Chapter 2

Project Management and Maintenance

LEARNING OBJECTIVES

- *Project management*
- *Software design*
- *Modeling component level design*
- *SRS*
- *Software testing*
- *White-box testing*
- *Black box testing*
- *Implementation maintenance*
- *Software quality assurance*
- *Software Re-engineering*
- *COCOMO MODEL*

Project Management

Project management is a technique used to ensure successful completion of a project by the project managers.

The functions included in project management are:

- Estimating resource requirements
- Scheduling tasks and events
- Providing training and site preparation
- Selecting qualified staff and supervising their work
- Monitoring the projects program
- Documenting
- Periodic evaluation
- Contingency planning

Project management involves planning, organization and control projects. It uses tools and software packages for planning and managing projects.

Project planning involves plotting project activities against time frame.

Project Planning Tools

- Tools used during software planning
- Helps the top level managers to take critical decisions during planning stage

Gantt Charts

This activity scheduling method introduced in 1914 by Henry L. Gantt, uses horizontal bars to show the duration of actions or tasks. The left end marks the beginning of the task and the right end its finish. Earlier tasks appear in the upper left and later ones in the lower right.

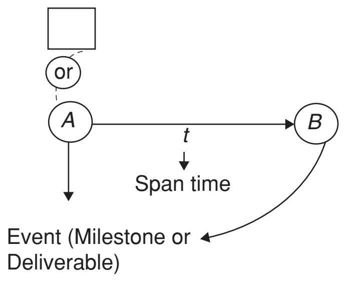
In real-life applications, an allowance for contingencies is provided. This is called **slack time.** Each project allows between 5 to 25 percent slack time for completion.

Program Evaluation and Review Technique (Pert)

Gantt charts do not show precedence relationships among the tasks and milestones of a project.

A PERT chart is a project management tool used to schedule, organize and coordinate tasks within a project.

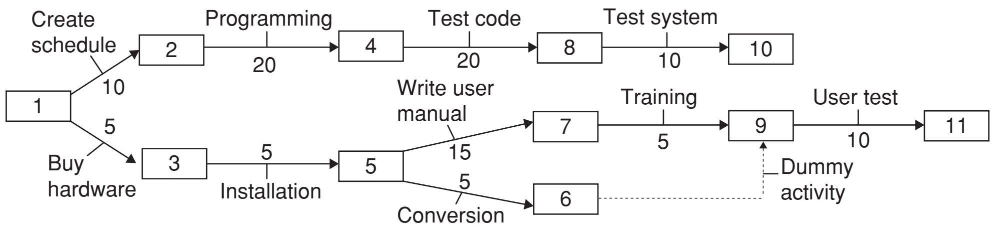
A PERT chart presents a graphic illustration of a project as a network diagram consisting of numbered nodes (either circles or rectangles) representing events, or milestones in the project linked by labelled vectors (directional lines) representing tasks in the project. The direction of the arrows on the lines indicates the sequence of tasks.



In the diagram, shown below the tasks between nodes 1, 2, 4, 8 and 10 must be completed in sequence and are called dependent or serial tasks. The tasks between nodes 1 and 2 and nodes 1 and 3 are not dependent on the completion of one to start the other and can be undertaken simultaneously. These tasks are called parallel or concurrent tasks. Tasks that must be completed in sequence but don't require resources or completion time are represented by dotted lines with arrows and are called dummy activates (Example: dashed arrow linking 6 and 9).

Numbers on the opposite sides of the vectors indicate the time allotted for the task.

The PERT chart is preferred over Gantt chart since it clearly illustrates task dependencies. But on complex projects, PERT chart may be much more difficult to interpret.



Thus in short,

Dependency diagrams can be defined as a formal notation to help in the construction and analysis of complex schedules. Dependency diagrams are drawn as a connected graph of nodes and arrows. Dependency diagrams consists of three elements:

- Event–A significant occurrence in the life of a project.
- Activity–Amount of work required to move from one event to the next.
- Span time–Actual calendar time required to complete an activity.

SOFTWARE DESIGN

Software design is the process in which requirements are translated into a blue print for constructing the software.

Once software requirements have been analyzed and modelled, software design is the last software engineering action within the modelling activity and sets the stage for construction (code generation and testing).

Architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns, that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.

The component-level design transforms structural elements of the software architecture into a procedural description of software components.

The major goals of the design process are:

- The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all the implicit requirements, desired by stakeholders.
- The design must be a readable, understandable guide for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional and behavioral domains from an implementation perspective.

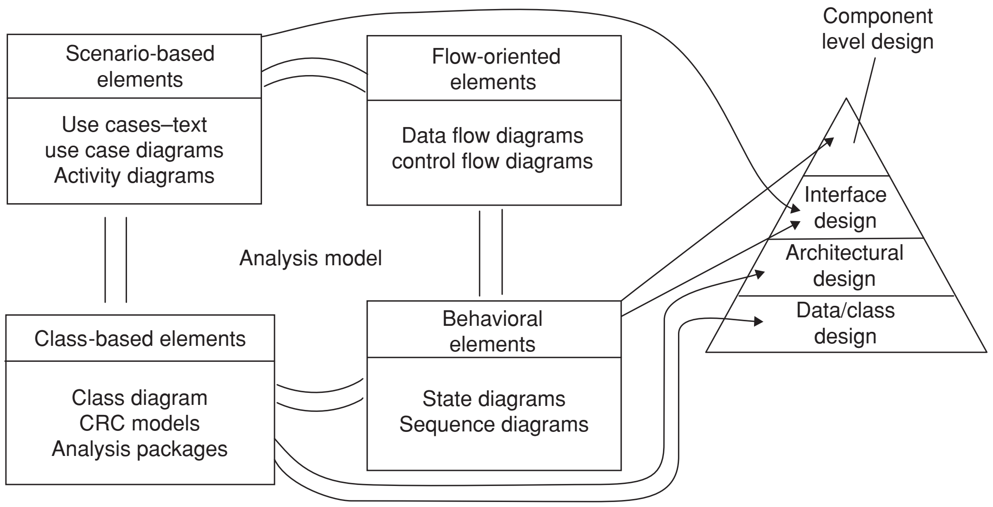


Figure 1 Design model.

