UNIT – V
SOFTWARE QUALITY AND TESTING

Software Quality Assurance

1. What are the tasks, goals of SQA?

A. **Software Quality Assurance:** Software quality assurance is composed of a variety of tasks associated with two different constituencies—the *software engineers* who do technical work and an *SQA group* that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

SQA Tasks: The Software Engineering Institute recommends a set of SQA actions that address quality assurance planning, oversight, record keeping, analysis, and reporting.

- *Prepares an SQA plan for the projects:* Quality assurance actions performed by the software engineering team and the SQA group are governed by the plan. The plan identifies evaluations to be performed, audits and reviews to be conducted.
- *Participates in the development of the project's software process description:* The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, external standards, and other parts of the software project plan.
- *Reviews software engineering activities to verify compliance with the defined software process:* The SQA group identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
- *Audits designated software work products to verify compliance with those defined as part of the software process:* The SQA group reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made; and periodically reports the results of its work to the project manager.
- *Ensures that deviations in software work and work products are documented and handled according to a documented procedure:* Deviations may be encountered in the project plan, process description, applicable standards, or software engineering work products.
- *Records any noncompliance and reports to senior management:* Noncompliance items are tracked until they are resolved.

SQA Goals: The SQA actions described in the preceding section are performed to achieve a set of pragmatic goals:

➡ *Requirements quality:* The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

➡ *Design quality:* Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements.

➡ *Code quality:* Source code and related work products must conform to local coding standards and exhibit characteristics that will facilitate maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

➡ *Quality control effectiveness:* A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

****

2. What are the quality metrics?

A. Software Quality Metrics: A measure of some property of a piece of software or its specifications. Basically, as applied to the software product, a software metric measures a characteristic of the software. Some common software metrics are:-

- Source lines of code.
- Cyclomatic complexity, is used to measure code complexity.
- Function point analysis (FPA), is used to measure the size (functions) of software.
- Bugs per lines of code.
- Code coverage, measures the code lines that are executed for a given set of software tests.
- Cohesion, measures how well the source code in a given module work together to provide a single function.
- Coupling, measures how well two software components are *data* related, i.e. how independent they are.

The SATC (Software Assurance Technology Center) Software Quality Model which includes Goals and Metrics as well as the software attributes:

GOALS	ATTRIBUTES	METRICS
	Ambiguity	Number of Weak Phrases. Number of Optional Phrases.
	Completeness	Number of To Be Determined (TBDs) and To be Added (TBAs).

Requirements Quality	Understandability	Document Structure. Readability Index.
	Volatility	Count of Changes / Count of Requirements. Life cycle stage when the change is made.
	Traceability	Number of software requirements not traced to system requirements. Number of software requirements not traced to code and tests.
Product (Code) Quality	Structure/Architecture	Logic complexity. GOTO usage. Size.
	Maintainability	Correlation of complexity/size.
	Reusability	Correlation of complexity/size.
	Internal Documentation	Comment Percentage.
	External Documentation	Readability Index.
Implementation Effectivity	Resource Usage	Staff hours spent on life cycle activities.
	Completion Rates	Task completions. Planned task completions.
Testing Effectivity	Correctness	Errors and criticality. Time of finding of errors. Time of error fixes. Code Location of fault.

****

3. Explain Software reliability?

A. The reliability of a computer program is an important element of its overall quality. Software reliability can be measured directly and estimated using historical and developmental data. ***Software reliability*** is defined in statistical terms as "*the probability of failure-free operation of a computer program in a specified environment for a specified time*".

Whenever software reliability is discussed, a question arises: What is *failure*? Failure is nonconformance to software requirements. One failure can be corrected within seconds, while another requires weeks or even months to correct. The correction of one failure may in fact result in the introduction of other errors that ultimately result in other failures.

Measures of Reliability and Availability: All software failures can be traced to design or implementation problems; if we consider a computer-based system, a simple measure of reliability is *mean-time-between-failure* (MTBF):

$$MTBF = MTTF + MTTR$$

Where the acronyms MTTF and MTTR are *mean-time-to-failure* and mean-time-to-repair respectively. In addition to reliability measure, we should also develop a measure of availability. Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as:

$$\text{Availability} = \frac{MTTF}{MTTF + MTTR} \times 100\%$$

The MTBF reliability measure is equally sensitive to MTTF and MTTR.

Software Safety: Software safety is a software quality assurance activity that focuses on the identification and assessment of hazards (risks) that may affect software negatively and cause an entire system to fail. Once hazards are identified and analyzed, safety-related requirements can be specified for the software. That is, the specification can contain a list of undesirable events and the desired system responses to these events.

****

Software Testing

4. Explain Software testing fundamentals?

A. Software testing is a process of executing a program or application with the intent of finding the software bugs. It can also be stated as the process of validating and verifying that a software program or application or product: Meets the business and technical requirements that guided its design and development.

Testing Objectives: Glen Myers states a number of testing objectives;

- Software testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as – yet – undiscovered error.
- A successful is one that uncovers errors in the software.

