unit - 6 Design Engineering

Design concept :-

Software design is the process to transform user requirement in some suitable form, which helps programmer in software coding and implementation. During the software design phase The design document is produced based on customer requirement as documented in SRS.

Hence the aim of this phase is to transform SRS document into design.

Following items are designed and documented during design phase :-

1) Different module required.
2) control relationship among modules.
3) Interface among different modules
4) Data structure among the different modules.
5) Algorithm required to implement among the individual module

Objective of Software Design -

1) correctness - It should correctly implement all the functions and features to system.

2) Efficiency - a good software design should address the resource, time and cost ~~optimast~~ optimisation issue

3). understandability:-

a good design should be easily understandable for which it should be modular. and all modules should be arranged in layes.

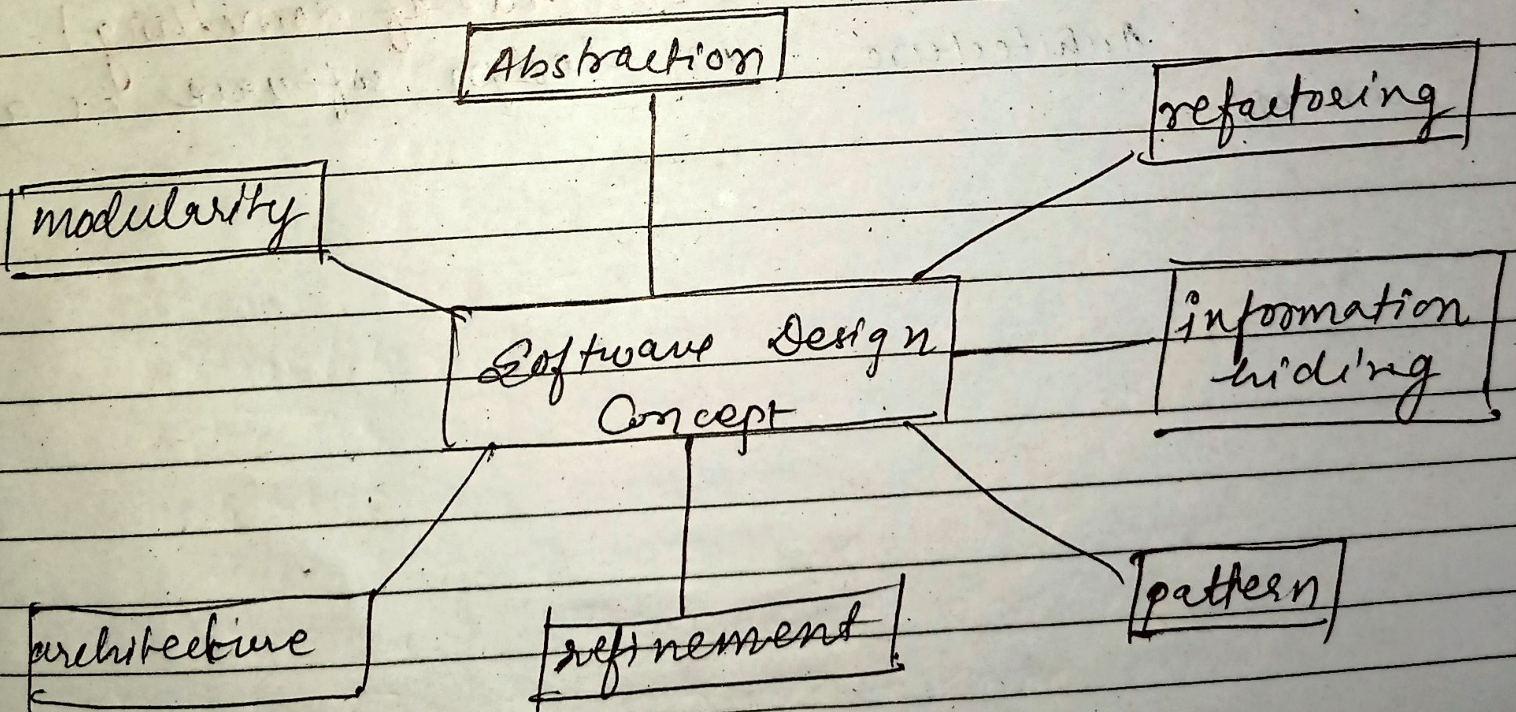
4) complerness -

The design should have all component like data structure, module & external interface etc

5) Maintainability-

a good software design should be capable of getting changes easily from request made from customer.

Software Design Concept:-



9.3.1 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 (e.g., a user story). At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology to state a solution (e.g., use case). Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented (e.g., pseudocode).

