Q2 (a) Define software process. Explain the software process model.

Answer Page Number 8 of Text Book

Q2 (b) List the different steps of management activities.

Answer

Following are the different management activities:-

- Proposal writing
- Project writing and scheduling
- Project cost
- Project monitoring and reviews
- Personnel selection and evaluation

Report writing and presentations

Q2 (c) What are the advantages of incremental development process?

Answer

The incremental development process has a number of advantages:

- Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.
- Customers can use the early increments as prototype and gain experience that informs their requirements for later system increments.
- There is a lower risk of overall project failure. Although problems may be encountered in some increments, it is likely that some will be successfully delivered to the customer.
- As the highest priority services are delivered first, and later increments are integrated with them, it is inevitable that most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

Q3 (a) Give example of the type of system models that you might create during the analysis process?

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.

Q3 (b) Give the structure suggested by IEEE/ANSI 830 – 1998 for requirements documents.

Answer

The most widely known standard IEEE/ANSI 830 – 1998 suggests the following structure for requirements documents:

1. **Introduction**
   - 1.1 Purpose of the requirements document
   - 1.2 Scope of the product
   - 1.3 Definitions, acronyms and abbreviations
   - 1.4 References
   - 1.5 Overview of the remainder of the document
2. **General description**
   - 2.1 Product perspective
   - 2.2 Product functions
   - 2.3 User characteristics
   - 2.4 General constraints
   - 2.5 Assumptions and dependencies
3. **Specific requirements** cover functional, non-functional and interface requirements. This is obviously the most substantial part of the document but because of the wide variability in organisational practice, it is not appropriate to define a standard structure for this section. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.
4. **Appendices**
5. **Index**
6.

Q3 (c) What do you understand by requirement elicitation? Discuss any two techniques in detail?

Answer

Requirement Elicitation: It is the activity that helps to understand the problem to be solved. Requirements are gathered by asking question, writing down the answers, asking other questions, etc. Hence, a requirement gathering is the most communications intensive activity of the software development. Requirements elicitation requires the collaboration of several groups of participants who have different background.

The two important techniques of requirement elicitation are explained below:

Initiating the Process

The most commonly used requirements elicitation technique is to conduct a meeting or interview. The first meeting between a software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask; both are worried that what they do say will be misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here); both want to get the thing over with, but at the same time, both want it to be a success.

The analyst must start by asking *context-free questions*. That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself. The first set of context-free questions focuses on the customer, the overall goals, and the benefits. For example, the analyst might ask:

- Who is behind the request for this work?

- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development. The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions called as *meta-questions* focuses on the effectiveness of the meeting.

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Facilitated Application Specification Techniques

Too often, customers and software engineers have an unconscious "us and them" mind-set. Rather than working as a team to identify and refine requirements, each constituency defines its own "territory" and communicates through a series of memos, formal position papers, documents, and question and answer sessions. History has shown that this approach doesn't work very well. Misunderstandings abound, important information is omitted, and a successful working relationship is never established.

It is with these problems in mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called *facilitated application specification techniques* (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements. FAST has been used predominantly by the information systems community, but the technique offers potential for improved communication in applications of all kinds.

Many different approaches to FAST have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- A meeting is conducted at a neutral site and attended by both software engineers and customers.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

Q4 (a) What are the activities involved during the process of developing a forma specification of a sub-system interface?

Answer

The process of developing a formal specification of a sub-system interface includes the following activities:

- ***Specification structuring:*** Organise the informal interface specification into a set of abstract data types or object classes. One should informally define the operations associated with each class.
- ***Specification naming:*** Establish a name for each abstract type specification, decide whether they require generic parameters and decide on names for the sorts identified.
- ***Operation selection:*** Choose a set of operations for each specification based on the identified interface functionality. You should include operations to create instances of the sort, to modify the value of the instances and to inspect the instance values. You may have to add functions to those initially identified in the informal interface definition.
- ***Informal operation specification:*** Write an informal specification of each operation. You should describe how the operations affect the defined sort.
- ***Syntax definition:*** Define the syntax of the operations and the parameters to each. This is the signature part of the formal specification. You should update the informal specification at this stage if necessary.

Axiom definition: define the semantics of the operations by describing what conditions are always true for different operation combinations.

Q4 (b) What is Pair Programming? What are the advantages of pair programming?

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 th refactoring so they are likely to support the process.

Q4 (c) What do you mean by RAD? What are the tools included in RAD environment?

Answer

Rapid application development (RAD) techniques evolved from so-called fourth-generation languages in the 1980s and are used for developing applications that are data-intensive. Consequently, they are usually organised as a set of tools that allow data to be created, searched, displayed and presented in reports.