Testing Principles: The following are a set of testing principles that guide software testing:

1. All tests should be traceable to customer requirements.
2. Tests should be planned long before testing begins.
3. The Pareto principle (80% of all errors uncovered during testing) applies to software testing.
4. Testing should begin *in the small* and progress toward testing *in the large*.
5. Exhaustive testing is not possible.
6. To be most effective, testing should be conducted by an independent third party.

Testability: Software testability is simply how easily can be tested. The following are the characteristics that lead to testable software;

- Operability – "The better it works the more efficiently it can be tested".
- Observability – "What you see is what you test".
- Controllability – "The better we can control the software, the more the testing can be automated and optimized".
- Decomposability – "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting".
- Simplicity – "The less there is to test, the more quickly we can test it".
- Stability – "The fewer the changes, the fewer the disruptions to testing".

- **Understandability** – "The more information we have, the smarter we will test".

The following are the attributes of good test:

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex.

****

5. Explain White – Box testing?

A. White box testing is also called as glass box testing is a test case design method that uses the control structure of the procedural design to derive test cases. White – box testing derives the test cases that –

(1) Guarantee that all individual paths within the module have been exercised at least once.

(2) Exercise all logical decisions on their true or false sides.

(3) Executes all loops at their boundaries and within their operational bounds.

(4) Exercise internal data structures to ensure their validity.

The reasons for choosing white box testing are:

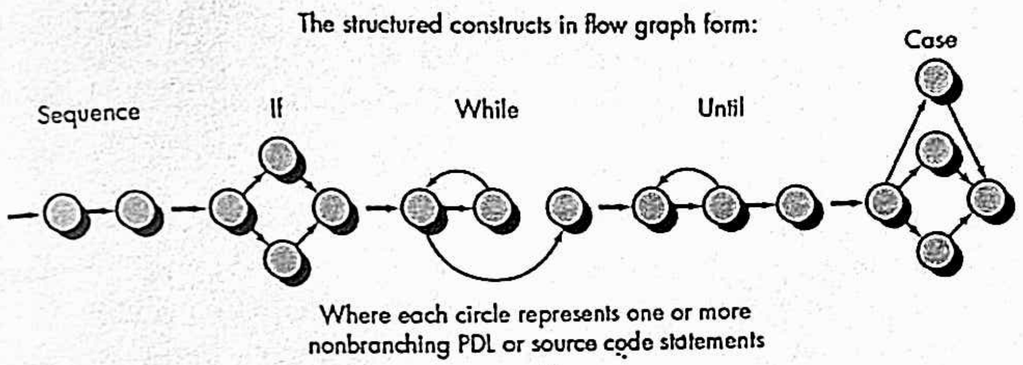
- Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- Typological errors are random.
- We often believe that logical path is not likely executed; in fact, it may be executed on a regular basis.

****

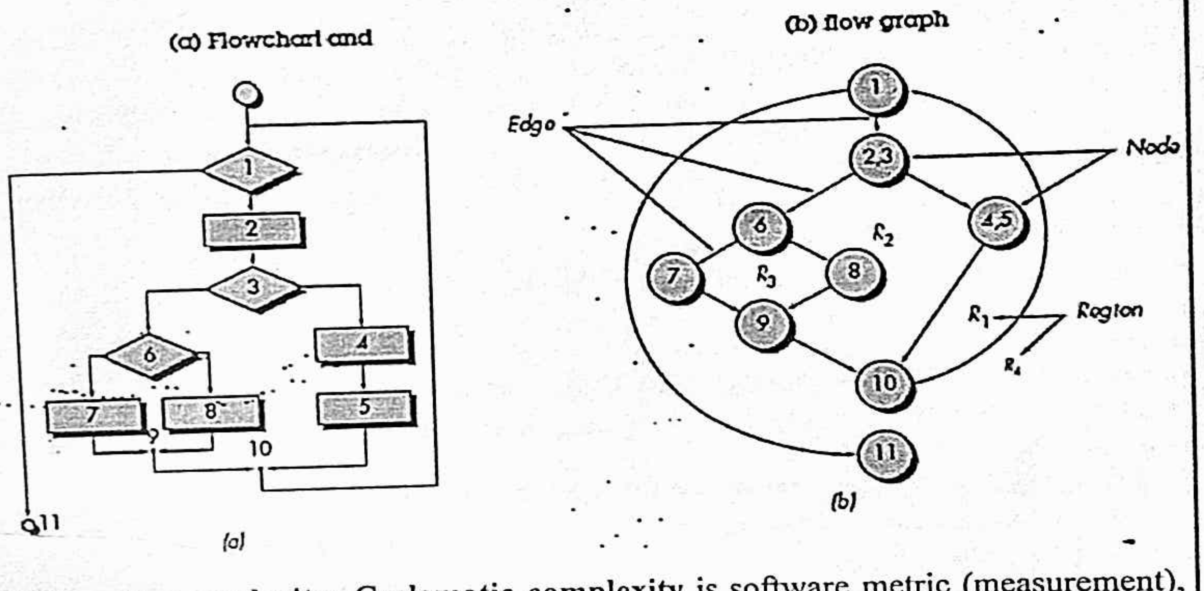
6. Explain Basis path testing?

A. Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

__**Flow graph notation:**__ The flow graph depicts logical control flow using the following notation:


The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

The following maps the flowchart into a corresponding flow graph. In the flow graph, each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements.



Cyclomatic complexity: Cyclomatic complexity is software metric (measurement), used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code.

Independent Program paths: It is any path through the program that introduces at least one new set of processing statements or a new condition. In the context of flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the above flow graph are:

Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path
1-2-3-4-5-10-1-2-3-6-8-9-10-1-11
is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Cyclomatic complexity has a foundation in graph theory and provides you with extremely useful software metric. Complexity is computed in one of three ways:

- The number of regions of the flow graph corresponds to the cyclomatic complexity.
- Cyclomatic complexity V(G) for a flow graph G is defined as:

  V (G) = E – N + 2

  Where E is the number of edges, N is the number of flow graphs.
- Cyclomatic complexity V(G) for a flow graph G is defined as:

  V (G) = P + 1

  Where P is the number of predicate nodes contained in the flow graph G.

The cyclomatic complexity for the above graph is,
1] V (G) = 11 edges – 9 nodes + 2 = 4
2] V (G) = 3 predicate nodes + 1 = 4
Therefore, the cyclomatic complexity of the above flow graph is 4.

<u>Deriving test cases:</u> The basis path testing method can be applied to procedural design or to source code. The following steps can be applied to derive the basis set:

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

****

7. Explain control structure testing?

A. Control structure testing is a white-box testing technique which improves the quality of it.

Condition Testing: *Condition testing* is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E1 <relational-operator> E2

Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, <=, =, !=, >, >=. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

Types of errors in a condition include:

1] Boolean operator error
2] Boolean variable error
3] Boolean parenthesis error
4] Relational operator error
5] Arithmetic expression error

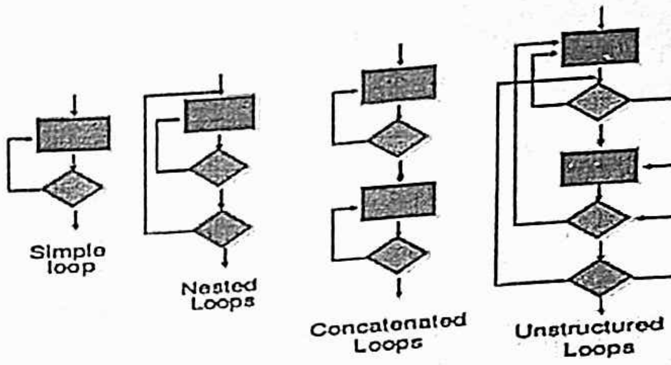
Data Flow Testing: The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

DEF(S) = {X| statement S contains a definition of X}
USE(S) = {X| statement S contains a use of X}

If statement S is an *if* or *loop* statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be *live* at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

Loop Testing: Loops are the cornerstone for the vast majority of all algorithms implemented in software.



Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

- **Simple Loop:** The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.
    - Skip the loop entirely.
    - Only one passes through the loop.
    - Two passes through the loop.
    - m passes through the loop where m<n.
    - n-1, n, n+1 passes through the loop.

- **Nested Loop:** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. The following approach will help to reduce the number of tests:
    - Start at the inner loop. Set all other loops to minimum values.
    - Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values.
    - Conduct tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to typical values.
    - Continue until all loops have been tested.

- **Concatenated Loop:** Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

- **Unstructured Loop:** Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

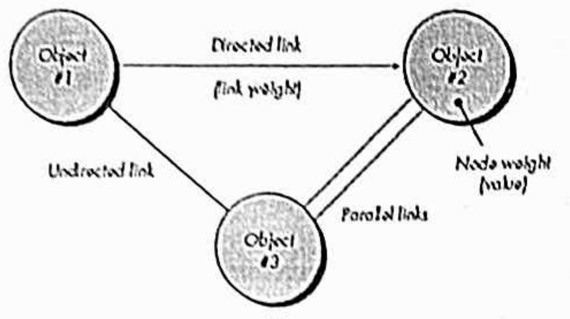
****

8. Explain black box testing?

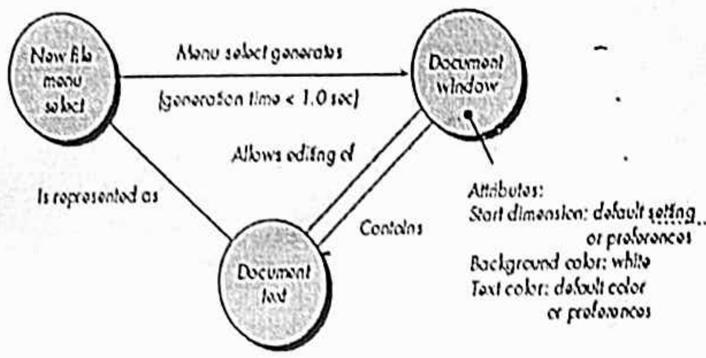
A. Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.

