



SEN Chapter 3

Computer science (K.L.E. Dr. M.S. Sheshgiri College of Engineering and Technology)



Scan to open on Studocu

Chapter 3

3.1 Translating requirement model into design model –

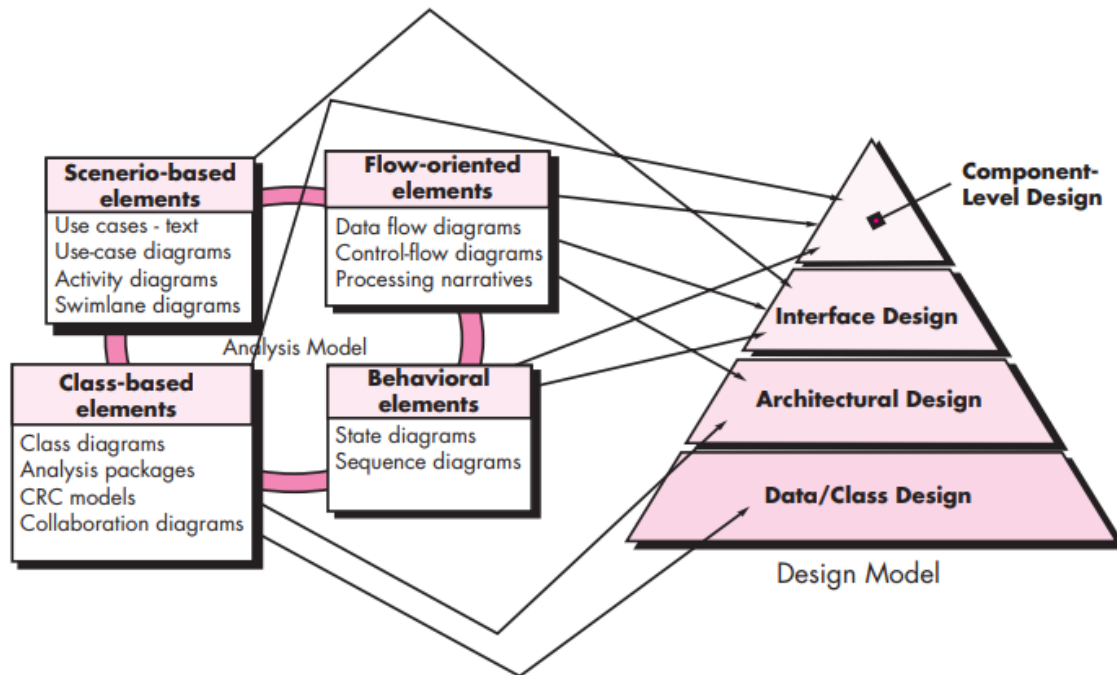


Fig 3.1

Once 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 flow of information during software design is illustrated in Figure 3.1. The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task.

- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action.
- 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, 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. An interface implies a flow of information (e.g., data and/or 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. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

3.2 Analysis modeling

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet.

The requirements modeling action results in one or more of the following types of models:

- Scenario-based models of requirements from the point of view of various system "actors"
- Data models that depict the information domain for the problem
- Class-oriented models that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- Flow-oriented models that represent the functional elements of the system and how they transform data as it moves through the system
- Behavioral models that depict how the software behaves as a consequence of external "events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs.

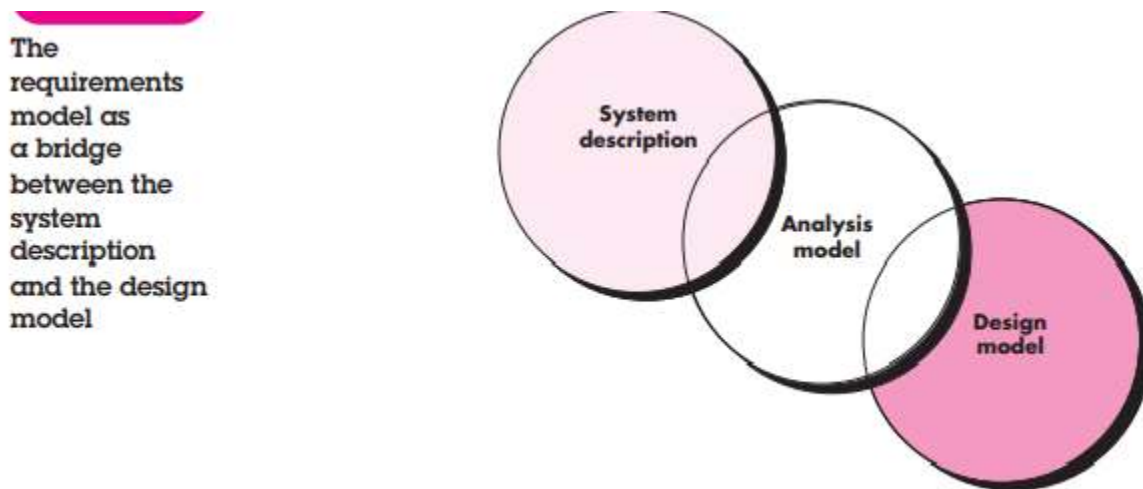


Fig3.2

3.3 Rules of thumb that should be followed when creating the analysis model:

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high. "Don't get bogged down in details" [Arl02] that try to explain how the system will work.

- Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.
- Delay consideration of infrastructure and other nonfunctional models until design. That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- Minimize coupling throughout the system. It is important to represent relationships between classes and functions. However, if the level of “interconnectedness” is extremely high, effort should be made to reduce it.
- Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.
- Keep the model as simple as it can be. Don’t create additional diagrams when they add no new information. Don’t use complex notational forms, when a simple list will do.

3.4 Domain Analysis

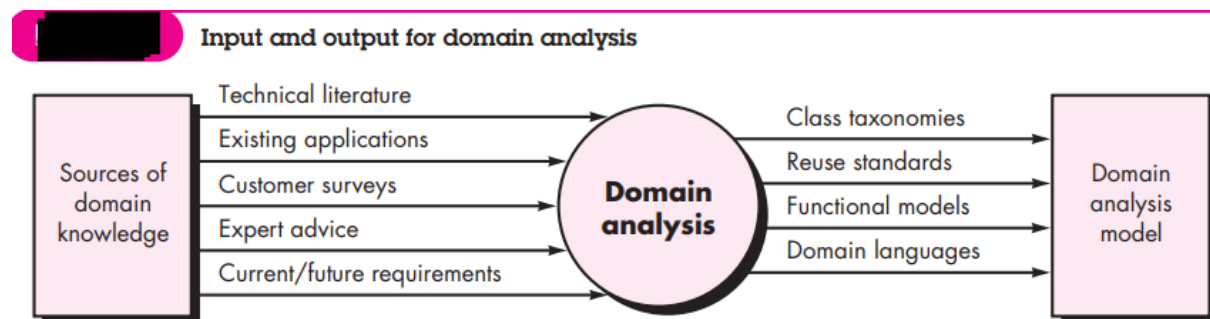


Fig 3.3

analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is advanced.