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. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *use* for a camera in the *SafeHome* system. *Use* implies a long sequence of procedural steps (e.g., activate the *SafeHome* system on a mobile device, log on to the *SafeHome* system, select a camera to preview, locate the camera controls on mobile app user interface, etc.).[6]

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **camera.** Like any data object, the data abstraction for **camera** would encompass a set of attributes that describe the camera (e.g., camera ID, location, field view, pan angle, zoom). It follows that the procedural abstraction *use* would make use of information contained in the attributes of the data abstraction **camera.**

9.3.2 Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [Sha15]. In its simplest form, architecture is the structure or organization of program components (modules), the ways in which these components interact, and the structure of data that are

6 It should be noted, however, that one set of operations can be replaced with another, if the function implied by the procedural abstraction remains the same. Therefore, the steps required to implement *use* would change dramatically if the camera were automatic and attached to a sensor that automatically triggered an alert on your mobile device.

used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design-level concepts.

9.3.3 Patterns

Brad Appleton defines a *design pattern* in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [App00]. Stated in another way, a design pattern describes a design structure that solves a well-defined design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

7 These families of related software products sharing common features are called *software product lines*.

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. Design patterns are discussed in detail in Chapter 14.

9.3.5 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.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and reduce the cost required to build the software.

Recalling our discussion of separation of concerns, it is possible to conclude that 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 9.2, the effort (cost) to develop an individual software module tends to decrease as the total number of modules increases.

Given the same set of requirements, the more modules used in your program means smaller individual sizes. However, as the number of modules grows, the effort (cost) associated with integrating modules with each other grows. These characteristics lead to a total cost or effort curve, shown in Figure 9.2. 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 9.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. Using too few modules or too many modules should be avoided.

9.3.6 Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of *information hiding* [Par72] suggests that modules should 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. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [Ros75].

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

9.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design, Wirth [Wir71] and Parnas [Par72] each allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, 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.

It is fair to ask why independence is important. 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. To summarize, functional independence is a key to good design, and design is the key to software quality. Evaluation of your CRC card model (Chapter 8) can help you spot problems with functional independence. User stories that contain many instances of words such as *and* or *except* are not likely to encourage you to design modules that are "single-minded" system functions.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept described in Section 9.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should

9.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design, Wirth [Wir71] and Parnas [Par72] each allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [Ste74] solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, 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.

It is fair to ask why independence is important. 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. To summarize, functional independence is a key to good design, and design is the key to software quality. Evaluation of your CRC card model (Chapter 8) can help you spot problems with functional independence. User stories that contain many instances of words such as *and* or *except* are not likely to encourage you to design modules that are "single-minded" system functions.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept described in Section 9.3.6. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should

9.3.8 Stepwise Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. Successively refining levels of procedural detail is a good way to develop an application. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is 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 indication of the internal workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more detail as each successive refinement (elaboration) occurs.

9.3.9 Refactoring

An important design activity suggested for many agile methods (Chapter 3), *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."

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 failure that can be corrected to yield a better design. For example, a first design iteration might yield a large component that exhibits low cohesion (i.e., it performs three functions that have only a limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each of which exhibits high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Although the intent of refactoring is to modify the code in a manner that does not alter its external behavior, inadvertent side effects can and do occur. Refactoring tools [Soa10] are sometimes used to analyze code changes automatically and to "generate a test suite suitable for detecting behavioral changes."

Different level of software design

1) architectural design–
the architecture of system can be viewed as the overall structure of system & the way in which structure provide conceptual integrity of system. Achitectural design identifies the software as system with many component interacting with each other.

2) preliminary or high level design–
problem is decomposed into a set of module the control relationship among various module get identified. The outcome of this stage is known as program architecture.

3) Detailed Design–
once the high level design is complete the detailed design is undertaken each module is examined carefully to design data structure & algorithm, Outcome of this stage is documented form of module specifi-cation.

Design elements

1. data design element
the data design element produced a model of data that represent high level of abstraction

after that this get more refined to get implemented by computer based system.

Architectural design element

the architecture design element provide us overall view of the system

- the architectural design element is generally represented as set of interconnected bus system that are derived from analysis packages

this model derived from following sources -
- the information about the application domain to built the software
- requirement model element like data model, data flow digram
- architectural style and pattern as per availability.