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. In the beginning, once the software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

- The data/class design transforms analysis–class models into design class realizations and the requisite data structures required to implement the software.
- The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system.
- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (data/control) and a specific type of behavior.
- The component-level design transforms structural elements of the software architecture into a procedural description of software components.

Design Concepts

Important software design concepts:

Abstraction Many levels of abstraction can be posed while considering a modular solution to any problem. At highest level of abstraction, a solution is stated in broad terms and at lower levels, a more detailed description of the solution is provided. At the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Architecture Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by components.

Patterns The intent of each design pattern is to provide a description that enables a designer to determine:

1. whether the pattern is applicable to current work.
2. whether the pattern can be reused.
3. whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of concerns Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can be solved and/or optimized independently.

A concern is a feature or behaviour that is specified as part of the requirement model for the software.

Modularity common manifestation of separation of concerns. Software is divided into separately named and addressable components (modules) that are integrated to satisfy problem requirements.

Information hiding Modules should be specified and designed so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.

Functional independence Software should be designed in such a way that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure.

Functional independence is achieved by developing modules, which can perform a single function.

Refinement Refinement is a process of elaboration, begins with a statement or function defined at a high level of abstraction and then elaborates the original statement, providing more and more details as each successive refinement (elaboration) occurs.

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

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures or any other design failures that can be corrected to yield a better design.

Modeling Component Level Design

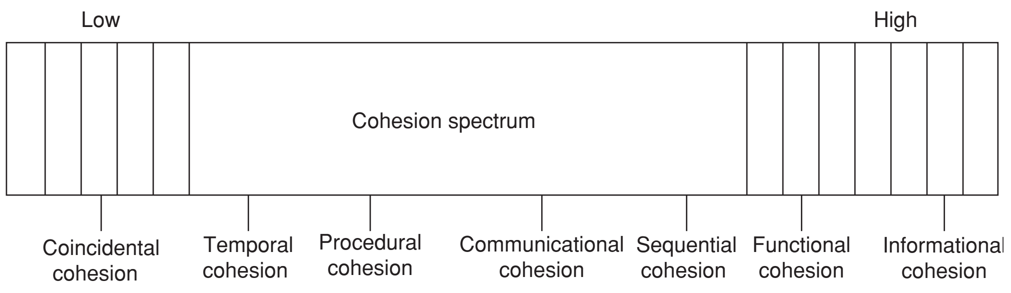
Component level design occurs after the first iteration of architectural design has been completed. At this stage the overall data and program structure of the software has been established.

Component A component is a modular building block for computer software.

Cohesion Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. Cohesion is a measure of internal relative strength of a module. It should be more. Different types of cohesion are:

1. **Coincidental cohesion:** If elements of a module are unrelated, then it is coincidental cohesive.
2. **Logical cohesion:** If elements of a module are related, then it is logical cohesion.
3. **Temporal cohesion:** If the elements of a module are elated and the elements are confined to initialization or time, it is temporal cohesion.
4. **Procedural cohesion:** If the elements are confined to one name and if they perform a set of operations, then the module is said to be procedural cohesive.
5. **Communicational cohesion:** If the elements in a module interact through data declared in it, then the module is said to be communicational cohesion.

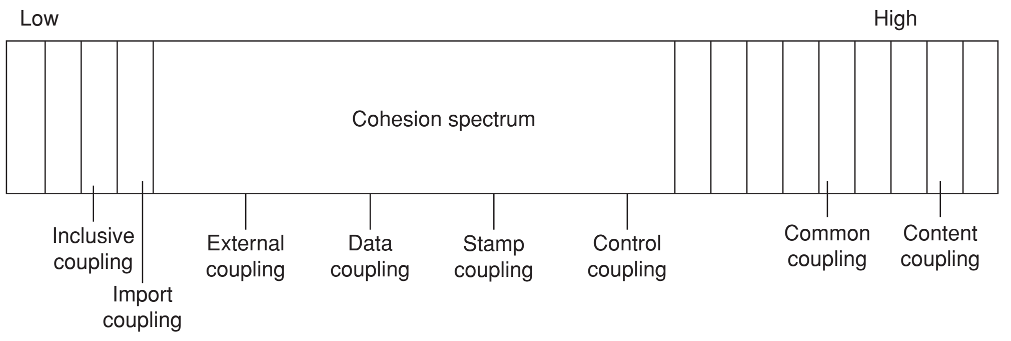
6. **Sequential cohesion:** If the elements are related and if they perform a set of operations in which the output of one operation is the input for another operation.
7. **Functional cohesion:** If the elements are related and if they are confined to one name and if they perform one and only one task, the module is functional cohesive.
8. **Informational cohesion:** If the elements of a module are confined to abstraction, it is informational cohesion.



Note: Cohesion metric should be high.

Coupling Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes and components become more interdependent, coupling increases. In component-level design coupling is to be kept as low as possible. It includes:

1. **Procedural or routine call coupling:** A form of coupling in which modules interact nominally more or less they are almost independent.
2. **Low coupling:** Form of coupling in which modules interact minimally. In extreme case there is no coupling between them.
3. **Inclusion coupling:** A coupling in which source code of one module is included into another module.
4. **Import coupling:** A coupling in which one module is declared in another module for its functionality.
5. **External coupling:** A coupling in which modules interact with modules written by some third party, which may include specific hardware or software.
6. **Data coupling:** Occurs when operations pass long strings of data arguments.
7. **Stamp coupling:** Occurs when a class is declared as a type for an argument of an operation of another class.
8. **Control coupling:** Coupling in which one module controls the order of execution of other module by using flags.
9. **Common coupling:** If the components make use of a global variable, it can lead to uncontrolled error propagation and unforeseen side effects when changes are made.
10. **Content coupling:** Type of coupling in when one module refers to other module, in extreme case, it changes internal structure of other modules for its functionality.