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain.

The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

Figure 3.3 illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

3.5 Elements of analysis modeling

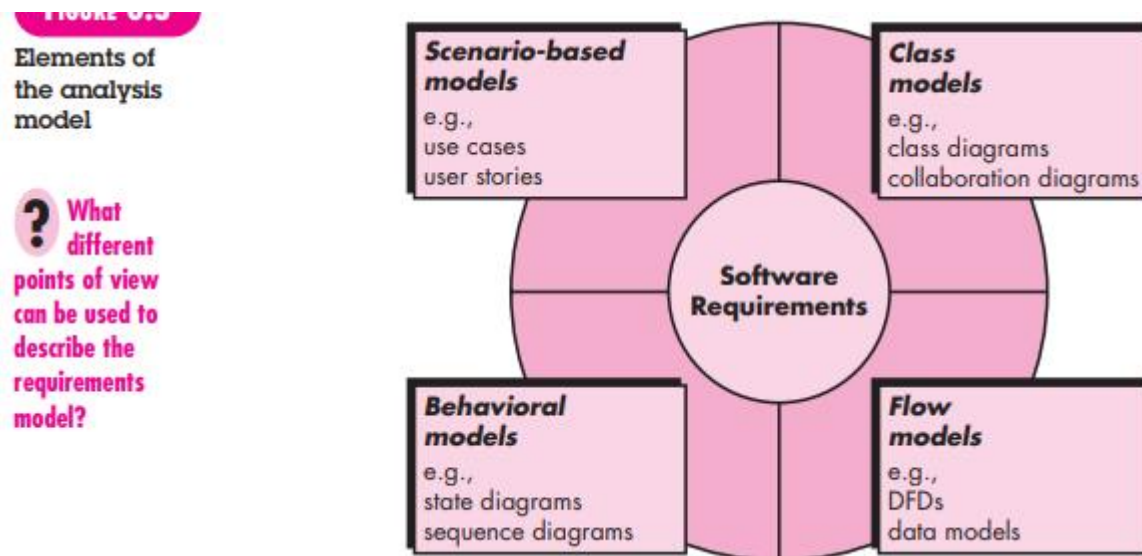


Fig 3.4

Each element of the requirements model (Figure 3.4) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

3.6 Data Modeling

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling.

The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

3.6.1 Data objects

A data object is a representation of composite information that must be understood by software. Composite information, means something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

The data object can be represented as a table as shown in Figure 3.5. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color, and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.

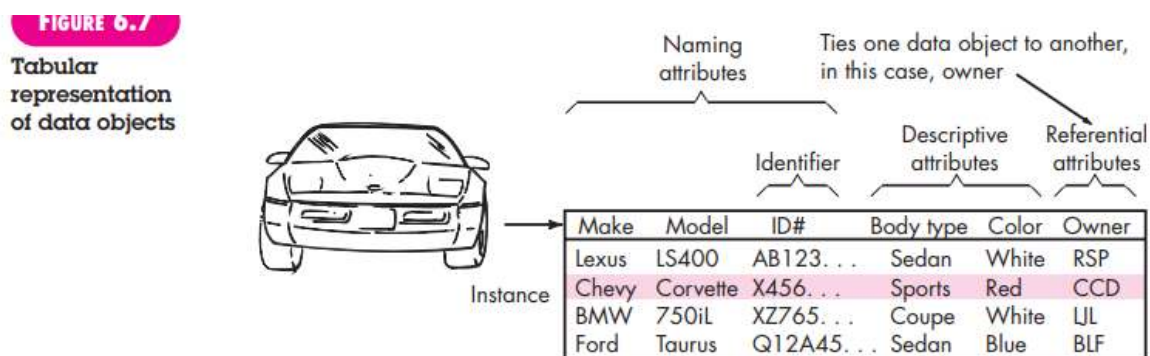


Fig 3.5

3.6.2 Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a “key” when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number.

3.6.3 Relationships

Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation illustrated in Figure 3.6

You can establish a set of object/ relationship pairs that define the relevant relationships.

For example,

- A person owns a car.
- A person is insured to drive a car

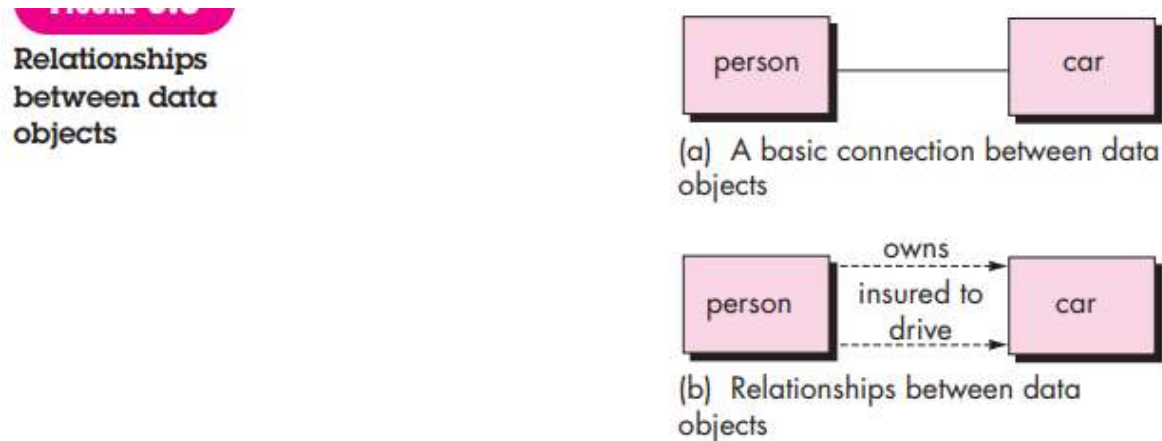


Fig 3.6

3.7 Design Concepts

A set of fundamental software design concepts has evolved over the history of software engineering

1. Abstraction - When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. As different levels of abstraction are developed, you work to create both procedural and data abstractions.

A procedural abstraction refers to a sequence of instructions that have a specific and limited function.

A data abstraction is a named collection of data that describes a data object.

2. Architecture - Software architecture alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”. In its simplest form, 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 the components.