Black-box testing attempts to find errors in the following categories:
(1) Incorrect or missing functions,
(2) Interface errors,
(3) Errors in data structures or external database access,
(4) Behavior or performance errors, and
(5) Initialization and termination errors.
Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.

<u>Graph – Based Testing Methods:</u> The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another".



(A)



(B)

Reference (A) – Nodes are represented as circles connected by links that take a number of different forms. A <u>*directed link*</u> (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link*, also called a *symmetric* *link*, implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

Reference (B) –

Object #1 = **newFile (menu selection)**

Object #2 = **documentWindow**

Object #3 = **documentText**

Referring to the figure, a menu select on **newFile** generates a **document window.** The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **newFile menu selection** and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.**

Graph-based testing begins with the definition of all nodes and node weights. Once nodes have been identified links and link weights should be established. The *transitivity* of relationships, *symmetry* of relationships, and *reflexive* relationships are tested here.

Equivalence Partitioning: *Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. Equivalence classes may be defined according to the following guidelines:

i. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
ii. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
iii. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
iv. If an input condition is Boolean, one valid and one invalid class are defined.

Boundary Value Analysis: A greater number of errors occur at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

i. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.

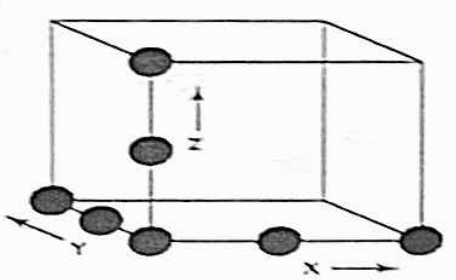
ii. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

iii. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

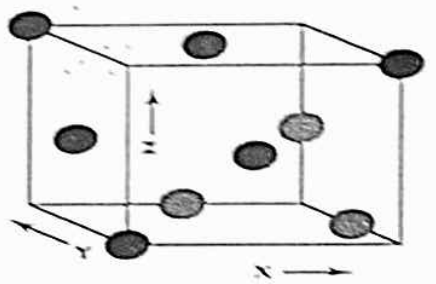
iv. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

<u>**Comparison Testing:**</u> In aircrafts, automobile banking systems in which the reliability of software is absolutely critical. In such applications redundant hardware and software are often used to minimize the possibility of errors. When redundant software developed, then independent versions of applications are developed using the same specifications. Each version is tested independently with the same values for identical results. Then all the versions are executed in parallel with real time comparison of results to ensure consistency. These independent versions form the basis of black box testing technique called *comparison testing or black box testing.*

<u>**Orthogonal Array Testing:**</u> Orthogonal array testing can be applied to problems in which the input domain is relatively small. The orthogonal array testing method is particularly useful in finding *region faults* —an error category associated with faulty logic within a software component. The difference between *orthogonal array testing* and more conventional *"one input item at a time"* approaches, consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are $3^3=27$ possible test cases.



One input item at a time



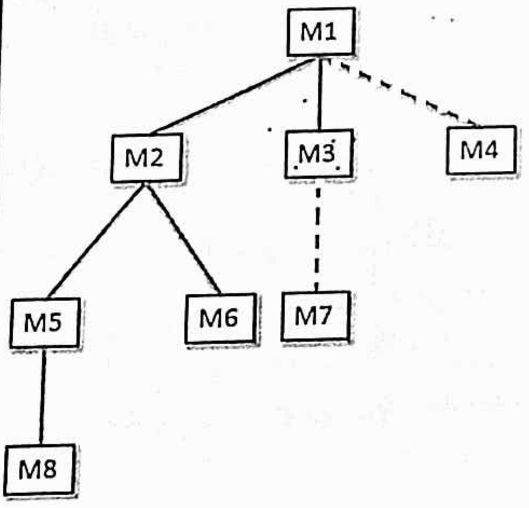
L9 orthogonal array

To illustrate the L9 orthogonal array, consider the send function for a fax application. Three parameters, P1, P2, and P3, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:
P1 = 1, send it now
P1 = 2, send it one hour later
P1 = 3, send it after midnight
P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions. If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3) would be specified: (1, 1, 1), (2, 1, 1), (3, 1, 1), (1, 2, 1), (1, 3, 1), (1, 1, 2), (1, 1, 3) and so on.
****

9. Explain Integration testing?

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

Top – Down integration: Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure in either a *depth-first* or *breadth-first* manner. The *depth first integration* integrates vertical modules where as *breadth first integration* integrates horizontal modules.