Interface design Element

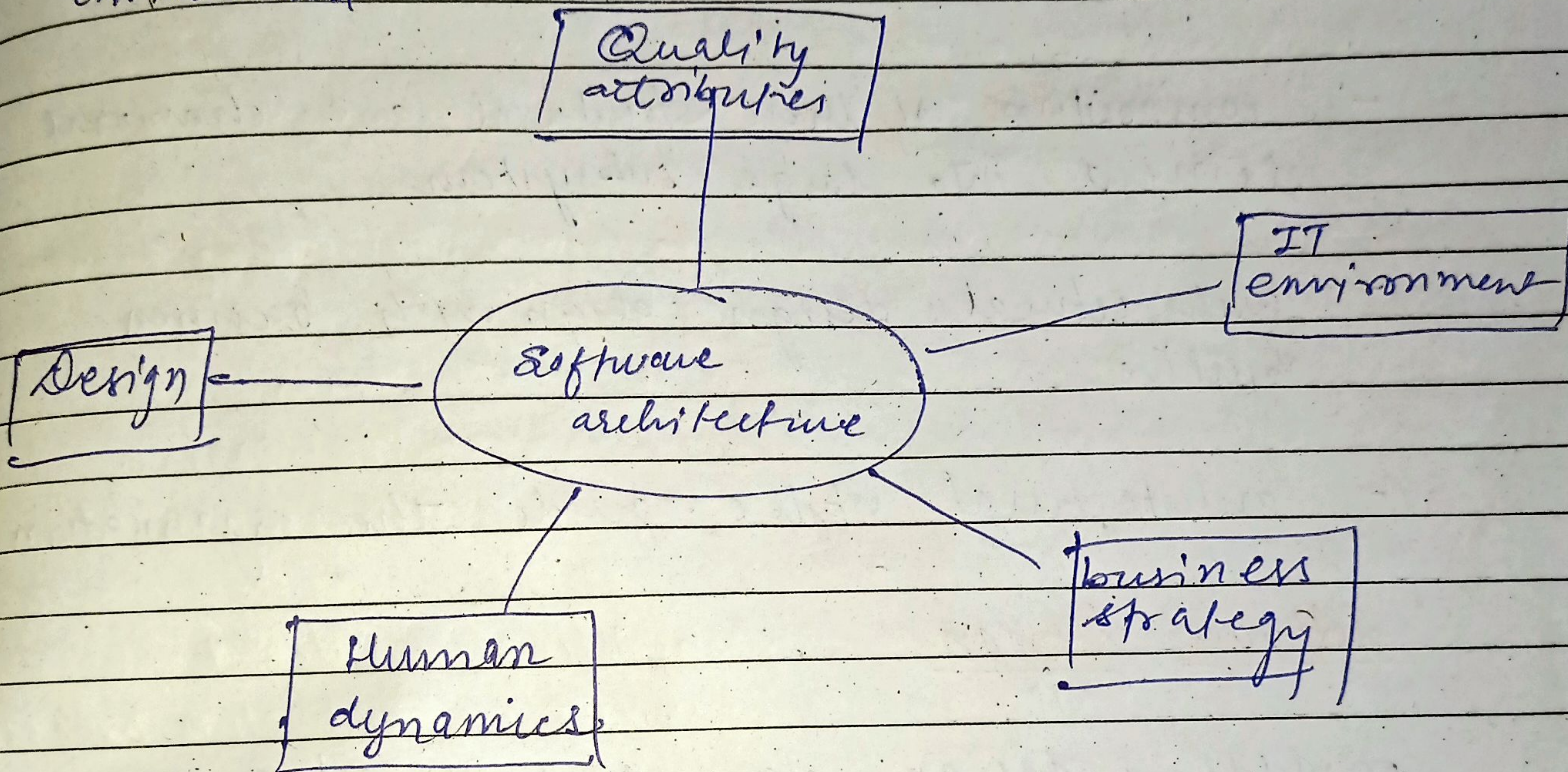
- represent the information flow within it and out of the system

- communicate between component defined as part of architecture
  - ↳ important element
    1. user interface
    2. external interface
    3. internal interface

Software architecture

the architecture of a system describes its major component. their relationship. and how they interact with each other.

This includes several factors - business strategy quality attribute, human dynamics. design and IT environment



in architecture non functional decisions are cast and seperated by functional requirements.

Architecture serve as blueprint for a system. It provides an abstraction to manage the system complexity and establish cordination among components.

(a) Defines structure to meet all the technical and operational requirements. while optimising the common quality attributes like performance and stability

- selection of structural elements and their interface by which system will be made up of.

- Behaviour as specified in collaboration among those elements.

- composition of these structural and behavioral element into large subsystem.

- architectural design align with business objectives

- architectural styles guide the organization.

Software Design

provides a design plan that describes the element of a system, how they fit, and work together to fulfil the requirement of the system.