3. Patterns - “A pattern is a proven solution to a recurring problem within a certain context amidst competing concerns” The intent of each design pattern is to provide a description that enables a designer to determine (1) whether the pattern is applicable to the current work, (2) whether the pattern can be reused (hence, saving design time), and (3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. 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 each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

5. Modularity - Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

If you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 3.7, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance. The curves shown in Figure 8.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M . Undermodularity or overmodularity should be avoided.

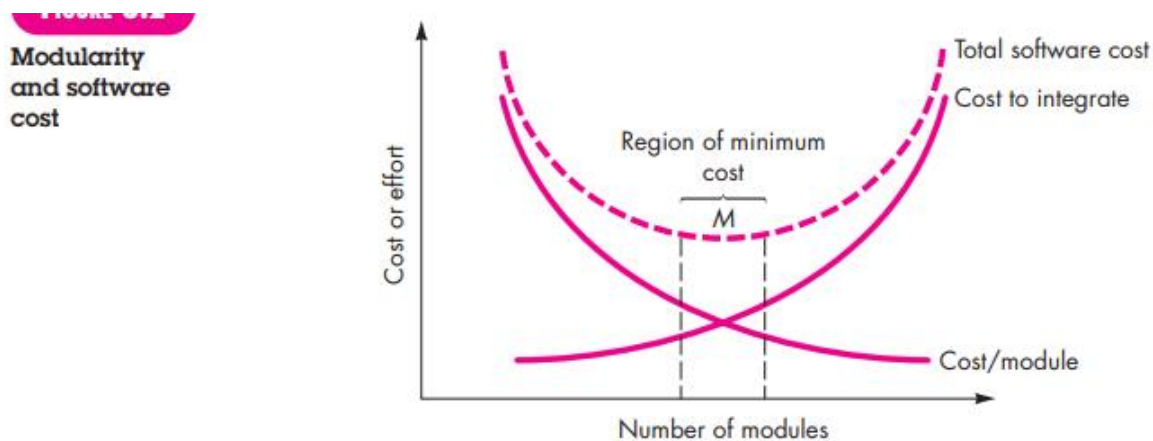


Fig 3.7

6. Information Hiding - The principle of information hiding suggests that modules be “characterized by design decisions that (each) hides from all others.” In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function.

7. Functional Independence - Functional independence is achieved by developing modules with “singleminded” function. You should design software so that each module addresses a specific subset of

requirements and has a simple interface when viewed from other parts of the program structure. Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible.

8. Refinement - Refinement is actually a process of elaboration. You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

9. Aspects - a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account"

An aspect is a representation of a crosscutting concern.

It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components.

10. Refactoring - Refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

3.8 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. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the

system at various levels. DFD does not contain any control or branch elements.

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

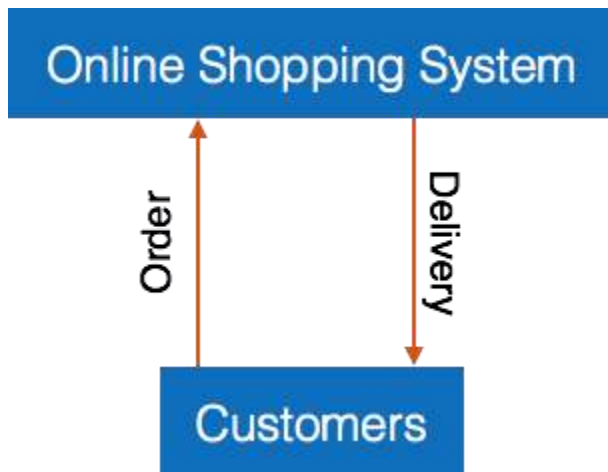
DFD can represent Source, destination, storage and flow of data using the following set of components -



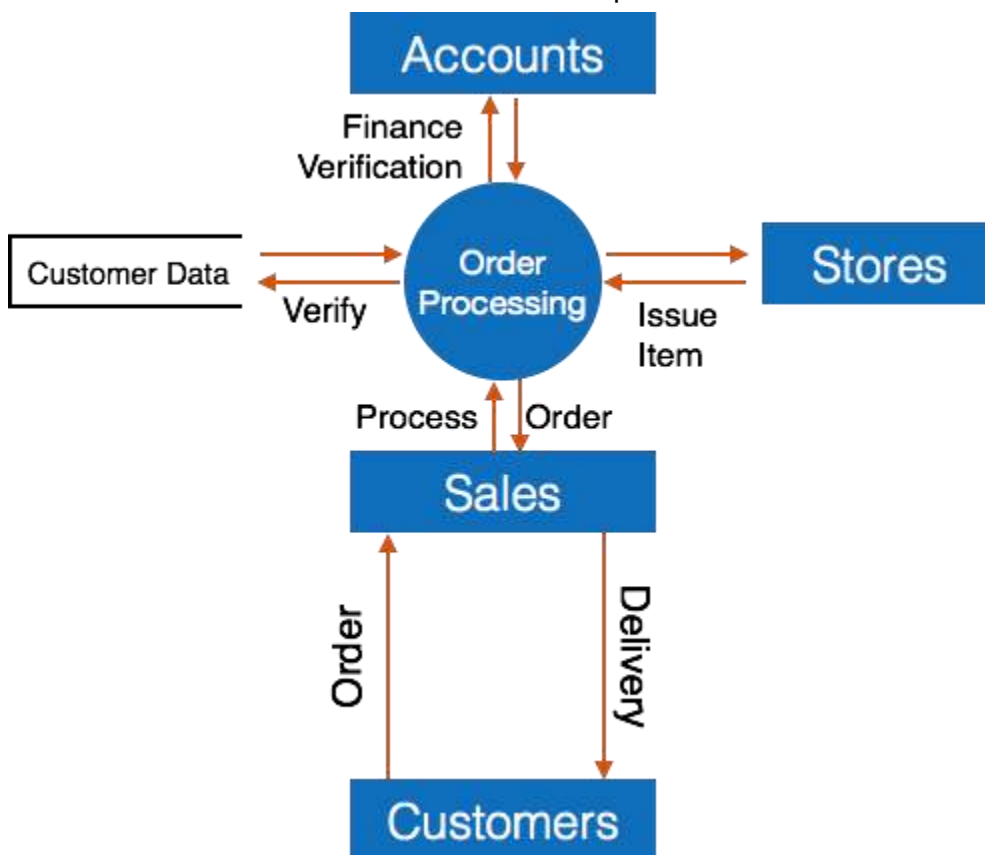
- **Entities** - Entities are source and destination of information data. Entities are represented by a rectangles with their respective names.
- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage** - There are two variants of data 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.

Levels of DFD

- **Level 0** - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



- **Level 1** - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.