Note: Coupling metric should be low.

Coding

Coding may be

1. The direct creation of programming language source code (e.g., Java, C).
2. The automatic generation of source code using an intermediate design like representation of the component to be built or
3. The automatic generation of executable code using a 'fourth generation programming language' (**e.g.,** VC++).

The principles that guide the coding task are closely aligned with programming style, programming languages and programming methods.

The fundamental principles are:

- Understand the problem you are trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make the work easier.

Create a set of unit tests that will be applied once the component code is completed.

Characteristics of Good Srs

The characteristics of good SRS are

1. Correctness: The requirements specified in the software should meet, then the SRS is correct.
2. Unambiguous: The SRS is said to be unambiguous if every specified requirement can be interpreted in only one way.
3. Completed: The SRS is said to be complete, if and only if it has all significant requirements, definition of software responses to input data and labels and references to tables, figures and diagrams.
4. Consistent: The SRS is said to be consistent if the individual requirements are not defined in a conflict way and the SRS should be a high level document.
5. Stability: The SRS is said to be stable (or) ranked for the importance if each requirement has a preference. All the requirements may not have same importance; identify the requirements which are essential and requirements having least preference.
6. Verifiable: If each requirement is verifiable then the SRS is said to be verifiable.
7. Modifiable: The SRS is said to be modifiable, if the changes to the requirements can be made easily, consistent.
8. Traceable: Requirements should be clear so that each requirement can be referenced for enhancement, (or) future developments, which makes the SRS traceable.

Validation of SRS Validation of SRS is done to check whether the SRS is reflection of actual requirements and also to check the SRS documents is of good quality.

Testing

Testing is the 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 test is one that uncovers an as-yet-undiscovered error.

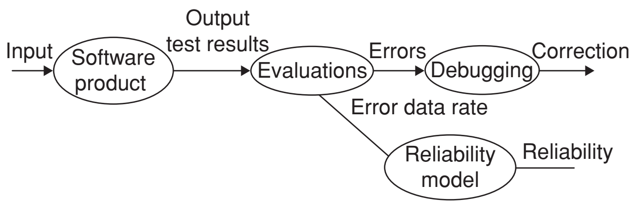
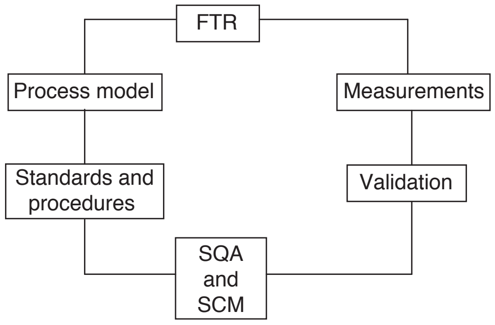


Figure 2 Formal technical review committee (FTR)



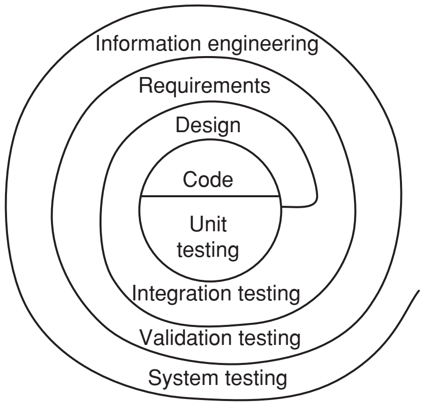


Figure 3 Verification

There are four software testing strategies:

1. Unit testing
2. Integration testing
3. Validation testing
4. System testing

Unit testing

Unit testing concentrates on each unit (e.g., class, component, etc). Unit test focuses on the internal processing logic and data structures within the boundaries of a component. Important control paths are tested to uncover errors within the boundary of the module, using component-level design description as a guide.

Integration testing

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

- **Top-down integration** 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.
- **Bottom-up Integration** begins construction and testing with the components at the lowest levels in the program structure.
- **Regression testing** in the context of an integration test strategy, regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- **Smoke testing** is an integration testing approach that is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

Validation testing

Validation succeeds when software functions in a manner that can be reasonably expected by the customer.

In validation testing, the requirements established as part of requirements modeling are validated against the software that has been constructed.

Software validation is achieved through a series of tests that demonstrate conformity with requirements.

Alpha and beta testing can be used to uncover errors that occur only at the end user.

The alpha test is conducted at the developer's site by a representative group of end users. The software is used in a natural setting by end users in the presence of the developer and the developer records usage problems.

The beta test is conducted at one or more end user sites in the absence of developer. Therefore, beta test is a 'live' application of the software in an environment that cannot be controlled by the developer. The customer records all problems and reports to developer.

System testing

In system testing, the software and other system elements are tested as a whole.

System testing is a series of different tests whose primary purpose is to fully exercise the computer-based system. The types of system tests used for software-based systems are:

- **Recovery testing** is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- **Security testing** attempts to verify that protection mechanisms built into a system will protect it from improper penetration.
- **Stress testing** executes a system in a manner that demands resources in abnormal quantity, frequency or volume. A variation of stress testing called sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.
- **Performance testing** is designed to test the run-time performance of software within the context of an integrated system.
- **Deployment testing** also called configuration testing exercises the software in each environment in which it is to operate. It also examines all installation procedures and specialized installation software that will be used by customers, and all documentation that will be used to introduce the software to end users.

SOFTWARE TESTING

The goal of testing is to find errors and a good test is one that has a high probability of finding an error.

The two ways of testing a software:

1. White-box testing (Internal testing)
2. Black-box testing (External testing)

White-box Testing

In white-box testing (also called glass-box testing) of software, tests are conducted to ensure that all internal operations are performed according to specifications and all internal components have been adequately exercised.