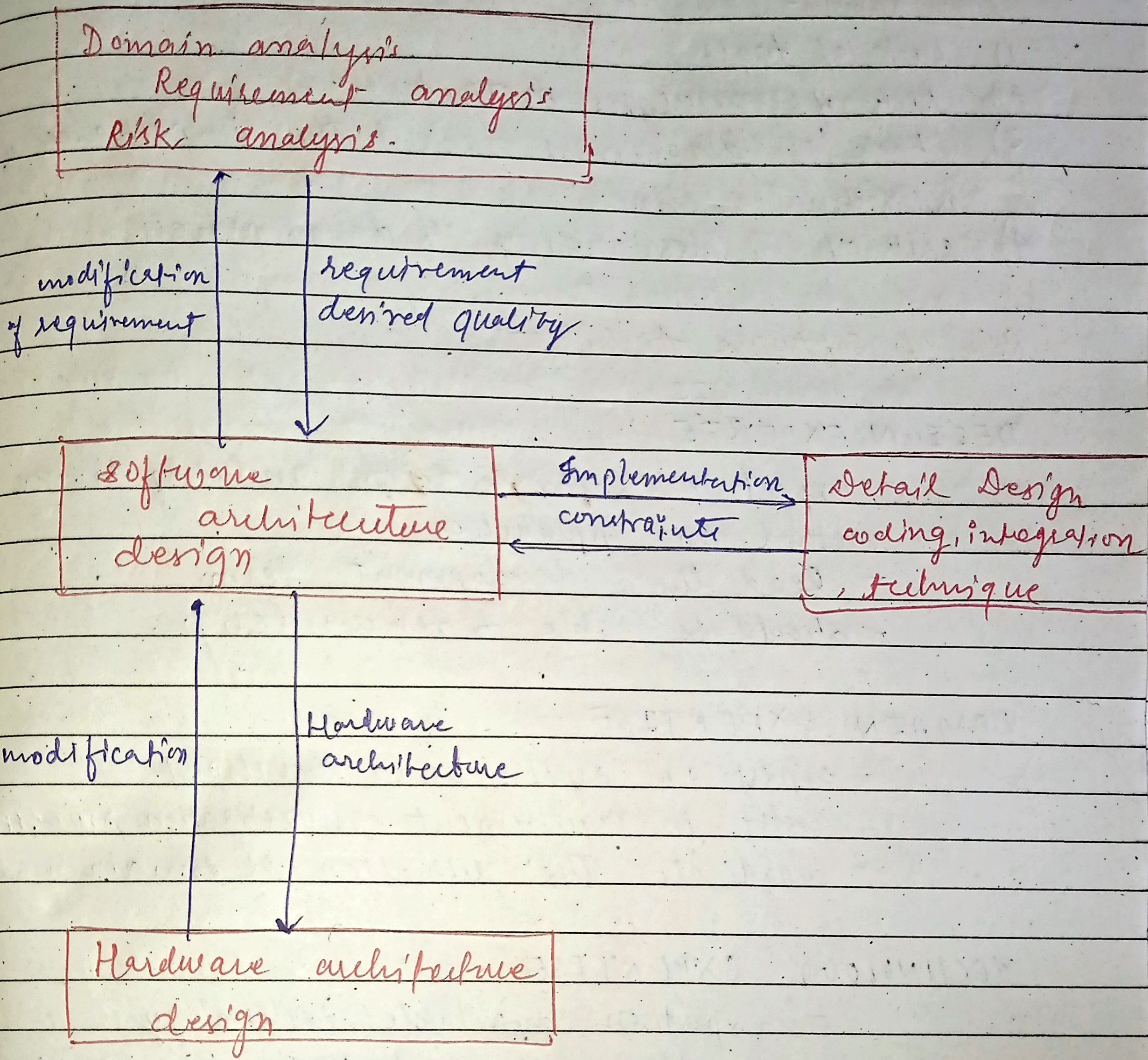
Objective -

1. To negotiate system requirement & to set expectation with customer, marketing and management personnel.

2. act as a blueprint during the development process

3. Guide the implementation task, including detailed design, coding, integration & testing.

Date:



Goals of architecture —

1) identify requirement that affect structure of application.

2) expose the structure of system, but hide its implementation.

3) Realise all the use-case & scenario.

4) Try to address the requirement of various stakeholders

5) Handle both functional & quality requirement

6) improve quality and functionality of system

Limitations-

1) Lack of tools
2) Lack of analysis method
3) Lack of awareness about the importance of architectural design
4) Lack of understanding of design process

Role of Software Architect

IS

DESIGN EXPERTE

- expert in software design, including diverse method and approaches.
- lead the development team
- should be able to review design

DOMAIN EXPERTISE

- expert on system being developed
- assist in requirement investigation process
- coordinate the defination of domain model

TECHNOLOGY EXPERTISE

- expert on available technology
- coordinate the selection of programming language

Methodological Expertise

- expert on software SDLC methodology
- choose the appropriate approaches for development that help entire team

Design-Model

- Design model can be viewed as two different dimension. process dimension indicates the evolution of design model as design task are executed.
  The abstraction dimension indicates the level of detail as each element of analysis model get transformed into design equivalent and then refined iteratively.