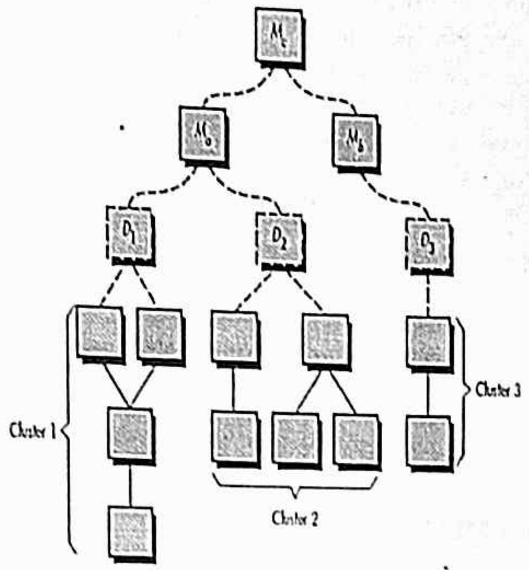


1] The integration process is performed in a series of five steps:
2] The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
3] Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

4] Tests are conducted as each component is integrated.
5] On completion of each set of tests, another stub is replaced with the real component.
6] Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced

Bottom – Up integration: *Bottom-up integration* testing, as its name implies, begins construction and testing with *atomic modules*. A *bottom-up integration* strategy may be implemented with the following steps:

1] Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
2] A *driver* (a control program for testing) is written to coordinate test case input and output.
3] The cluster is tested.
4] Drivers are removed and clusters are combined moving upward in the program structure.



Regression Testing: Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the re-execution of some subset of tests that have already been conducted.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:
1] A representative sample of tests that will exercise all software functions.
2] Additional tests that focus on software functions that are likely to be affected by the change.
3] Tests that focus on the software components that have been changed.

Smoke Testing: *Smoke Testing* is an integration testing approach used when *"Shrink – Wrapped"* software products are being developed. These products are designed to pacing mechanism for time – critical projects. The smoke test encompasses the following activities:

1. Software components that are translated into code are integrated into "*a build*". A build includes data files, libraries, reusable modules and engineered components.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function.
3. The build is integrated with other builds and entire product is smoke tested daily.

****

10. Explain Validation testing?

A. *Validation testing* begins after integration testing, when individual components are tested, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. Validation succeeds when software functions can be reasonably expected by the customer.

Reasonable specifications are defined in the *Software Requirements Specification – document.* It contains the section *validation criteria.* The information in validation criteria is basis for validation testing.

Validation Test Criteria:

Software validation is achieved through a series of tests. A test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met.

After each validation test, one of two possible conditions exists:

1. The function or performance characteristic conforms to specification and is accepted. (OR)
2. A deviation from specification is uncovered and a deficiency list is created.

Alpha and Beta Testing: Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

- The *Alpha* test is conducted at the developer's site by a customer. This test is conducted in a controlled environment.
- The *Beta* test is conducted at one or more customer sites by the end user of the software.

****

11. Explain System testing?

A. *System testing* is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.

<u>**Recovery Testing:**</u> Many computer-based systems must recover from faults and resume processing within a pre specified time. In some cases system must be fault tolerant; that is processing faults must not cause overall system function to cease.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

<u>**Security Testing:**</u> Any computer-based system that manages sensitive information which causes illegal penetration (gain access by force). Penetration spans a broad range of activities:

- Hackers who attempt to penetrate systems for sport,
- Dissatisfied employees who attempt to penetrate for revenge,
- Dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system which protects it from improper penetration.

<u>**Stress Testing:**</u> Earlier software testing steps resulted in thorough evaluation of normal program functions and performance. A variation of stress testing is a technique called *sensitivity testing*. In some situations (the most common occur in mathematical algorithms), a very small range of data may cause extreme and even incorrect processing. Sensitivity testing attempts to uncover these errors.

<u>**Performance Testing:**</u> *Performance testing* is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

****

Reverse and Re-engineering

12. Explain Reverse engineering?

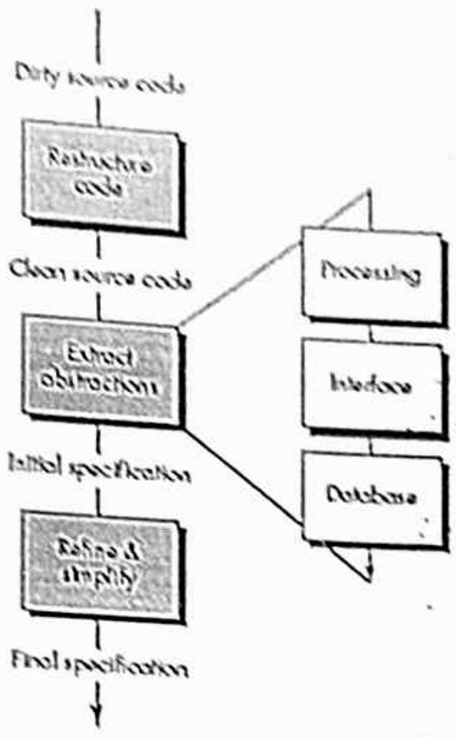
A. *Reverse engineering* is the reproduction of another manufacturer's product following detailed examination of its construction or composition.

Reverse engineering can extract design information from source code, but the *abstraction level*, the *completeness* of the documentation, and the *directionality* of the process.

The *abstraction level* of a reverse engineering process and the tools can be extracted from source code. The abstraction level should be as high as possible. As the abstraction level increases, you are provided with information (program and data structure information, object models, data or control flow models, and entity relationship models) that will allow easier understanding of the program.

The *completeness* of a reverse engineering process refers to the level of detail that is provided at an abstraction level.

If the *directionality* of the reverse engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a reengineering tool that attempts to restructure or regenerate the old program.



__Reverse engineering to understand data:__ Reverse engineering of data occurs at different levels of abstraction and is the first reengineering task. At the *program level*, internal program data structures must often be reverse engineered. At the *system* level, *global* data structures are often reengineered.

- **Internal Data Structures:** Reverse engineering techniques for internal program data focus on the definition of classes of objects. The data organization within the code identifies abstract data types. For example, record structures, files, lists, and other data structures often provide an initial indicator of classes.
- **Data Structure:** Regardless of its logical organization and physical structure, a database allows the definition of data objects and supports some method for establishing relationships among the objects.

__Reverse engineering to understand processing:__ Reverse engineering to understand processing begins with an attempt to understand and then extract procedural abstractions represented by the source code. To understand procedural abstractions, the code is analyzed at varying levels of abstraction: system, program, component, pattern, and statement.

Reverse engineering user interfaces: GUIs have become required for computer based products and systems of every type. Therefore, the redevelopment of user interfaces has become one of the most common types of reengineering activity. But before a user interface can be rebuilt, reverse engineering should occur.

****

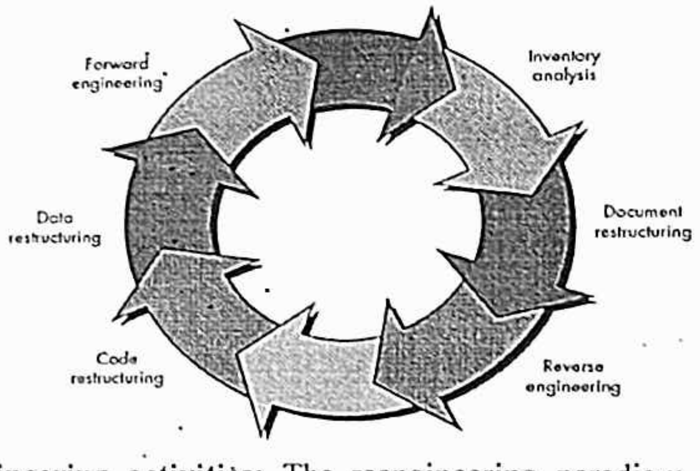
13. Explain Software Reengineering?

A. An application was served the business needs of a company for 10 or 15 years, during that time it has to be corrected, adapted and enhanced many times. ***Software*** *maintenance* is so difficult because 60% of the software is enhanced every time. Software maintenance is described by four activities:

- Corrective Maintenance
- Adaptive Maintenance
- Perfective (or) Enhancement Maintenance
- Preventive Maintenance (or) Reengineering

A software reengineering process model:

Reengineering of information systems is an activity that will absorb information technology resources for many years.



Software reengineering activities: The reengineering paradigm shown in above figure is a cyclical model. This means that each of the activities presented as a part of the paradigm may be revisited.

- **Inventory analysis:** Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. The inventory should be revisited on a regular cycle.

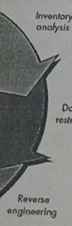
GUIs have become required for computer
pe. Therefore, the redevelopment of user
mmon types of reengineering activity. But
e engineering should occur.

****

eeds of a company for 10 or 15 years,
ed and enhanced many times. *Software*
the software is enhanced every time.
vities:

g

is an activity that will absorb



g paradigm shown in above
ctivities presented as a part of

should have an inventory of
e than a spreadsheet model
escription (e.g., size, age,
The inventory should be

- **Document Restructuring:**
  - Creating documentation is far too time consuming.
  - Documentation must be updated, but your organization has limited resources.
  - The system is business critical and must be fully redocumented.
- **Reverse Engineering:** The term reverse engineering has its origins in the hardware world. A company disassembles a competitive hardware product in an effort to understand its competitor's design and manufacturing "secrets."

  Reverse engineering for software is quite similar. Reverse engineering is a process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program.
  - ***Code Restructuring:*** The most common type of reengineering is code restructuring. The source code is analyzed using restructuring tool, and the internal code documentation code is updated.
  - ***Data Restructuring:*** Data restructuring is full scale reengineering activity. It begins with a reverse engineering. Current data architecture is dissected and necessary data models are defined.
  - ***Forward Reengineering:*** Applications would be rebuilt using an automated "reengineering engine." The old program would be fed into the engine, analyzed, restructured, and then regenerated in a form that exhibited the best aspects of software quality.

****

CASE Tools

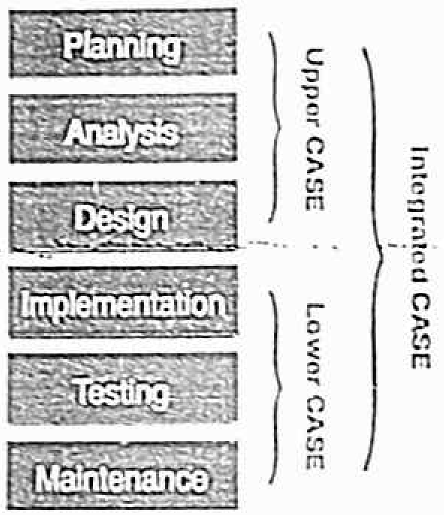
14. Explain CASE tools?

A. CASE stands for Computer Aided Software Engineering. It means development and maintenance of software projects with help of various automated software tools. CASE tools are set of software application programs, which are used to automate SDLC activities. CASE tools are used by software project managers, analysts and engineers to develop software system.

There are number of CASE tools available to simplify various stages of Software Development Life Cycle such as Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools etc.

__Components of CASE tools:__ CASE tools can be broadly divided into the following parts based on their use at a particular SDLC stage:

- Central Repository - CASE tools require a central repository, which can serve as a source of common, integrated and consistent information. Central repository is a central place of storage where product specifications, requirement documents, related reports and diagrams, other useful information regarding management is stored. Central repository also serves as data dictionary.



- Upper Case Tools - Upper CASE tools are used in planning, analysis and design stages of SDLC.
- Lower Case Tools - Lower CASE tools are used in implementation, testing and maintenance.
- Integrated Case Tools - Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

__Types of CASE tools:__ The following are some of the CASE tools:

1. *Diagram tools:* These tools are used to represent system components, data and control flow among various software components and system structure in a graphical form.

2. *Process Modeling Tools:* Process modeling is method to create software process model, which is used to develop the software. Process modeling tools help the managers to choose a process model or modify it as per the requirement of software product. For example, EPF Composer

3. *Project Management Tools:* These tools are used for project planning, cost and effort estimation, project scheduling and resource planning.

   Project management tools help in storing and sharing project information in real-time throughout the organization. For example, Creative Pro Office, Trac Project, Basecamp.

4. *Analysis Tools:* These tools help to gather requirements, automatically check for any inconsistency, inaccuracy in the diagrams, data redundancies or erroneous omissions. For example, Accept 360, Accompa, Case Complete for requirement analysis, Visible Analyst for total analysis.

5. *Design Tools:* These tools help software designers to design the block structure of the software, which may further be broken down in smaller modules using refinement techniques. These tools provides detailing of each module and interconnections among modules. For example, Animated Software Design

6. *Programming Tools:* These tools consist of programming environments like IDE (Integrated Development Environment), in-built modules library and simulation tools. These tools provide comprehensive aid in building software product and include features for simulation and testing. For example, Cscope to search code in C, Eclipse.

7. *Maintenance Tools:* Software maintenance includes modifications in the software product after it is delivered. Automatic logging and error reporting techniques, automatic error ticket generation and root cause Analysis are few CASE tools, which help software organization in maintenance phase of SDLC. For example, Bugzilla for defect tracking, HP Quality Center.

****

15. Explain Project Management tools?

A. Project management is one of the high-responsibility tasks in modern organizations. In order to execute a project successfully, the project manager or the project management team should be supported by a set of tools. These tools can be specifically designed tools or regular productivity tools that can be adopted for project management work.

- **Project Plan:** All the projects that should be managed by a project manager should have a project plan. The project plan details many aspects of the project to be executed.First of all,it describes the approach or strategy used for addressing the project scope and project objectives. The resource allocation and delivery schedule are other two main components of the project plan.
- **Milestone Checklist:** This is one of the best tools the project manager can use to determine whether he is on track in terms of the project progress. The project manager can use a simple Excel template to do this job. The milestone checklist should be a live document that.should be updated once or twice a week.
- **Gantt chart:** Gantt charts are universally used for any type of project from construction to software development. Although deriving a Gantt chart looks quite easy, it is one of the most complex tasks when the project is involved in hundreds of activities.
- **Project management softwares:** MS Project can be used as a standalone tool for tracking project progress or it can be used for tracking complex projects distributed in many geographical areas and managed by a number of project managers.
- **Project reviews:** In project reviews, the project progress and the adherence to the process standards are mainly considered. Usually, project reviews are accompanied by project audits by a 3rd party (internal or external).
- **Delivery reviews:** Usually, a 3rd party team or supervisors (internal) conduct the delivery review and the main stakeholders of the project delivery do participate for this event.
- **Score Cards:** When it comes to performance of the project team, a scorecard is the way of tracking it. Every project manager is responsible of accessing the performance of the team members and reporting it to the upper management and HR.

****

16. Explain analysis and design tools?

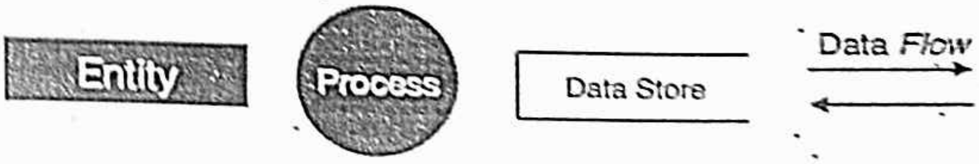
A. Software analysis and design includes all activities, which help the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable,and understandable documents, to which a computer has nothing to do. The following are few analysis and design tools used by software designers:

1) Data Flow Diagram: Data flow diagram is graphical representation of flow of data in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data.

Types of DFD: Data Flow Diagrams are either Logical or Physical.

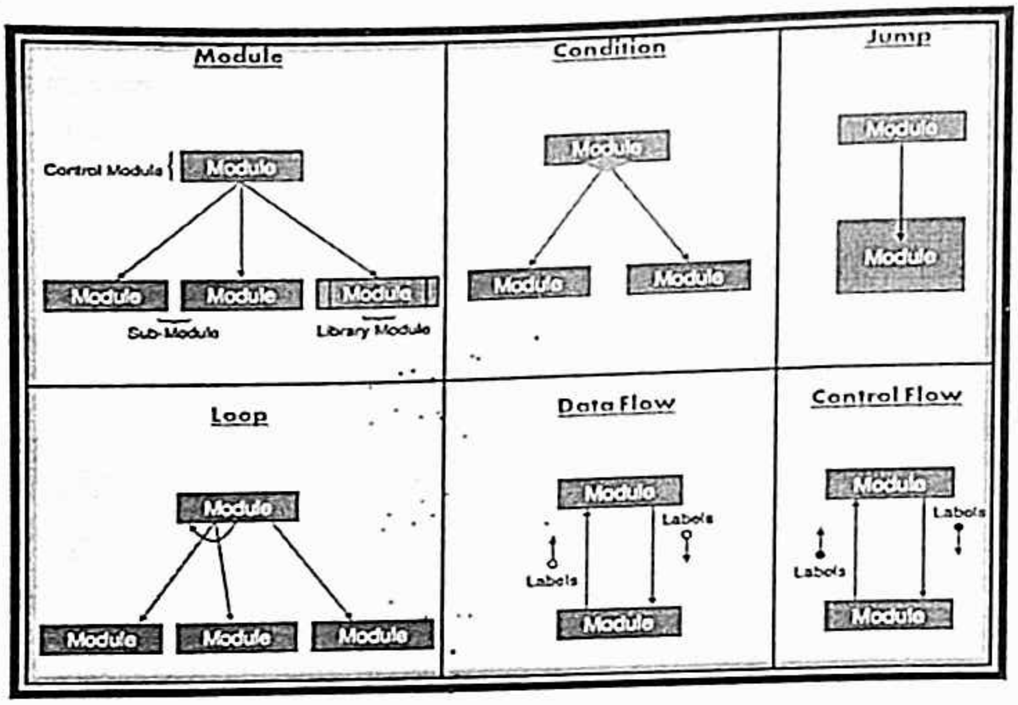
- **Logical DFD** - This type of DFD concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.
- **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

DFD Components: DFD can represent Source, destination, storage and flow of data using the following set of components -



- **Entities** - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.
- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
- **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

2) Structure Charts: Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD. Here are the symbols used in construction of structure charts -

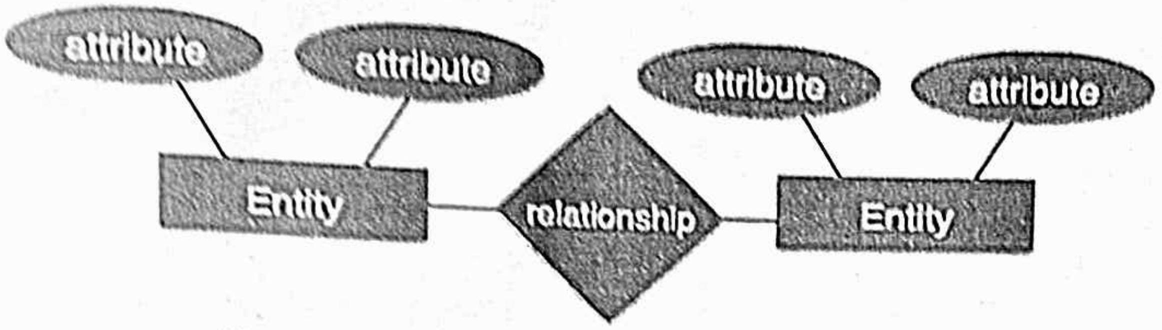


3) HIPO Diagram: HIPO (Hierarchical Input Process Output) diagram represents the hierarchy of modules in the software system. Analyst uses HIPO diagram in order to obtain high-level view of system functions. It decomposes functions into sub-functions in a hierarchical manner. Both parts of HIPO diagram, Hierarchical presentation and IPO Chart are used for structure design of software program as well as documentation of the same.

4) PSEUDO Code: Pseudo code is written more close to programming language. It may be considered as augmented programming language, full of comments and descriptions. Pseudo code avoids variable declaration but they are written using some actual programming language's constructs, like C, FORTRAN, and Pascal etc.

5) Decision Tables: A Decision table represents conditions and the respective actions to be taken to address them, in a structured tabular format. It is a powerful tool to debug and prevent errors. It helps group similar information into a single table and then by combining tables it delivers easy and convenient decision-making.

6) Entity Relationship Model: Entity-Relationship model is a type of database model based on the notion of real world entities and relationship among them. We can map real world scenario onto ER database model. ER Model creates a set of entities with their attributes, a set of constraints and relation among them. ER Model is best used for the conceptual design of database. ER Model can be represented as follows:


attribute
attribute
attribute
attribute
Entity
relationship
Entity

****

[illegible]