators for I/O means that the timing of transaction sequences is repeatable.
When used for large system testing, tools have to be configured and adapted for the specific system that is being tested.

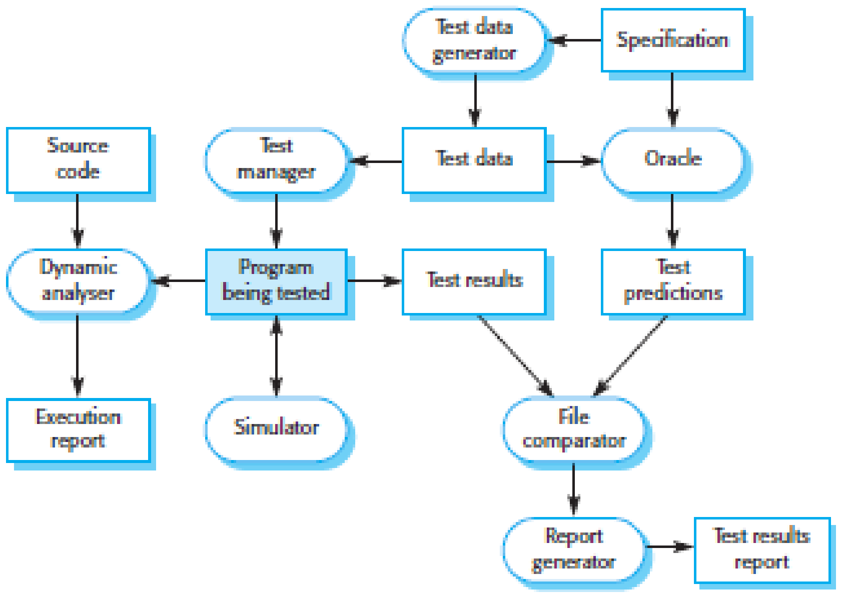
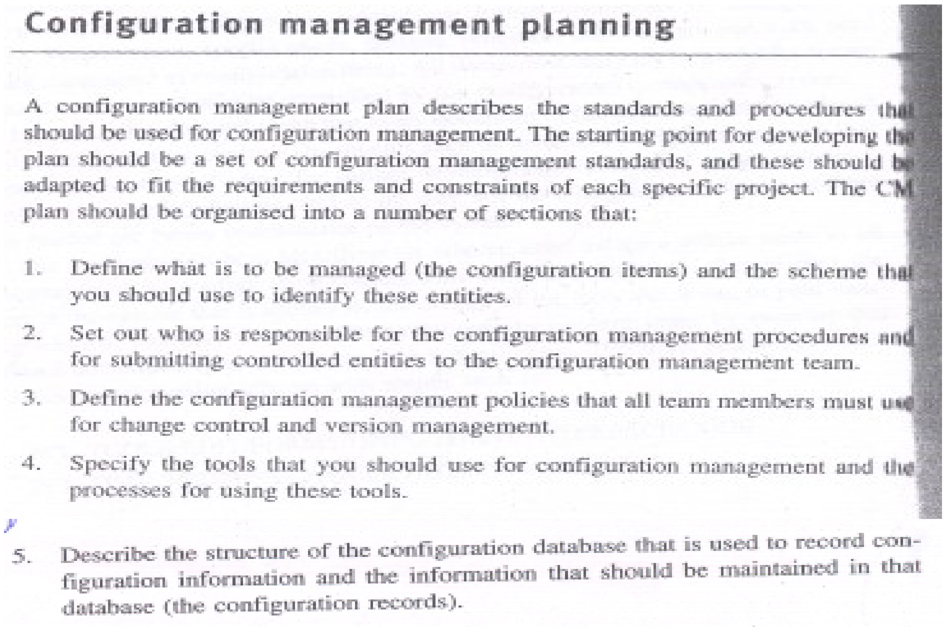


Figure 8(b) A testing workbench

c. What is a configuration management plan? Briefly describe its sections. (5)

Answer:



Configuration management planning

A configuration management plan describes the standards and procedures that should be used for configuration management. The starting point for developing the plan should be a set of configuration management standards, and these should be adapted to fit the requirements and constraints of each specific project. The CM plan should be organised into a number of sections that:

1. Define what is to be managed (the configuration items) and the scheme that you should use to identify these entities.
2. Set out who is responsible for the configuration management procedures and for submitting controlled entities to the configuration management team.
3. Define the configuration management policies that all team members must use for change control and version management.
4. Specify the tools that you should use for configuration management and the processes for using these tools.
5. Describe the structure of the configuration database that is used to record configuration information and the information that should be maintained in that database (the configuration records).

Q.9 a. Describe the key stages of software measurement process that may be part of a quality control process. (8)

Answer:

A software measurement process that may be part of a quality process is shown in figure 9(a) below. Each of the components of the system is analysed separately, and the values of the metric compared both with each other and, perhaps, with historical measurement data collected on previous projects. Anomalous measurements should be used to focus the quality assurance effort on components that may have quality problems.