Dimension of Design model.

analysis model	class diagram analysis pack CRC model ↓	use case text + diagram state diagram ↓	class diagram analysis pack CRC model ↓	requirement: constraint interoperability ↓
abstraction dimension	Design class realisation subsystem ↓	technical interface design GUI design ↓	component diagram. design class activity diagram	Design class realisation subsystem collaboration ↓
Design model	refinement: Design class subsystem architecture element	interface element	refinement: component diag design class activity diagram component level element	Deployment diagram deployment level element

Data Design Element

Data Design element create a model of data and information that is represented at high a high level of abstraction. This model is often refined into progressively more implementation specific representation that can processed by a computer.

The design of data structure and algorithm associated with software which is require to manipulate is certainly a necessary part to create high quality softwares

At application level the translation of a data model into a database is done this gives a pivotal to achieving the business model and business objective of system

at business level the collection of data is stored at disparate database and is known as "data warehouse". enables data mining or knowledge discovery that can have an impact on business

Architectural design elements -

Architectural design gives an overall design of software architectural model gets derived from three sources -

1) information about the domain of software that will be built
2) specific requirement model elements.

3) availability of architectural styles and patterns.

architectural design is usually ~~depreciat~~ depicted as set of interconnected system often derived from analysis package within the requirement model. Each subsystem may have it's own architecture.

component level Design

component level design for software fully define each part of software in details. It defines data structure for all local data objects and algorithmic detail for all processing that occurs within components.

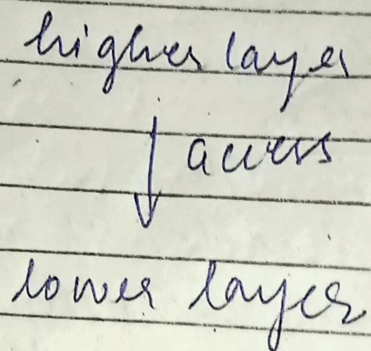
The design details in the model of component can be addressed at different level of abstraction A UML diagram can be used to represent processing logic. Algorithmic structure can be represented as pseudocode or the programming language which is going to be used on the implementation of software.

Cohesion :- it implies that a component ~~of~~ or class encapsulate only attributes and operation that are closely related to each other and to the class or component itself.

type :- 1) functional - exhibited primarily by operation. this level of cohesion occurs when a module

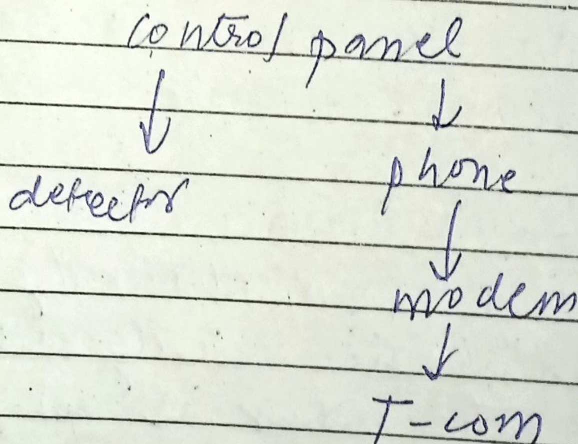
perform one and only one computation and they return a result.

Layer - Exhibited by packages, components and classes this type of cohesion occur when higher layer accesses the service of lower layer but lower layer do not access higher layer



communicational - all operation that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Layer cohesion



Coupling -

Coupling is a qualitative measure of degree to which classes are connected to one another. As components become more independent, coupling increase. An important objective in component level design is to keep coupling as low as possible.

coupling ∝ independent component

types -

Content Coupling -
occurs when one component "secretly modifies data that is internal to another component which violates information hiding.

Control Coupling -
when operation A() invokes operation B() and passes a control flag to B. The control flag than directs logical flow from B.

This can create necessity change in A() which is unrelated to B

External coupling - occurs when a component communicate or collaborate with ~~infrastr~~ infrastructure component

eg - OS, database etc.

(7) Software testing

What is it?

Software is tested to uncover errors that were made accidentally when it was designed & constructed. A software mon component testing of individual component and integrating them into a system

who does it?

Project manager. testing specialist, Software engineers.

why is it important?

Testing often account for more project effort than any other software engineering action if it is conducted in a reckless manner, time is wasted unnecessary effort wasted and worst case errors get undetected

A strategic approach for testing-

Testing is a set of activity that can be planned in advance and conducted system-atically. Therefore a set of steps including test case design techniques and testing methods should be defined.

1) to perform effective testing you should conduct technical reviews.

2. Testing begins at component level and work outwards toward the entire system

3. Different testing techniques are appropriate for different engineering approaches, at different points.

4. Testing is conducted by developer of s/w and for large project group of people are hired to perform test.

5. Testing and debugging are different task but debugging

Verification & Validation

verification refers to set of task that ensures that software correctly implement a specific function

Validation refer to a different set of task that ensures the software that has been built is traceable to customer requirement.