White-box testing methods should guarantee that:

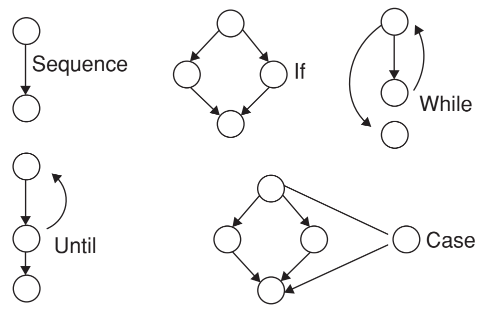
1. All independent paths, within a module are exercised at least once.
2. Exercise all logical decisions on their true or false sides.
3. Execute all loops at their boundaries and within their operational bounds and
4. Exercise internal data structures to ensure their validity.

Basis path testing

Basis path testing is a white-box testing technique. This method enables the test case designer to derive a logical complexity measure of a procedural design and uses this measure as a guide for defining a basis set of execution paths. Test cases derived are guaranteed to execute every statement in the program at least one time during testing.

Flow graphs can be used for better understanding the control flow and thus helps basis path testing to execute every statement in the program at least once.

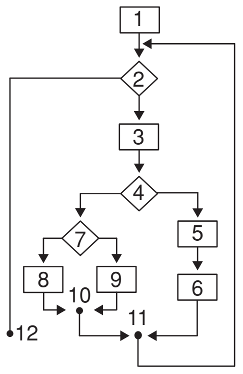
The flow graph symbols are:



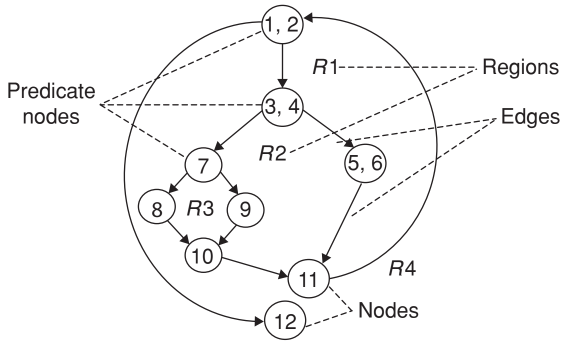
Each circle represents one or more non-branching PDL (Program Design Language) or source code statements.

Example:

Flowchart



Corresponding flow graph is



Each node that contains a condition is called a predicate node. Independent paths (any path through the program that introduces at least one new set of processing statements or a new condition) in the above example are:

Path 1: 1-2-12

Path 2: 1-2-3-4-5-6-11-2-12

Path 3: 1-2-3-4-7-8-10-11-2-12

Path 4: 1-2-3-4-7-9-10-11-2-12

Thus if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time, and every condition will have been executed on its true and false sides.

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Complexity is calculated in one of the three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity. (i.e., four Regions *R*1, *R*2, *R*3, *R*4 in the above case)
2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as $V(G) = E - N + 2$, when E is the number of flow graph edges and N is the number of flow graph nodes (i.e., in the above case, there are 11 edges and 9 nodes. Thus $V(G) = 11 - 9 + 2 = 4$)
3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as $V(G) = P + 1$, where P is the number of predicate nodes contained in the flow Graph G. In the above flow graph, there are 3 predicate nodes.

$$\therefore \quad V(G) = 3 + 1 = 4$$

Control structure testing

Some of the variations on control structure testing to improve the quality of white-box testing are:

Condition testing

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. This method focuses on testing each condition in the program to ensure that it does not contain errors.

Control Structure Testing

Condition testing

A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A compound condition is composed of two or more simple conditions, Boolean operators and parentheses. The possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of parentheses (surrounding a simple or compound Boolean condition), a relational operator, or an arithmetic expression.

Dataflow testing

This method selects test paths of a program according to the locations of definitions and use of variables in the program.

Loop testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four classes of loop can be defined as:

Simple loops

The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop
4. m passes through the loop where $m < n$
5. $n - 1$, n, $n + 1$ passes through the loop.

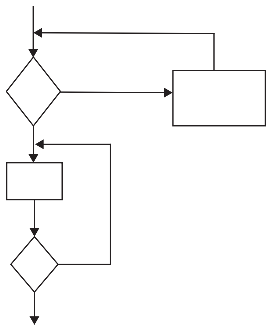


Figure 4 Simple loop.

Nested loops

Here the number of possible tests grows geometrically as the level of nesting increases. This results in an impractical number of tests.

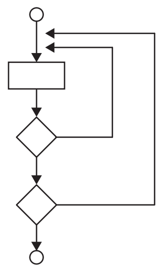


Figure 5 Nested loops.

Concatenated loops

Concatenated loops can be tested using approach of simple loops, if each of the loops is independent of the other. If two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, than the loops are not independent.

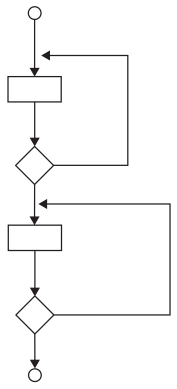


Figure 6 Concatenated loops.

Unstructured loops

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

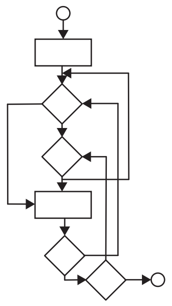


Figure 7 Unstructured loop.

Black-box Testing

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. Behaviour or performance errors and
5. Initialization and termination errors

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria:

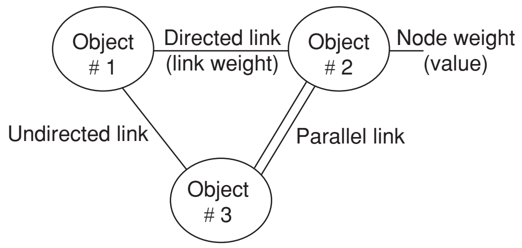
1. Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing.
2. Test cases that tell something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

In graph-based black-box testing methods, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

Graph-based testing methods

To accomplish these steps, the software engineer begins by creating a graph – a collection of nodes that represent objects; links that represent the relationships between objects; node weights that describe the properties of a node and link weights that describe some characteristic of a link.

The symbolic representation of a graph is as shown in the figure.



- Nodes are represented as circles connected by links that take a number of different forms.
- A directed link indicates that a relationship moves in only one direction.
- A bidirectional link (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.

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 partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions.

Boundary value analysis (BVA)

It is developed as a testing technique used to test bounding values since a greater number of error occurring at the boundaries of the input domain than at the centre.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the 'edges' of the class. BVA derives test cases from the input conditions as well as from the output domain.

Orthogonal array testing

The orthogonal array testing method is useful in finding region faults; an error category associated with faulty logic within a software component.

Orthogonal array testing can be applied to problems in which the input domain is relatively small.

When orthogonal array testing occurs, an Lg orthogonal array of test cases is created. This array has a 'balancing property', i.e., test cases are dispersed uniformly throughout the test doming.

Model-based testing (MBT)

It is a black-box testing technique that uses information contained in the requirements model as the basis for the generation of test cases.

White-box testing is usually performed at the early stages of testing process, while black-box testing tends to be applied during later stages of testing.

Implementation and Maintenance

System Implementation

Implementation is the process of converting a new or a revised system design into an operational one. Major aspects of implementation are conversion, post-implementation review and software maintenance.

There are three types of implementations:

1. Implementation of a computer system to replace a manual system.
2. Implementation of a new computer system to replace an existing one.
3. Implementation of a modified application to replace an existing one using the same computer.

Conversion

Conversion means changing from one system to another. The objective of conversion is to put the tested system into operation, while holding into costs, risks and personal irritation to a minimum.

It involves:

1. Creating computer-compatible files
2. Training the operating staff
3. Installing terminals and hardware

A very important aspect of conversion is not disrupting the functioning of the organization.

File conversion involves capturing data and creating a computer file from existing files.

Post implementation review

Every system requires periodic evaluation after implementation. A post-implementation review measures the system's performance against predefined requirements.

Unlike system testing, which determines where the system fails so that the necessary adjustments can be made, a post-implementation review determines how well the system continues to meet performance specifications. Post-implementation review is done after design and conversion are completed.

Software Project Estimation

Software is the most expensive element of virtually all computer-based systems. For complex, custom systems, a large cost estimation error can make the difference between profit and loss.

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cast and effort estimate for a software project) is too complex to be considered in one piece. For this reason, we decompose the problem recharacterizing it as a set of smaller problems.

Problem-based estimation

Lines of code (LOC) and function point (FP) are used in two ways during software project estimation.

1. As an estimation variable to 'size' each element of the software.
2. As baseline metrics collected from past projects and used in conjunction with estimated variables to develop cost and effort projections.

The project planner begins by estimating a range of values of each information domain value. Using the historical data, the planner estimates an optimistic, most likely, and pessimistic size value for each function or count for each information domain value.

The expected value for the estimation variables is computed as

$$S = \frac{\text{optimistic} + 4 * \text{Most likely} + \text{pessimistic}}{6}$$

Empirical estimation models

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP. The model should be tested by applying data collected from completed projects, plugging the data into the model and then comparing actual to predicted results.

Some of the LOC-oriented estimation models are

$E = 5.2 \times (\text{KLOC})^{0.91}$	Walston-Felix model
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Bailey-Basili model
$E = 3.2 \times (\text{KLOC})^{1.05}$	Boehm simple model
$E = 5.288 \times (\text{KLOC})^{1.047}$	Doty model for KLOC > 9

The software equation The software equation is a multivariable model that assumes a specific distribution of effort over the life of a software development project.

$E = [\text{LOC} \times B^{0.333}/P]^3 \times (1/t^4)$

where

E = effort in person – months or person – years

t = project duration in months or years

B = Special spills factor

P = Productivity parameter that reflects overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of software environment, the skills and experience of the software team, and the complexity of the application.

Note: B increases slowly as 'the need for integration, testing, quality assurance, and documentation and management skills grows'. For small programs KLOC = 5 to 15, $B = 0.16$.

For programs greater than 70 KLOC, $B = 0.39$

Putnam and Myers suggest a set of equations derived from the software equation.

Minimum development time is defined as

$t_{\min} = 8.14\ (\text{LOC}/P)^{0.43}$ in months for $t_{\min} > 6$ months

$E = 180\ \text{B}\ t^3$ in person – months for $E \geq 20$ person – months

Software Maintenance

Maintenance means restoring something to its original condition.

Maintenance is actually the implementation of the post-implementation review plan.

Maintenance is classified into corrective, adaptive or perfective maintenance.

Corrective maintenance repairs processing or performance failures or make changes because of previously uncorrected problems or false assumptions.

Adaptive maintenance means changing the program function.

Perfective maintenance enhances the performance or modify the programs to respond to the user's additional or changing needs.

About 50–80% of the total system development cost accounts for maintenance. Analysts and programmers spend far more time maintaining programs than they do writing them.

A manufacturer wants to minimize the variation among the products that are produced by maintaining the quality.

User satisfaction = compliant product + good quality + delivery within budget and schedule.

Software Quality Assurance (SQA)

Software Quality is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are points regarding quality is expected of all professionally developed software. In addition to the above definition some important

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Activities performed by SQA group:

1. Prepares an SQA plan for a project.
2. Participates in the development of the project's software process description
3. Reviews software engineering activities to verify compliance with the defined software process.
4. Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
5. Records any non-compliance and reports to senior management.

Software Reliability

Software reliability is defined as the probability of failure-free operation of a computer program in a specified environment for a specified time.

Measures of reliability and availability:

- A simple measure of reliability is mean-time-between-failure (MTBF).

  MTBF = MTTF + MTTR

  where

  MTTF = mean-time-to-failure

  MTTR = mean-time-to-repair

Although debugging (and related corrections) may be required as a consequence of failure, in many cases the software will work properly after a restart with no other change.

- In addition to a reliability measure, we must 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

  Availability = [MTTF/(MTTF + MTTR)] × 100%

Software Safety

- Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.

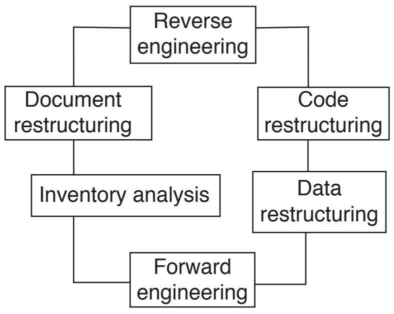
- Software safety examines the ways in which failures result in conditions that can lead to a mishap. That is the failures are evaluated in the context of an entire computer-based system and its environment.

Software Reengineering

Cost of redevelopment is very high compared to development.

The maintenance of existing software can account for over 60% of all effort expended by a development organization, and the percentage continues to rise as more software is produced.

A reengineering process model is shown below:



- Reengineering takes time, costs significant amount of money and absorbs resources that might be otherwise occupied on immediate concerns.
- Reengineering of information systems is an activity that will absorb information technology resources for many years.
- Inventory analysis : The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description of every active application. It should be revisited on a regular cycle.
- Document restructuring : It creates a framework of documentation that is necessary for the long-term support of an application.
- Code restructuring : The source code is analyzed using a restructuring tool. The restricted code is reviewed and tested to ensure that no anomalies have been introduced.
- Data restructuring : It is a full-scale reengineering activity. Current data architecture is dissected and necessary data models are defined.
- Forward engineering : Also called renovation or reclamation, covers design information from existing software and uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.
- Reverse engineering : It is the process of analyzing a program in an effort to extract data, architectural, and procedural design information.

The abstraction level of a reverse engineering process and the tools used to affect it refers to the sophistication of the design information that can be extracted from source code.

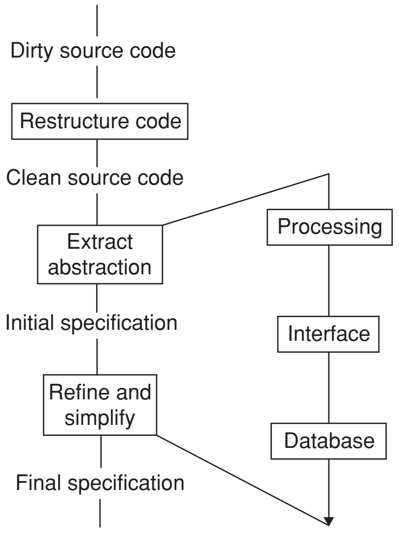


Figure 8 The reverse engineering process.

COCOMO Model

One of the famous model structures used to estimate the software effort is the constructive cost model, which is often called as COCOMO model. COCOMO was developed by Boehm. The model helps in defining the mathematical relationship between the software development time, the effort in man-months and the maintenance effort.

Basic COCOMO is defined as computers software development effort (and cost) as a function of program size. Program size is expressed in estimated thousand lines of code (KLOC) COCOMO is applied to three classes of software projects:

1. Organic projects
2. Semi-detached projects
3. Embedded projects

Organic projects

Organic projects are projects that are having small teams with good working experience with less than rigid requirements.

Semi-detached projects

Semi-detached projects are projects with medium teams having mixed working experience with a mix of rigid and less than rigid requirements

Embedded projects

Project that are developed within a set of tight constraints (hardware, software, operational…)

The general formula of the basic COCOMO model is

$E = a(s)^b$

where

$E \rightarrow$ Represents effort in person-months

$S \rightarrow$ Size of the software development in KLOC

'a' and '$\rightarrow$' 5 Values dependent on the development mode

Development Mode	Value of *a*	Value of *b*
Organic	2.4	1.05
Semi-detached	3.0	1.12
Embedded	3.6	1.20

Development time $D = C(E)^d$

People required $(P) = \frac{E}{D}$[count]

Development Mode	Value of *c*	Value of *d*
Organic	2.5	0.38
Semi-detached	2.5	0.35
Embedded	2.5	0.32

For intermediate COCOMO model, the value of coefficient Q and the exponent b are given in the table below:

Development Mode	Value of *a*	Value of *b*
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Exercises

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

Common data for questions 1 and 2: Consider the following payroll program that prints a file of employees and a file of information (transaction file) for the current month and for each employee.

In addition, the program updates the employee file, and produces an earnings report, a deduction report and analysis report. The application is capable of interactive command to print an individually requested pay slip. It also processes a file containing details of payment. This program can give printout of pay slips when they are requested individually. The weight table is shown below:

	Simple	Average	Complex
No. of inputs	3	4	6
No. of outputs	4	5	7
No. of enquiries	3	4	6
No. of files	7	10	15
No. of interfaces	5	7	10

1. What is the unadjusted function point for the given payroll program?
(A) 60 (B) 62
(C) 68 (D) 72

2. From the above problem, find adjusted function point where $F4 = 4$, $F5 = 3$, $F12 = 2$, $F14 = 5$?
(A) 49 (B) 62
(C) 82 (D) 90

Common data for questions 3 and 4: The size estimated for software of a certain project is 45,000 lines of code. The average salary paid per engineer is ₹20,000 per month.

3. Calculate the effort required if the software is of organic type.
(A) 100 pm (B) 120 pm
(C) 130 pm (D) 140 pm

4. Calculate the cost required if the software is of semi-detached type.
(A) 113000 (B) 213000
(C) 315000 (D) 326515

5. A 40 KDSI embedded program for teleprocessing is to be developed. Estimate the time required for the project using basic COCOMO model.
(A) 12 pm (B) 14 pm
(C) 16 pm (D) 18 pm

6. Consider the following code:
```
begin
If (x ≤ 0) then x = 0 - x;
a = x;
end
```
Lata wants to test the program with test data. What are the sufficient values to execute both branches of the decision box?
(A) $x = 0, 4$ (B) $x = 0, -4$
(C) $x = 1, 4$ (D) $x = 0, -1$

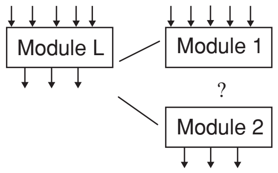
7. What is the maintainability of a software with average number of days of repairing code is 10, adapting code is 20 and for enhancing code is 10?
(A) 6.3 (B) 12.5
(C) 32.6 (D) 40

8. Consider a Java program and the SLOC is given as 1000.

Class A
```
{
int x(int a);
int y(int b);
int z(int c);
}
```
What is the modularity?
(A) 0.001 (B) 0.002
(C) 0.003 (D) 0.004

9. Raj has written a program to add two numbers. Assuming a 32-bit representation for an integer, to exhaustively test his program, the number of test cases required are
(A) 2^8 (B) 2^{16}
(C) 2^{32} (D) 2^{64}

10. The module of the length '*L*' is split up in two sub modules, module 1 and module 2, each of length $\frac{L}{2}$. How many links between the sub modules are allowed so that we maintain the value of information flow metric at same level?



(A) 2.4 (B) 3.6
(C) 4.8 (D) 1.2

11. The three estimates of the code size for a particular application for geometric analysis were most optimistic 4600, most likely 6900, most pessimistic is 8600. The value of estimated size that should be taken is
(A) 4600 (B) 6800
(C) 6900 (D) 8600

12. For an application of developing new operating system the KLOC is 34.5. What is the number of person-month (effort) best estimated using the intermediate COCOMO model?
(A) 126 (B) 130
(C) 158 (D) 196

13. For a real-time software systems the KLOC is 28.2. What is the effort in person–month calculated by using basic COCOMO model?
(A) 146 (B) 198
(C) 220 (D) 248

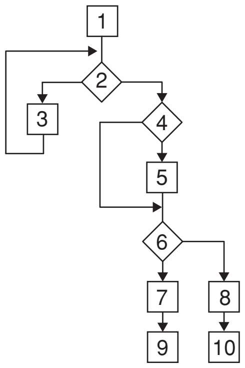
14. For inventory management system the KLOC is 25.5, what is the effort in person-month, using basic COCOMO model?
(A) 110 (B) 113
(C) 120 (D) 140

15. For the above, what is the estimated project duration in months?
(A) 6 (B) 8
(C) 10 (D) 13

Practice Problems 2

Directions for questions 1 to 16: Select the correct alternative from the given choices.

Common data for questions 1 and 2: Consider the below flow graph:



1. What is the number of paths to node 9?
(A) 2 (B) 3
(C) 4 (D) 5

2. What is the reachability measure?
(A) 1.8 (B) 2.8
(C) 2.4 (D) 2.1

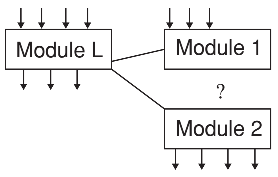
Common data for questions 3 and 4: For a software project the estimation is carried out by the Delphi method. Below table shows 5 experts with estimates:

Estimate	Pessimistic	Most likely	Optimistic
Expert 1	30	50	60
Expert 2	10	55	75
Expert 3	20	50	70
Expert 4	30	60	70
Expert 5	25	40	75

3. What is the average estimate?
(A) 48.3 (B) 49.4
(C) 50.8 (D) 56.7

4. What is the average variance?
(A) 5.0 (B) 6.7
(C) 7.8 (D) 8.3

5. The module of length *L* is split up into two sub modules (module-1 and module-2) each of length $\frac{L}{2}$. How many links between the sub modules exists so that we maintain the value of the information flow metric at the same level as found in the original module?



(A) 3 (B) 4
(C) 5 (D) 6

6. Constructive cost model is used to estimate
(A) Effort in man-month.
(B) Effort and schedule based on the size of the software.
(C) Size and duration based on the effort of the software.
(D) None of these

7. The theoretic concept that will be useful in software testing is
(A) Hamiltonian circuit
(B) Cyclomatic number
(C) Eulerian cycle
(D) None of these

8. Testing method that is normally used as the acceptance test for a software system is
(A) Regression testing
(B) Integration testing
(C) Unit testing
(D) None of these

9. Acceptance testing is
(A) The manner in which each component functions with other component of the system are tested.
(B) Running the system with given data by the actual user.
(C) The process of testing the changes in a new system or an existing system.
(D) None of these

10. Which of the following statements is true?
(A) Use of independent path testing criterion guarantees execution of each loop in a program under test more than once.
(B) Validation is the process of evaluating software at the end of the software development to ensure compliance with the software requirements.
(C) Statement coverage cannot guarantee execution of loops in a program under test.
(D) None of these

11. The size estimated for software of a certain project is 40,000 lines of code. The average salary paid per engineer is ₹15,000 per month. Calculate the cost required if the software is of organic type.
(A) 1,60,000
(B) 2,20,000
(C) 7,90,000
(D) 2,25,000

12. The size estimated for a software project is 35 Kloc. The average salary paid per engineer is ₹25,000 per month. Calculate the cost required if the software is of semi-detached type.
(A) 3,07,500
(B) 3,17,500
(C) 3,69,952
(D) 2,45,000

13. Which of the following statements is false?
(A) The cyclomatic complexity of a module is the number of decisions in the module plus one where a decision is effectively any conditional statement in the module.
(B) A direct flow of control in flow chart representing the lowest cyclomatic complexity.
(C) The reasonable limit of the cyclomatic complexity measure is 10.
(D) The cyclomatic complexity depends on the number of statements in the flowchart.

14. Which of the following is true regarding software testing?
(A) Software testing techniques are most effective if applied immediately after requirement specification.
(B) Software testing techniques are most effective if applied immediately after design.
(C) Software testing techniques are most effective if applied after coding.
(D) Software testing methods are most effective if applied after integration.

Common data for questions 15 and 16: A software project involves execution of 4 activities A_1, A_2, A_3, and A_4, of duration 11, 7, 8 and 3 days respectively. A_1 is the first one and needs to be completed before any other activity can commence. Activity A_2 and A_3 can be executed in parallel. Activity A_4 cannot commence until both A_2 and A_3 are completed.

15. Find the critical path of the above project.
(A) $A_1 - A_2 - A_4$
(B) $A_1 - A_3 - A_4$
(C) $A_1 - A_2 - A_3 - A_4$
(D) None of these

16. Find the slack time of the project.
(A) 0 (B) 1
(C) 12 (D) 13

Previous Years' Questions

1. The coupling between different modules of a software is categorized as follows:

I. Content coupling

II. Common coupling

III. Control coupling

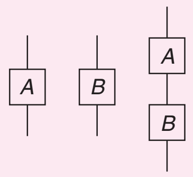
IV. Stamp coupling

V. Data coupling

Coupling between modules can be ranked in the order of strongest (least desirable) to weakest (most desirable) as follows: **[2009]**

(A) I-II-III-IV-V (B) V-IV-III-II-I
(C) I-III-V -II-IV (D) IV-II-V -III-I

2. The cyclomatic complexity of each of the modules A and B shown below is 10. What is the cyclomatic complexity of the sequential integration shown below? **[2010]**



(A) 19 (B) 21
(C) 20 (D) 10

3. A company needs to develop digital signal processing software for one of its newest inventions. The software is expected to have 40000 lines of code. The company needs to determine the effort in person-months needed to develop this software using the basic COCOMO model. The multiplicative factor for this model is given as 2.8 for the software development on embedded systems, while the exponentiation factor is given as 1.20. What is the estimated effort in person months? **[2011]**

(A) 234.25 (B) 932.50
(C) 287.80 (D) 122.40

4. The following is the comment written for a C function.

/ * This function computes the roots of a quadratic equation $ax \wedge 2 + bx + c = 0$. The function stores two real roots in * root 1 and * root 2 and returns the status of validity of roots. It handles four different kinds of cases.

(i) When coefficient 'a' is zero irrespective of discriminant.

(ii) When discriminant is positive.

(iii) When discriminant is zero.

(iv) When discriminant is negative.

Only in case (ii) and (iii), the stored roots are valid. Otherwise 0 is stored in the roots. The function returns 0 when the roots are valid and −1 otherwise.

The function also ensures root1 > = root2

int get_QuadRoots (float a, float b, float c, float * root1, float * root 2); */

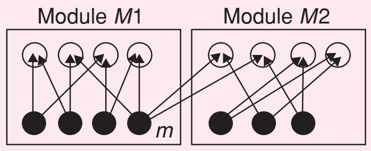
A software test engineer is assigned the job of doing black box testing. He comes up with the following test cases, many of which are redundant.

Test Case	Input Set			Expected Output Set		
	a	**b**	**c**	**root1**	**root2**	**Return value**
$T1$	0.0	0.0	7.0	0.0	0.0	–1
$T2$	0.0	1.0	3.0	0.0	0.0	–1
$T3$	1.0	2.0	1.0	–1.0	–1.0	0
$T4$	4.0	–12.0	9.0	1.5	1.5	0
$T5$	1.0	–2.0	–3.0	3.0	–1.0	0
$T6$	1.0	1.0	4.0	0.0	0.0	–1

Which one of the following options provide the set of non-redundant tests using equivalence class partitioning approach from input perspective for black-box testing? **[2011]**

(A) $T1$, $T2$, $T3$, $T6$ (B) $T1$, $T3$, $T4$, $T5$
(C) $T2$, $T4$, $T5$, $T6$ (D) $T2$, $T3$, $T4$, $T5$

5. The following figure represents access graphs of two modules $M1$ and $M2$. The filled circles represent methods and the unfilled circles represent attributes. If method m is moved to module $M2$ keeping the attributes where they are, what can we say about the average cohesion and coupling between modules in the system of two modules? **[2013]**



(A) There is no change
(B) Average cohesion goes up but coupling is reduced
(C) Average cohesion goes down and coupling also reduces.
(D) Average cohesion and coupling increase.

Common data for questions 6 and 7: The procedure given below is required to find and replace certain characters inside an input character string supplied in array A. The characters to be replaced are supplied in array 'oldc', while their respective replacement characters are supplied in array 'newc'. Array A has fixed length of five characters, while arrays 'oldc' and 'newc' contain three characters each.

However, the procedure is flawed.

```
void find_and_replace (char *A, char
*oldc, char *newc) {
for (int i = 0; i <5; i++)
for (int j=0; j<3; j++)
if (A[i] == oldc[j]) A[i] = newc[j];
}
```

The procedure is tested with the following four test cases.

1. oldc = “abc”, newc = “dab”
2. oldc = “cde”, newc = “bcd”
3. oldc = “bca”, newc = “cda”
4. oldc = “abc”, newc = “bac”

6. If array *A* is made to hold the string “abcde”, which of the above four test cases will be successful in exposing the flaw in this procedure? **[2013]**

(A) None (B) 2 only
(C) 3 and 4 only (D) 4 only

7. The tester now tests the program on all input strings of length five consisting of characters ‘*a*’, ‘*b*’, ‘*c*’, ‘*d*’ and ‘*e*’ with duplicates allowed. If the tester carries out this testing with the four test cases given above, how many test cases will be able to capture the flaw? **[2013]**

(A) Only one (B) Only two
(C) Only three (D) All four

8. In the context of modular software design, which one of the following combinations is desirable? **[2014]**

(A) High cohesion and high coupling
(B) High cohesion and low coupling
(C) Low cohesion and high coupling
(D) Low cohesion and low coupling

Answer Keys

Exercises

Practice Problems 1

1. B	**2.** A	**3.** C	**4.** D	**5.** C	**6.** A	**7.** B	**8.** C	**9.** D	**10.** B
11. B	**12.** C	**13.** B	**14.** B	**15.** D					

Practice Problems 2

1. C	**2.** B	**3.** B	**4.** C	**5.** B	**6.** A	**7.** B	**8.** D	**9.** B	**10.** B
11. D	**12.** C	**13.** D	**14.** B	**15.** B	**16.** B				

Previous Years’ Questions

1. A	**2.** A	**3.** A	**4.** C	**5.** A	**6.** C	**7.** B	**8.** B