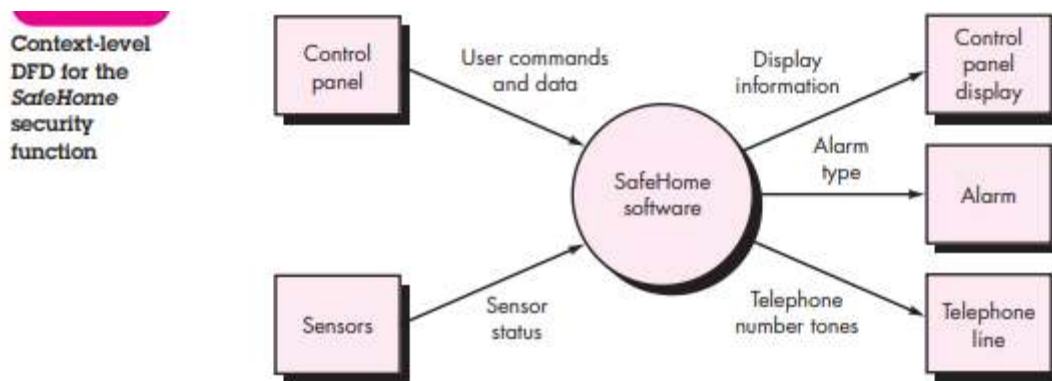


- **Level 2** - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

DFD(from book) –

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles). The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or context diagram) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.



Creating a Data Flow Model

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram: (1) the level 0 data flow diagram should depict the software/system as a single bubble; (2) primary input and output should be carefully noted; (3) refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level; (4) all arrows and bubbles should be labeled with meaningful names; (5) information flow continuity must be maintained from level to level,² and (6) one bubble at a time

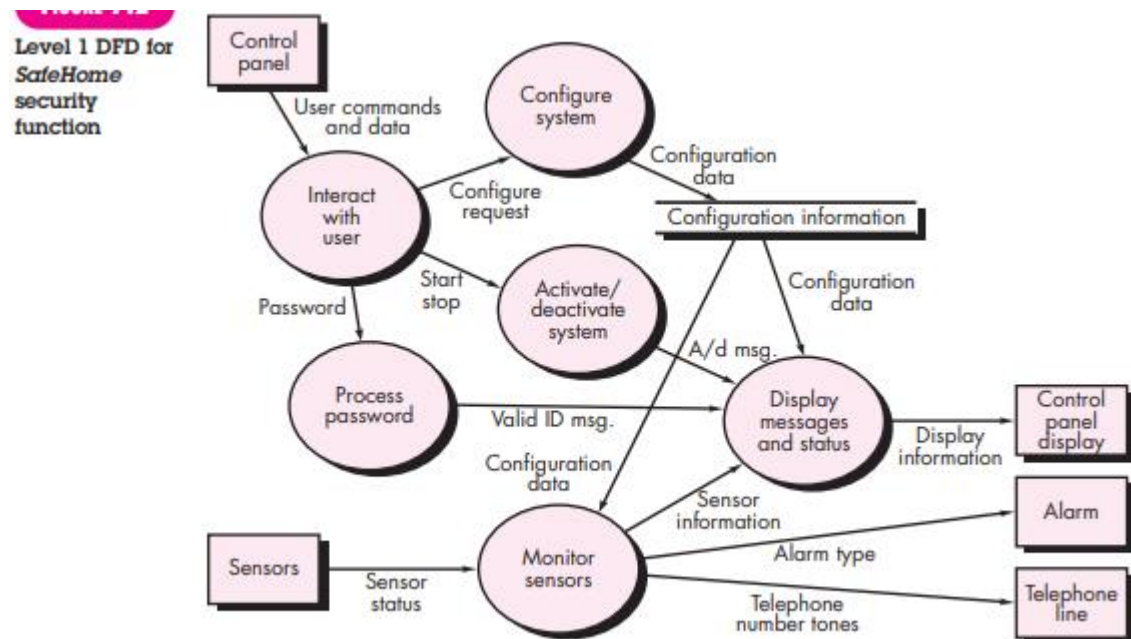
should be refined. There is a natural tendency to overcomplicate the data flow diagram. This occurs when you attempt to show too much detail too early or represent procedural aspects of the software in lieu of information flow.

To illustrate the use of the DFD and related notation, we again consider the

SafeHome security function. A level 0 DFD for the security function is shown in

Figure above. The primary external entities (boxes) produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies. For example, user commands and data encompasses all configuration commands, all activation/deactivation commands, all miscellaneous interactions, and all data that are entered to qualify or expand a command.

A level 1 DFD is shown in Figure below. The context level process shown in Figure above has been expanded into six processes derived from an examination of the grammatical parse. Similarly, the information flow between processes at level 1 has been derived from the parse. In addition, information flow continuity is maintained between levels 0 and 1.



3.9 Structure Charts

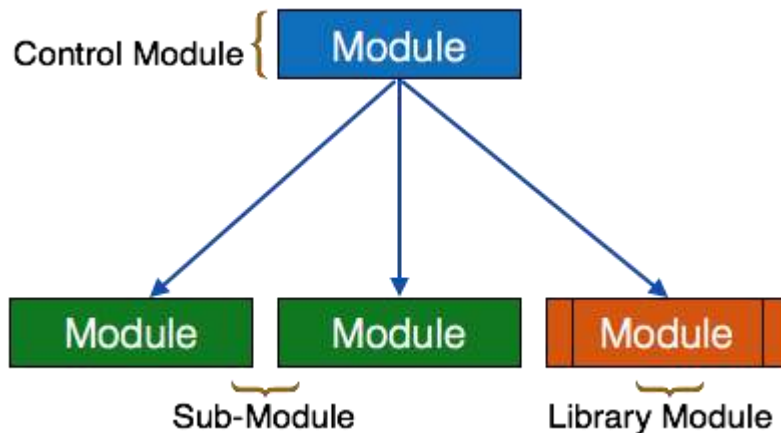
Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into

lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.

Structure chart represents hierarchical structure of modules. At each layer a specific task is performed.

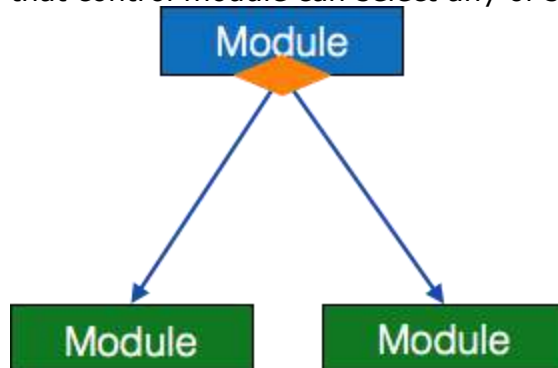
Here are the symbols used in construction of structure charts -

- **Module** - It represents process or subroutine or task. A control module branches to more than one sub-module. Library Modules are re-usable and invocable from

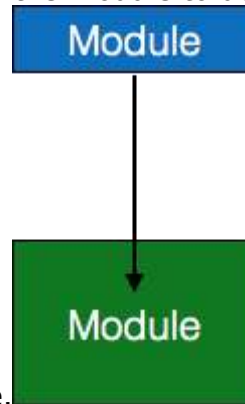


any module.