Boehm states —

verification : " are we building the product right ?"

validation : " are we building the right product?"

Testing strategy →

system testing

validation testing

integration testing

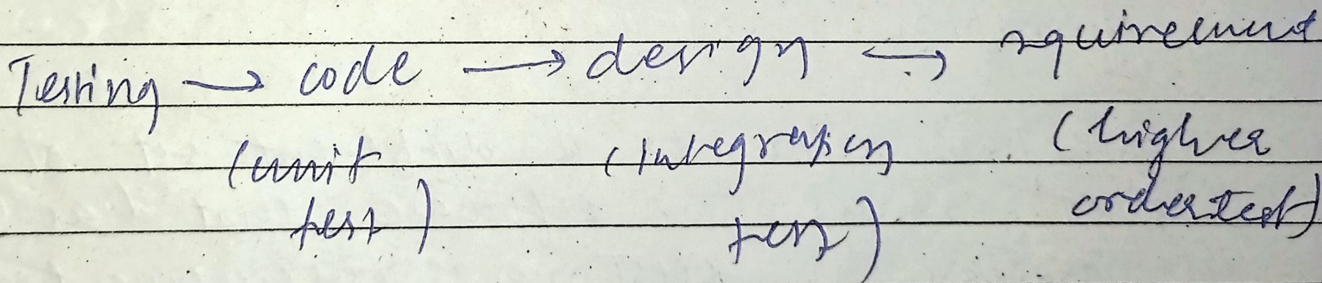
unit testing

code

design

requirement

system requirement



White box testing –

White box Testing sometime called glass box testing or structural testing is a test case design philosophy that uses the control structure describe as a part of component level derive test cases.

using this we can derive –

1) guarantee that all independent path within module have been exercised once

2) exercise all logical decisions on their true and false sides

3) execute all loops at their boundaries and within their operational bound.

Basic Path testing –

Basic path testing is a white box testing technique proposed by Tom McCabe.

The basic path method enables test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basic set of test execution path.

Before basic path testing we need to make a simple notation representation of control flow called a flow graph. Draw only when logical structure is complex.

→ each circle called flow graph node represent one or more procedural statement.

a sequence of process box and a decision diamond can map into a single node.

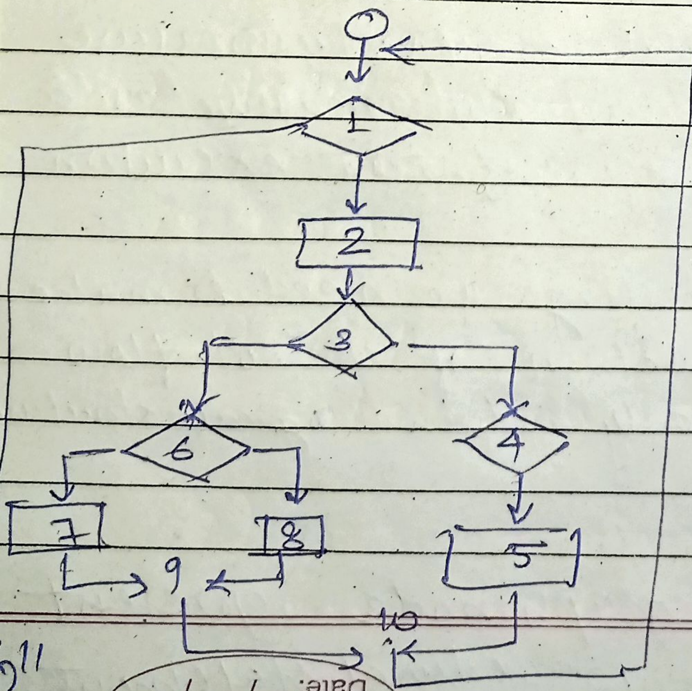
The arrow on flow graph called edges or link, represent flow of control and analogues to flow chart.

an edge must terminate at a node even if the node does not represent any procedural statement.

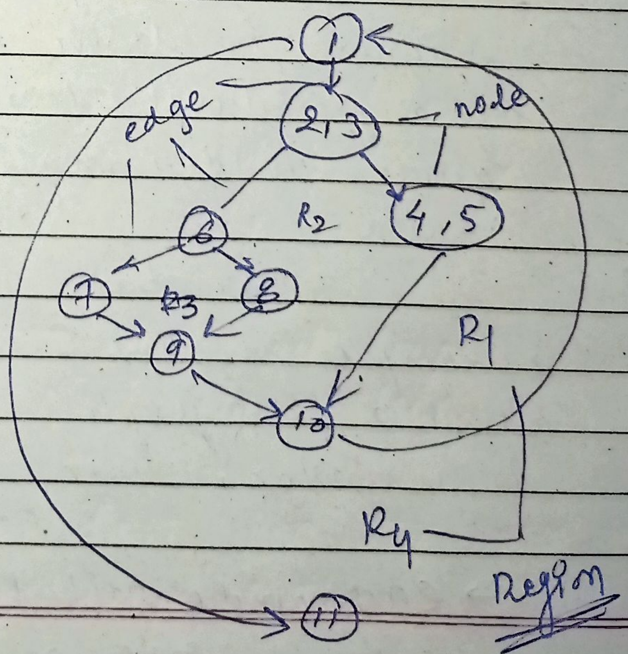
Area bounded by edges and node is known as region.