The key stages in this process are:

- ***Choose measurements to be made:*** The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined. Measurements that are not directly relevant to these questions need not be collected. Basili's GQM (Goal-Question-Metric) paradigm is a good approach to use when deciding what data is to be collected.
- ***Select components to be assessed:*** It may not be necessary or desirable to assess metric values for all of the components in a software system. In some cases, you can select a representative selection of components for measurement. In others, components that are particularly critical, such as core components that are in almost constant use, should be assessed.
- ***Measure component characteristics:*** The selected components are measured and the associated metric values computed. This normally involves processing the component representation (design, code, etc.) using an automated data collection tool. This tool may be specially written or may already be incorporated in CASE tools that are used in an organisation.
- ***Identify anomalous measurements:*** Once the component measurements have been made, you should compare them to each other and to previous measurements that have been recorded in a measurement database. You should look for unusually high or low values for each metric, as these suggest that there could be problems with the component exhibiting these values.
- ***Analyse anomalous components:*** Once components that have anomalous values for particular metrics have been identified, you should examine these components to decide whether the anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity does not necessarily mean a poor quality component. There may be some other reason for the high value and it may not mean that there are component quality problems.

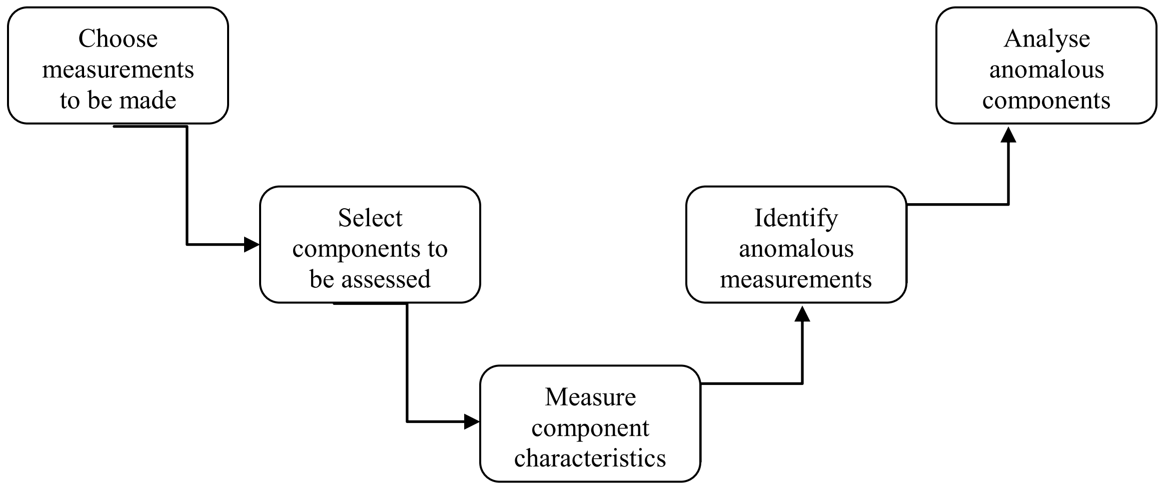

Choose measurements to be made
Select components to be assessed
Measure component characteristics
Identify anomalous measurements
Analyse anomalous components

Figure 9(a) The process of product measurement

b. **What are the types of standards that may be established as part of the quality assurance process? Why these standards are important? Give some examples of standards that may be included in standards handbook prepared by quality assurance team. (3+3+2)**

Answer:

The two types of standards that may be established as part of the quality assurance process are:

1. *Product standards:* These standards apply to the software product being developed. They include document standards, such as the structure of requirements documents; documentation standards, such as a standard comment header for an object class definition; and coding standards that define how a programming language should be used.
2. *Process standards:* These standards define the processes that should be followed during software development. They may include definitions of specification, design and validation processes and a description of the documents that should be written in the course of these processes.

Software standards are important for several reasons:

1. They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often only acquired after a great deal of trial and error. Building it into a standard helps the company avoid repeating past mistakes. Standards capture wisdom that is of value to the organisation.
2. They provide a framework for implementing the quality assurance process. Given that standards encapsulate best practice, quality assurance involves ensuring that appropriate standards have been selected and are used.
3. They assist in continuity where work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices. Consequently, learning effort when starting new work is reduced.

Quality assurance teams that are developing standards for a company should normally base their organisational standards on national and international standards. Using these standards as a starting point, the quality assurance team should draw up a standards ‘handbook’. This should define the standards that are needed by their organisation. Examples of standards that might be included in such a handbook are shown below:

Product standards	**Process standards**
Design review form	Design review conduct
Requirements document structure	Submission of documents to CM
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

TEXT BOOK

I. Software Engineering, Ian Sommerville, 7th edition, Pearson Education, 2004 (TB-I)