- **Condition** - It is represented by small diamond at the base of module. It depicts that control module can select any of sub-routine based on some condition.

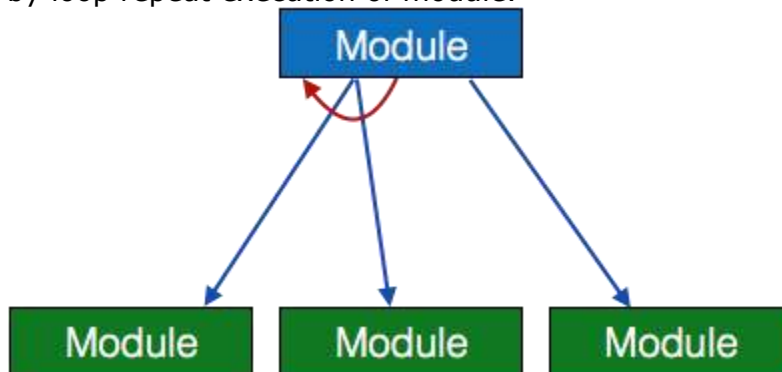


- **Jump** - An arrow is shown pointing inside the module to depict that the control

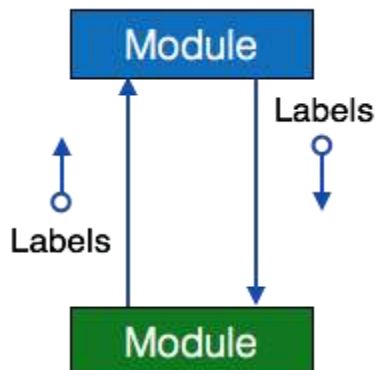


will jump in the middle of the sub-module.

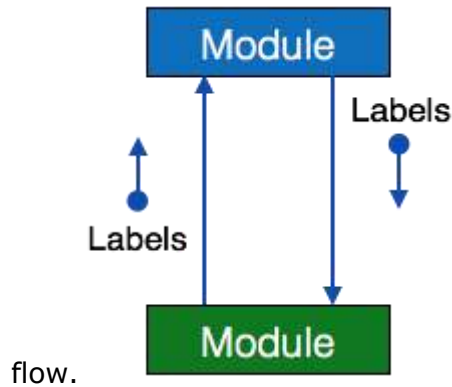
- **Loop** - A curved arrow represents loop in the module. All sub-modules covered by loop repeat execution of module.



- **Data flow** - A directed arrow with empty circle at the end represents data flow.



- **Control flow** - A directed arrow with filled circle at the end represents control



3.10 Decision Tables

Decision tables provide a notation that translates actions and conditions (described in a processing narrative or a use case) into a tabular form. The table is difficult to misinterpret and may even be used as a machine-readable input to a table-driven algorithm. Decision table organization is illustrated in Figure below. Referring to the figure, the table is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing rule.

The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or component).
2. List all conditions (or decisions made) during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what actions occur for a set of conditions.

Decision table nomenclature

Conditions	Rules					
	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

3.11 Testing

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computerbased system or a product with “testability” in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

1. Testability

Software testability is simply how easily [a computer program] can be tested.”

The following characteristics lead to testable software –

- **Operability.** “The better it works, the more efficiently it can be tested.” If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.
- **Observability.** “What you see is what you test.” Inputs provided as part of testing produce distinct outputs. System states and variables are visible or querable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.
- **Controllability.** “The better we can control the software, the more the testing can be automated and optimized.” All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.
- **Decomposability.** “By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.” The software system is built from independent modules that can be tested independently.
- **Simplicity.** “The less there is to test, the more quickly we can test it.” The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to

meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

- Stability. “The fewer the changes, the fewer the disruptions to testing.” Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.
- Understandability. “The more information we have, the smarter we will test.” The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

2. Test Characteristics –

- A good test has a high probability of finding an error - To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- A good test is not redundant - Every test should have a different purpose (even if it is subtly different).
- A good test should be “best of breed” - In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
- A good test should be neither too simple nor too complex -each test should be executed separately.

3.12 Approach to Software Testing -

Testing is a set of activities that can be planned in advance and conducted systematically.

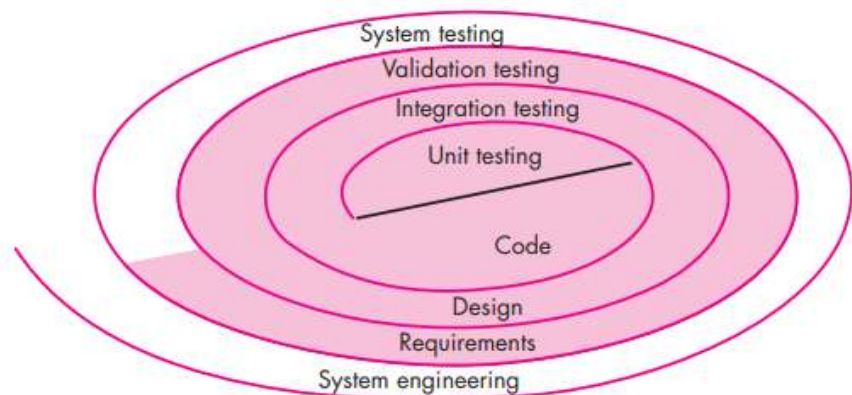
A number of software testing strategies have been proposed in the literature.

All provide you with a template for testing and all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews . By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

1. Verification and Validation - Software testing is one element of a broader topic that is often referred to as verification and validation (V&V).
Verification refers to the set of tasks that ensure that software correctly implements a specific function. Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements
Verification: "Are we building the product right?"
Validation: "Are we building the right product?"
Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.
2. Testing Strategy - The software process may be viewed as the spiral illustrated in Figure below. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counterclockwise) along streamlines that decrease the level of abstraction on each turn.

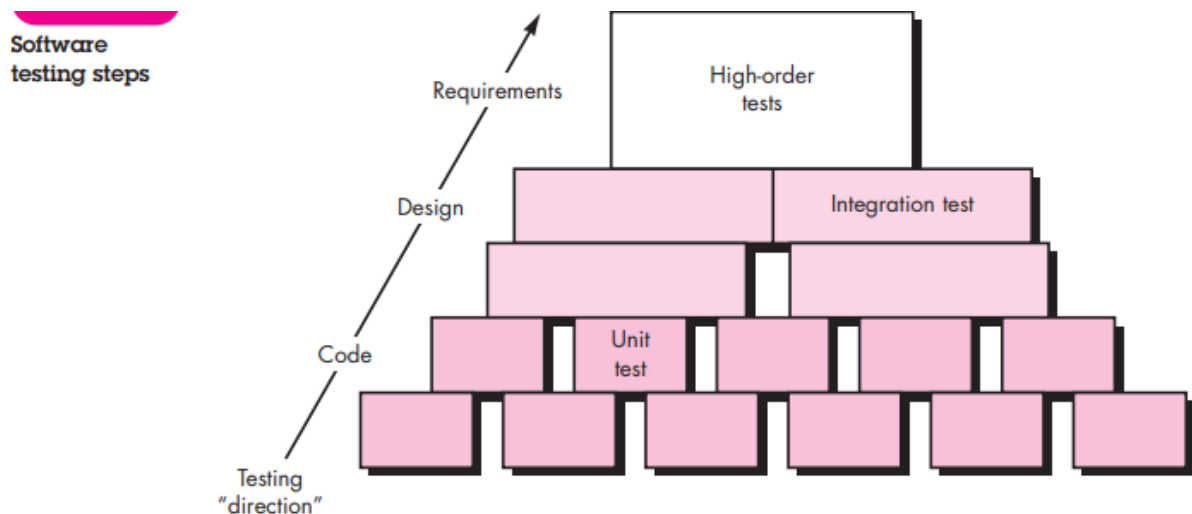
FIGURE 17-11
Testing strategy



A strategy for software testing may also be viewed in the context of the spiral. Unit testing begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code. Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter validation testing, where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at system testing, where the software and other system elements are tested as a whole. To test computer software, you spiral out in a clockwise direction along streamlines that broaden the scope of testing with each turn.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are

shown in Figure below. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name unit testing. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. After the software has been integrated (constructed), a set of high-order tests is conducted. Validation criteria (established during requirements analysis) must be evaluated. Validation testing provides final assurance that software meets all informational, functional, behavioral, and performance requirements. The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.



3.13 – Testing Methods

Any engineered product (and most other things) can be tested in one of two ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. (2) Knowing the internal workings of a product, tests can be conducted to ensure that “all gears mesh,” that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing.

Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

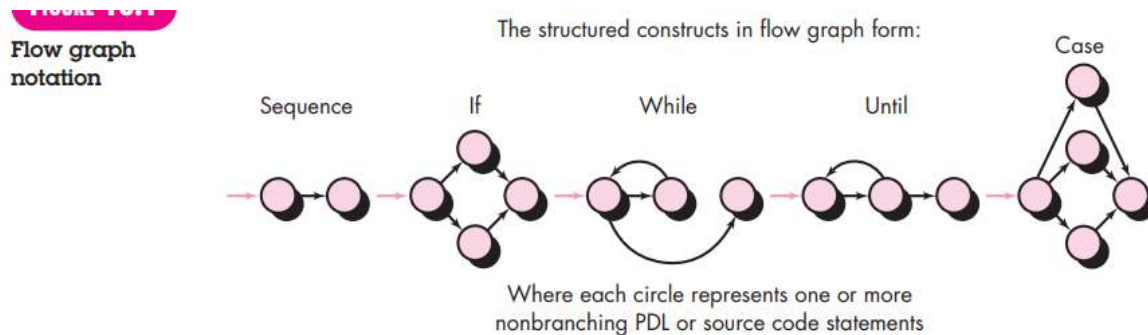
1. White-box testing

White-box testing, sometimes called glass-box testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

1.1 Basis Path testing –

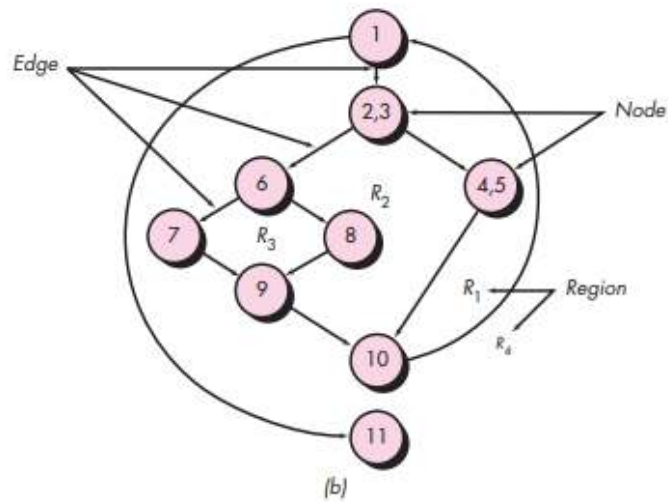
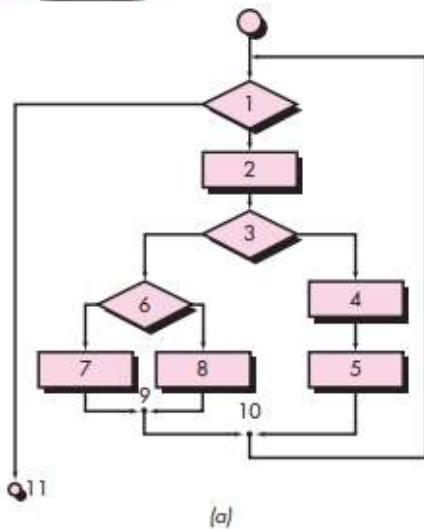
Basis path testing is a white-box testing technique. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed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

1.1.1 Flow Graph Notation - The flow graph depicts logical control flow using the notation illustrated in Figure below. Each structured construct has a corresponding flow graph symbol.



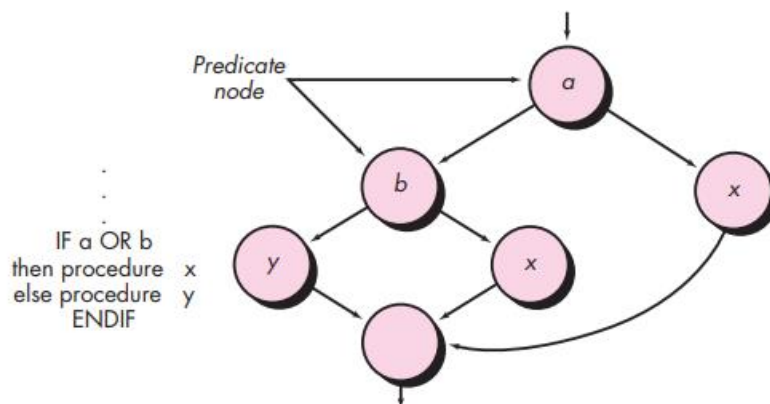
To illustrate the use of a flow graph, consider the procedural design representation in below Figure a. Here, a flowchart is used to depict program control structure. Figure below b, maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure b, 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 (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.

(a) Flowchart and (b) flow graph



When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure below, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

FIGURE 16.3
Compound logic



1.1.2 – Independent program path –

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a 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 flow graph illustrated in Figure b, is

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.

Paths 1 through 4 constitute a basis set for the flow graph in figure b. That is, if you can design tests 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.

2.Black Box Testing –

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.

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.

Tests are designed to answer the following questions:

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

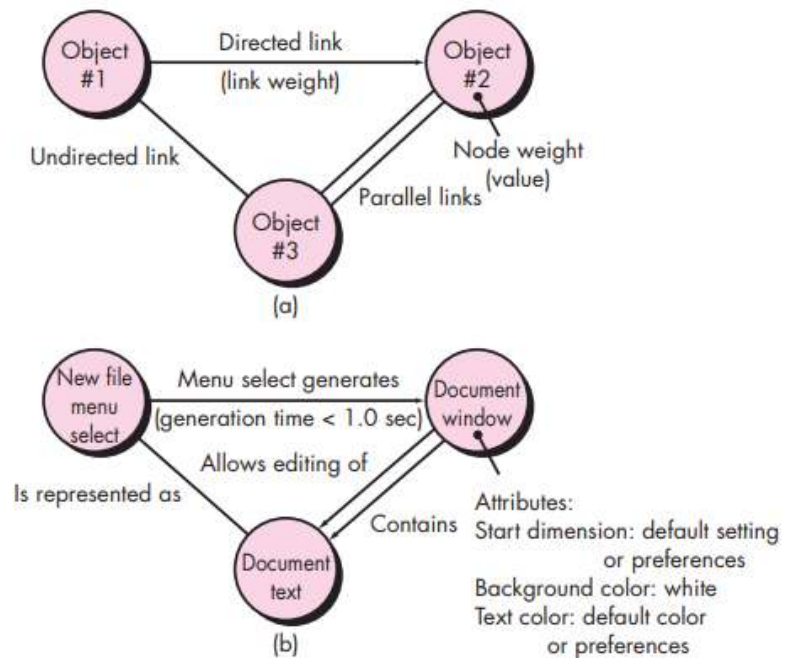
2.1 Graph based testing methods –

The first step in black-box testing is to understand the objects⁵ 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”. 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.

To accomplish these steps, you begin 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 (e.g., a specific data value or state behavior), and link weights that describe some characteristic of a link.

FIGURE 18.8
(a) Graph notation; (b) simple example



The symbolic representation of a graph is shown in Figure above. Nodes are represented as circles connected by links that take a number of different forms. A directed link (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. As a simple example, consider a portion of a graph for a word-processing application (Figure b) where

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.

A number of behavioral testing methods that can make use of graphs:

1. Transaction flow modeling- The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps (e.g., flightInformationInput is followed by validationAvailabilityProcessing).

2. Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the “screens” that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., orderInformation is verified during inventoryAvailabilityLook-up and is followed by customerBillingInformation input). The state diagram (Chapter 7) can be used to assist in creating graphs of this type.
3. Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node FICA tax withheld (FTW) is computed from gross wages (GW) using the relationship, $FTW = 0.62 \times GW$.
4. Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

2.2 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. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed. Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. If a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present. An equivalence class represents a set of valid or invalid states for input conditions.

Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed.

2.3 – Boundary Value Analysis –

A greater number of errors occurs 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. Boundary value analysis leads to a selection of test cases that exercise bounding values. 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. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

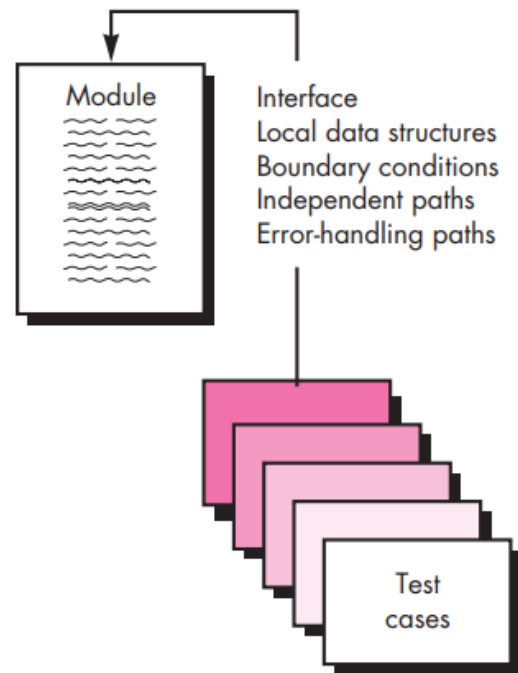
Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. 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.
2. 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.
3. 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.
4. 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.

3.13 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

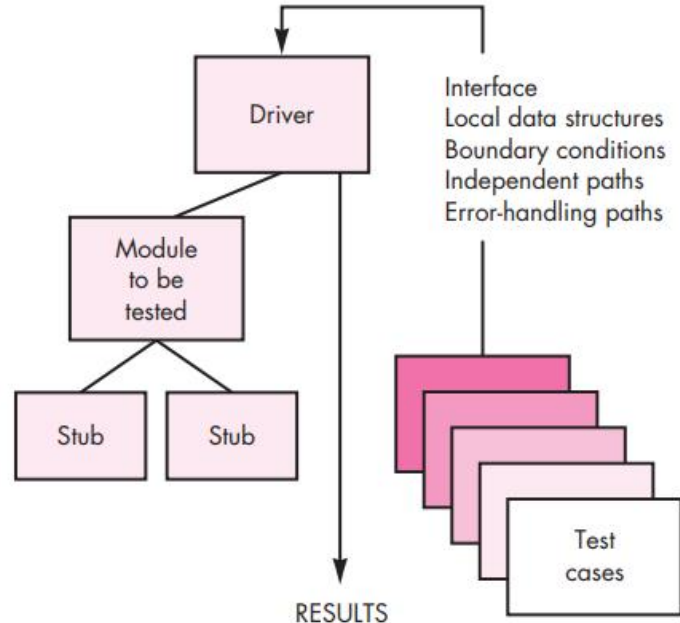
Unit test



Unit tests are illustrated schematically in Figure. The module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.

Unit-test procedures - Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results. Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test. The unit test environment is illustrated in Figure below. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Unit-test environment




3.14 Test Documentation –

A. Test case template – A test case template is a document comes under one of the test artifacts, which allows testers to develop the test cases for a particular test scenario in order to verify whether the features of an application are working as intended or not. Test cases

are the set of positive and negative executable steps of a test scenario which has a set of pre-conditions, test data, expected result, post-conditions and actual results.

Find the test case template screenshot below:

Project Name:	Google Email	 www.SoftwareTestingMaterial.com
Module Name:	Login	
Reference Document:	If any	
Created by:	Rajkumar	
Date of creation:	DD-MMM-YY	
Date of review:	DD-MMM-YY	

TEST CASE ID	TEST SCENARIO	TEST CASE	PRE-CONDITION	TEST STEPS	TEST DATA	EXPECTED RESULT	POST CONDITION	ACTUAL RESULT	STATUS (PASS/FAIL)
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name 2. Enter Password 3. Click "Login" button	<Valid User Name> <Valid Password>	Successful login	Gmail inbox is shown		
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name 2. Enter Password 3. Click "Login" button	<Valid User Name> <Invalid Password>	A message "The email and password you entered don't match" is shown			
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name 2. Enter Password 3. Click "Login" button	<Invalid User Name> <Valid Password>	A message "The email and password you entered don't match" is shown			
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name 2. Enter Password 3. Click "Login" button	<Invalid User Name> <Invalid Password>	A message "The email and password you entered don't match" is shown			

The main fields of a test case:

PROJECT NAME: Name of the project the test cases belong to

MODULE NAME: Name of the module the test cases belong to

REFERENCE DOCUMENT: Mention the path of the reference documents (if any such as Requirement Document, Test Plan, Test Scenarios etc.,)

CREATED BY: Name of the Tester who created the test cases

DATE OF CREATION: When the test cases were created

REVIEWED BY: Name of the Tester who created the test cases

DATE OF REVIEW: When the test cases were reviewed

EXECUTED BY: Name of the Tester who executed the test case

DATE OF EXECUTION: When the test case was executed

TEST CASE ID: Each test case should be represented by a unique ID. It's good practice to follow some naming convention for better understanding and discrimination purpose.

TEST SCENARIO: Test Scenario ID or title of the test scenario.

TEST CASE: Title of the test case

PRE-CONDITION (PRE-REQUISITES): Conditions which needs to meet before executing the test case.

TEST STEPS: Mention all the test steps in detail and in the order how it could be executed.

TEST DATA: The data which could be used an input for the test cases.

EXPECTED RESULT: The result which we expect once the test cases were executed. It might be anything such as Home Page, Relevant screen, Error message etc.,

POST-CONDITION: Conditions which needs to achieve when the test case was successfully executed.

ACTUAL RESULT: The result which system shows once the test case was executed.

STATUS: If the actual and expected results are same, mention it as Passed. Else make it as Failed. If a test fails, it has to go through the bug life cycle to be fixed.

B. Test Plan

A s/w test plan is a document that contains the strategy used to verify that the s/w product or system adheres to its design specification and other requirement.

Test plan may contain following test:-

1. Design verification test
2. Development or production test
3. Acceptance or commissioning test
4. Service and repair test
5. Regression test.

Test plan format varies organization to organization. Test plan should contain the three important elements like test coverage, test methods, test responsibilities.

Test Planning

- Plan is strategic document which describes how to perform a task in an effective, efficient and optimized way.
- A test plan is a document describing the scope, approach, objectives, resources and schedule of a s/w testing effort.
- A test plan identifies the items to be tested, items not to be tested, who will do the testing, the test approach followed, what will be the pass/ fail criteria, training needs for team.
- The goal of test planning is to take into account the important issues of testing strategy, resource utilization, responsibilities, risk and priorities.
- Test planning issues are reflected in the overall s/w project planning.
- The output of test planning is the test plan document. They are developed for each level of testing.

Preparing a Test Plan

- Testing any project should be driven by plan
- The test plan acts as the anchor for the execution, tracking and reporting of the entire testing project and covers-

1. What needs to be tested- the scope of testing, including clear identification of what will be tested and what will not be tested.
2. How the testing is going to be performed- breaking down the testing into small and manageable tasks and identifying the strategies to be used for carrying out the tasks.
3. What resources are needed for testing- computer as well as human resources.
4. The time lines by which the testing activities will be performed.
5. Risks that may be faced in all of the above, with appropriate mitigation and contingency plan.

C. Introduction to defect report –

DEFECT REPORT is a document that identifies and describes a defect detected by a tester. The purpose of a defect report is to state the problem as clearly as possible so that developers can replicate the defect easily and fix it.

Defect Report Template

In most companies, a defect reporting tool is used and the elements of a report can vary. However, in general, a defect report can consist of the following elements.

ID- Unique identifier given to the defect. (Usually, automated)

Project-Project name.

Product-Product name.

Release Version- Release version of the product. (e.g. 1.2.3)

Module-Specific module of the product where the defect was detected.

Detected Build Version- Build version of the product where the defect was detected (e.g. 1.2.3.5)

Summary-Summary of the defect. Keep this clear and concise.

Description-Detailed description of the defect. Describe as much as possible but without repeating anything or using complex words. Keep it simple but comprehensive.

Steps to Replicate- Step by step description of the way to reproduce the defect. Number the steps.

Actual Result- The actual result you received when you followed the steps.

Expected Results- The expected results.

Attachments- Attach any additional information like screenshots and logs.

Remarks- Any additional comments on the defect.

Defect Severity-Severity of the Defect. (See Defect Severity)

Defect Priority- Priority of the Defect. (See Defect Priority)

Reported By- The name of the person who reported the defect.

Assigned To- The name of the person that is assigned to analyze/fix the defect.

Status- The status of the defect. (See Defect Life Cycle)

Fixed Build Version- Build version of the product where the defect was fixed (e.g. 1.2.3.9)

- **D. Test summary report –**

Test Summary report- The final step in a test cycle is to recommend the suitability of a product for release. A report that summarizes the results of a test cycle is the test summary report. There are two types of test summary reports.

1. Phase wise test summary, which is produced at the end of every phase.
2. Final test summary reports (includes details of all testing phase)

- *A summary report should present--*

1. A summary of activities carried out during the test cycle or phase.
2. Variance of activities carried out from the activities planned.
3. Summary of results should include- test that failed with root cause, severity of impact of defects uncovered.
4. Comprehensive assessment and recommendation for release.

Preparing Test Summary Report:

- At the completion of a test cycle, a test summary report is produced.
- This report gives insights to the senior management about the fitness of the product for release.
- Test summary report is prepared after testing is completed.
- Summary report template provided is a useful guideline for what goes into such a report.
- The test summary report template is as given below---

Test summary report identifier:

Evaluation:

Summary:

Summary of activities:

Variances:

Approval:

Comprehensive assessment:

Summary of results