an Independent path is a any path through the program introduces atleast one new set of processing statement or a new condition, when stated in term of a flow graph, independent path must move along at least one edge that has not been traversed before the path is defined.

flow chart



flow graph



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 introduce new edge.

path → 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11
↳ will not be considered as independent path because it's just a combination of already known path

Control Structure Testing:-

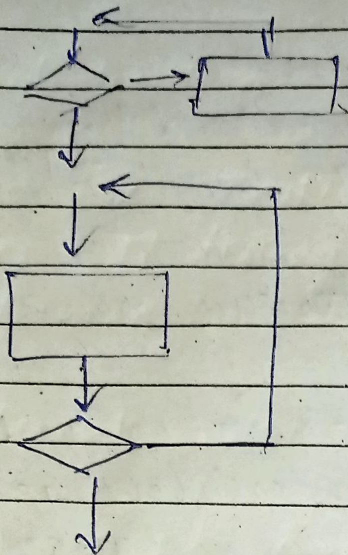
The basic path testing is one of the techniques for control structure testing. although a basic path testing is simple & highly effective

condition testing - is a test case design method that exercise the logical condition contained in a program module.

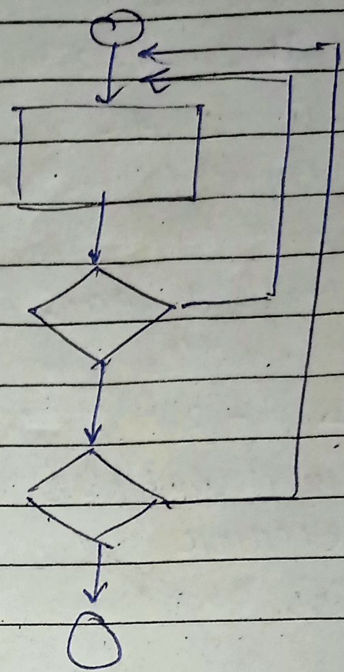
Data flow Testing - select test path of a program according to defination and uses a variable in program.

Loop testing - is a white box Testing technique that focuses on the validation of loop construct two different loops are 1) simple loop & 2) nested loop.

Simple loop



nested loop



⊛ simple loop:— n = maximum numbers of allowable passes

1) skips the loop entirely.
2) only one pass through loop.
3) two passes through loop.
4) m passes —"—
5) n-1, n, n+1, —"—

⊛ Nested loop:— numbers of possible test grow geometrically if level of testing increases.

1) Start at the ~~intermost~~ innermost loop, set all other loops to minimum value

2) conduct simple loop test for the innermost loop while holding loop at their

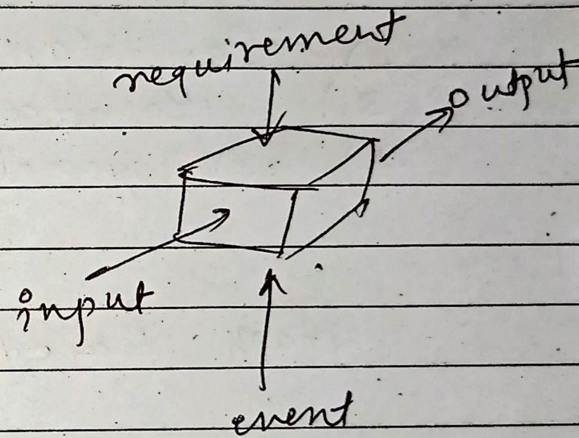
Genius Page: Date: / /

minimum iteration parameter value. Add other test for out of range or excluded values.

3) work outward, conducting test for the next loop, but keeping all other outer loop at minimum value and other nested loops to "typical" value.

4) Continue untill all the loop has been tested

Black Box Testing



also known as behavioural testing.

1) It focuses on the functional requirement of the software

2) functional testing of a component of a system

3) Examine behaviour through input & the corresponding output.

4) Input is properly accepted, output is correctly produced.

Black Box testing attempts to find error in following catagories

1) incorrect or missing function
2) interface error
3) error in data structure or database

4) Behaviour or performance error

5) initialisation & termination error

Black box testing is used during the later stages of testing after white box testing has been performed.