The tools that are included in RAD environment are:

1. ***A database programming language*** that embeds knowledge of the database structures and includes fundamental database manipulation operations. SQL is the standard database programming language. The SQL commands may be input directly or generated automatically from forms filled in by an end-user.
2. ***An interface generator,*** which is used to create forms for data input and display.
3. ***Links to office applications*** such as a spreadsheet for the analysis and manipulation of numeric information or a word processor for report template creation.
4. ***A report generator,*** which is used to define and create reports from information in the database.

RAD systems are successful because, there is great deal of commonality across business applications. These applications are often concerned with updating a database and producing reports from the information in the database. Standard forms are used for input and output. RAD systems are geared towards producing interactive applications that rely on abstraction information from an organisational database, presenting it to end-users on their terminal or workstations, and updating the database with changes made by users.

Q5 (a) What are the non-functional system requirements that may be chosen for an application for a particular style and structures?

Answer

The particular style and structure chosen for an application depend on the following non-functional system requirements:

- ***Performance:*** If performance is a critical requirement, the architecture should be designed to localise critical operations within a small number of subsystems, with as little communication as possible between these sub-systems. This may mean using relatively large-grain rather fine-grain components to reduce component communications.
- ***Security:*** If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and with a high level of security validation applied to these layers.
- ***Safety:*** If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single sub-system or in a small number of sub-systems. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems.
- ***Availability:*** If availability is a critical requirement, the architecture should be designed to include redundant components and so that it is possible to replace and update components without stopping the system.

Maintainability: If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be

changed. Producers of data should be separated from consumers and shared da structures should be avoided.

Q5 (b) Write the advantages and disadvantages of broadcast model approach?

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.

Q5 (c) Discuss the important characteristics of distributed approach to system development?

Answer

The important characteristics of distributed approach to system development are as follow:

- ***Resource sharing:*** A distributed system allows the sharing of hardware and software resources – such as disks, printers, files, and compilers – that are associated with computers on a network.
- ***Openness:*** Distributed systems are normally open systems, which mean they are designed around standard protocols that allow equipment and software from different vendors to be combined.
- ***Concurrency:*** In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.
- ***Scalability:*** In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability. If many new computers are added, then the network capacity may be inadequate.
- ***Fault tolerance:*** The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures. In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only tends to occur when there is a network failure.

Q6 (a) What are the key factors that one should consider while planning for software reuse?

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 focu 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.

Q6 (b) Define component? How components are different from objects?

Answer

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Components are usually developed using an object-oriented approach, but they differ from objects in a number of important ways:

1. ***Components are deployable entities:*** Hence they are not compiled into an application program but are installed directly on an extension platform. The methods and attributes defined in their interfaces can then be accessed by other components.
2. ***Components do not define types:*** A class definition defines an abstract data type and objects are instances of that type. A component is an instance, not a template that is used to define an instance.
3. ***Component implementations are opaque:*** Components are, in principle at least, completely defined by their interface specification. The implementation is hidden from component users. Components are often delivered as binary units so the buyer of the component does not have access to the implementation.
4. ***Components are language-independent:*** Object classes have to follow the rules of a particular object-oriented programming language and, generally, can only interoperate with other classes in that language. Although components are usually implemented using

object-oriented languages such as Java, you can implement them in non-object-oriented programming languages.

5. ***Components are standardised:*** Unlike object classes that you can implement in any way, components must conform to some component model that constrains their implementation

Q7 (a) 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.

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 retr systems

Q7 (b) For small and medium sized systems, what is the software engineering approach to develop tool, techniques and methods that leads to the production of fault-free software?

Answer

A goal of software engineering research has been to develop tools, techniques and methods that lead to the production of fault-free software. Fault-free software is software that conforms exactly to its specification. This does not mean that the software will never fail. There may be errors in the specification that are reflected in the software, or the users may understand or misuse the software system. However, eliminating software faults certainly has a huge impact on the number of system failures.

For small and medium-sized systems, our software engineering techniques are such that it is probably possible to develop fault-free software. To achieve this goal, you need to use a range of software engineering techniques:

1. ***Dependable software processes:*** The use of dependable software process with appropriate verification and validation activities is essential if the number of faults in a program is to be minimised, and those that do slip through are to be detected.
2. ***Quality management:*** The organisation developing the system must have a culture in which quality drives the software process. The culture should encourage programmers to write bug-free programs. Design and development standards should be established, and procedures should be in place to check that these have been followed.
3. ***Formal specification:*** There must be a precise system specification that defines the system to be implemented. Many design and programming mistakes are a result of misinterpretation of an ambiguous or poorly worded specification.
4. ***Static verification:*** Static verification techniques, such as the use of static analysers, can find anomalous program features that could be faults. Formal verification, based on the system specification, may also be used.
5. ***String typing:*** A strongly typed programming language such as Java or Ada must be used for development. If the language has strong typing, the language compiler can detect many programming errors before they can be introduced into the delivered program.
6. ***Safe programming:*** Some programming language constructs are more complex and error-prone than others, and you are more likely to make mistakes if you use them. Safe programming means avoiding or at least minimising the use of these constructs.
7. ***Protected information:*** An approach to software design and implementation based on information hiding and encapsulation should be used. Object-oriented languages such as Java obviously satisfy this condition. The development of programs that are designed for readability and understandability should be encouraged.

Q8 (a) What do you mean by software inspection? Write major advantages of inspection over testing?

Answer

Software inspection is a static V & V process in which a software system is reviewed to find errors, omissions and anomalies. Generally, inspections focus on source code, but any readable representation of the software such as its requirements or a design model can be inspected. When

you inspect a system, you use knowledge of the system, its application domain and th programming language or design model to discover errors.

There are three major advantages of inspection over testing:

- During testing, errors can mask (hide) other errors. Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.
- Incomplete version of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialised test harnesses to test the parts that are available. This obviously adds to the system development costs.

As well as searching for program defects, an inspection can also consider broader quality attributes of a program such as compliance with standards, portability and maintainability. You can look for inefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.

Q8 (b) What are Test Principles and what are the attributes of a "good" test?

Answer

Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing.

• All tests should be traceable to customer requirements. As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

• Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

• The Pareto principle applies to software testing. Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

• Testing should begin "in the small" and progress toward testing "in the large." The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

• Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

• To be most effective, testing should be conducted by an independent third party. By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing).

Following are the Attributes of a good test

1. A good test has a high probability of finding an error. To achieve this goal, the tester mu understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.
2. A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different). For example, a module of the *SafeHome* software is designed to recognize a user password to activate and deactivate the system. In an effort to uncover an error in password input, the tester designs a series of tests that input a sequence of passwords. Valid and invalid passwords (four numeral sequences) are input as separate tests. However, each valid/invalid password should probe a different mode of failure. For example, the invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, say 1235, would have the same purpose as 1234 and is therefore redundant. However, the invalid input 8081 or 8180 has a subtle difference, attempting to demonstrate that an error exists for passwords "close to" but not identical with the valid password.
3. A good test should be "best of breed". In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

5. A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

Q8 (c) Describe two metrics that have been used to measure software productivity?

Answer

Productivity estimates are usually based on measuring attributes of the software and dividing this by total effort required for development. There are two types of metric that have been used:

1. ***Size-related metrics:*** These are related to the size of some output form an activity. The most commonly used size-related metric is lines of delivered source code. Other metrics that may be used are the number of delivered object code instructions or the number of pages of system documentation.
2. ***Function-related metrics:*** These are related to overall functionality of the delivered software. Productivity is expressed in terms of the amount of useful functionality produced in some given time. Function points and object points are the best-known metrics of this type.

Q9 (a) What is Software configuration management? Explain the major tasks of SCM?

Answer Page Number 507 of Text-Book

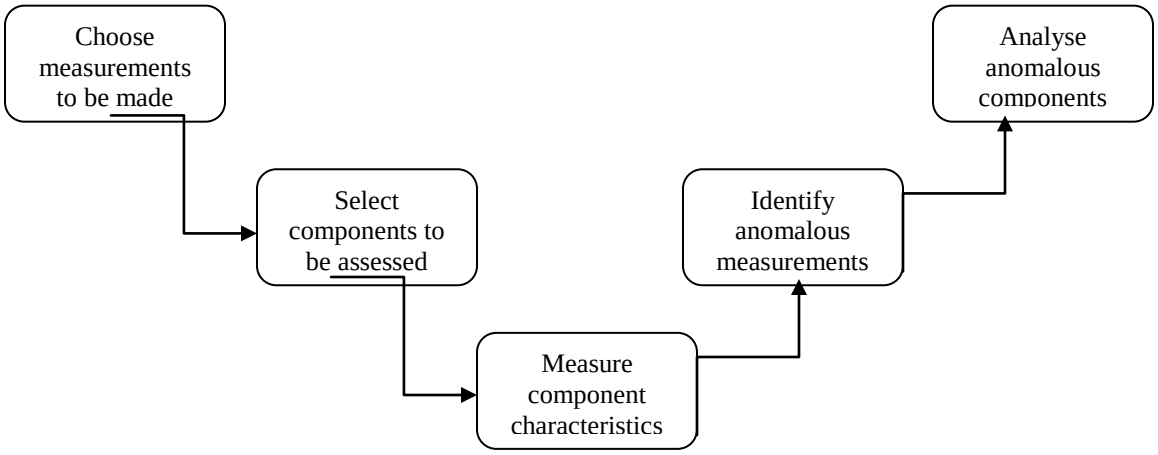
Q9 (b) With the help of a figure, explain the key stages of software measurement process which is a part of a quality control process?

Answer

A software measurement process that may be part of a quality process is shown in figure 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.



The process of product measurement

- ***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 quali problems

Text Book

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