Different Black box technique →

1) Graph Based Testing method
2) Equivalance Partitioning
3) Boundary value analysis
4) orthogonal array testing.

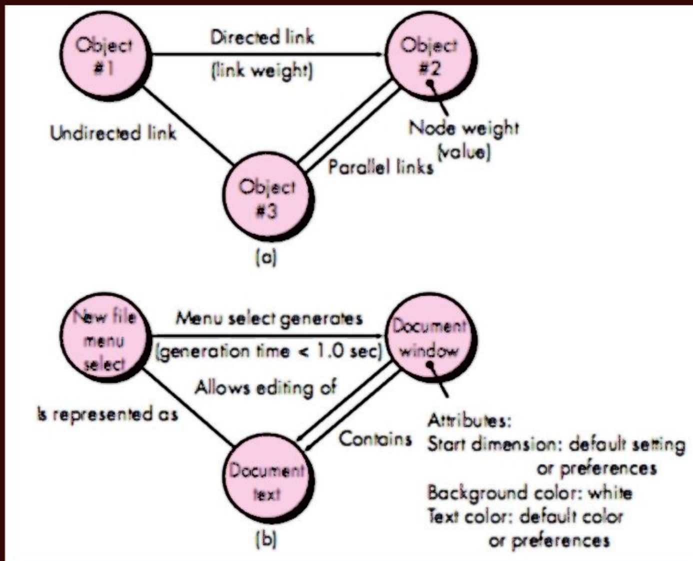
Graph Based testing method -

→ The first step in blackbox testing is to understand the object that are modelled in s/w and the relationship that connects these object

→ once this has been accomplished, the next step is to define a set to define a series of test that verify all object has expected relation between eachothers.

Black box testing techniques - (1) Graph-Based Testing Methods

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



- **Graph Representation**
    - 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),
    - **Link weights** that describe some characteristic of a link

- **The symbolic representation of a graph is shown in Figure.**
- **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.

Black box testing techniques - (2) Equivalence Partitioning Method

- 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 equivalence class represents a set of valid or invalid states for input conditions.
- **Equivalence classes guidelines**
- **If an input condition specifies a range,**
  - one valid and two invalid equivalence classes are

- one valid and two invalid equivalence classes are defined
- Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}

- **If an input condition requires a specific value,**
  - one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- **If an input condition specifies a member of a set,**
  - one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- **If an input condition is a Boolean value,**
  - one valid and one invalid class are define
  - Input: {true condition} Eq classes: {true condition}, {false condition}

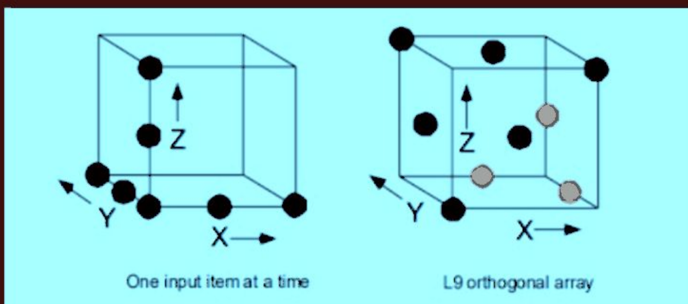
Black box testing techniques - (3) Boundary Value Analysis Technique

- A greater number of errors occurs at the boundaries of the input domain.
- It is for this reason that **boundary value analysis (BVA)** has been developed as a testing technique
- Test both sides of each boundary
- Look at output boundaries for test cases
- Test min, min-1, max, max+1, typical values
- **Example : 1 <= x <=100**
    - Valid : 1, 2, 99, 100
    - Invalid : 0 and 101
- **Guidelines for BVA**
    - 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.
    - 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.

Black box testing techniques - (4) Orthogonal Array Testing

Black box testing techniques - (4) Orthogonal Array Testing

- Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate complete testing.
- The orthogonal array testing method is particularly useful in finding region faults — an error category associated with faulty logic within a software component.
- Consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are 3^3 = 27 possible test cases.
- Phadke suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure...
- **Used when the number of input parameters is small and the values that each of the parameters may take are clearly**



- To illustrate the use of the L9 orthogonal array, consider the send function for a fax application.

- *Four parameters, P1, P2, P3, and P4, are passed to the send function. For example : Function (p1,p2,p3,p4)*
- Each takes on three discrete values P1 takes on values:
    - P1 = 1, send it now
    - P1 := 2, send it one hour later
    - P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.
- If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).

The L9orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3

cases than the exhaustive strategy

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Black Box Testing - Advantage & Disadvantages

- **Advantage**
  - Find missing functionality
  - Independent from code size and functionality.
- **Disadvantages**
  - No systematic search for low level errors.
  - Specification errors not found.

at 1:01:00 PM

Share

3 comments: