# Software Engineering

☆ These notes have been prepared mainly from the following book :-

Software Engineering
by
K.K. Aggarwal, Yogesh Singh

Kindly refer above book for more details.

www.ankurgupta.net

## Software Life Cycle Models :-

### (1) Waterfall Model :-

This model is generally used when :-

(i) No change in requirements

(ii) Deliverables expected at every stage.

(iii) We have carried out a similiar project earlier.

### Phases and Deliverables of Waterfall Model :-

| Phase | Purpose | Deliverables |
|---|---|---|
| System Engineering | Defining the scope of the project. | User Requirements |
| Requirement Analysis | Understanding the functional and non-functional requirements | (i) SRS<br>(ii) Acceptance Test Plan<br>(iii) System Test Plan |
| Design | Creating the structure of the modules | (i) HLD (ii) DLD<br>(iii) ITP, (iv) UTP |
| Coding | Building the software and unit testing | Unit tested code. |
| Testing | Ensuring that requirements are met | Integrated and system tested S/W. |
| Deployment | Assembling, Installation, End-user testing and sign-off by the customer. | User manual. |

Pros and cons of waterfall model:-

Pros:-

(i) Simple and systematic model.
(ii) Follows a disciplined approach.

Cons:-

(i) Not suitable for accomodating any change.
(ii) Potential delay in identifying the risks.
(iii) It does not scale up well to large projects.

(2) Prototyping Model:-

This model is generally used when:-

(i) Complete set of requirements not available.
(ii) Development with initial set of requirements is started.
(iii) Feel of the product with initial requirements expected.

→ SRS is finalized after prototype is ready.

Pros:-

(i) Less technical risks.
(ii) Scope for accomodating new requirements.
(iii) A part of the product is visible at an early stage

Cons:-

Expensive and time consuming.

(3) Iterative Enhancement Model:-

This model delivers an operational quality product at each release, but one that satisfies only a subset of the customer's requirements.

The complete product is divided into releases, and the developer delivers the product release by release.

(4) Evolutionary Development Model:-

It resembles iterative enhancement model. This model differs from iterative enhancement model in the sense that this does not require a useable product at the end of ~~the~~ each cycle.

Here requirements are implemented by category rather than by priority.

Example:- GUI in first phase, queries in another.

→ Useful ~~p~~ for projects using new technology that is not well understood.

(5) Spiral Model:-

This model is generally used when:-

(i) Many risks are expected.
(ii) At each stage of development, there are alternatives and we have to make right decisions.
(iii) Budget is not fixed for the project.

Pros:-

(i) Model for other models.
(ii) Captures potential risks at an early stage.
(iii) Iterative and realistic model.

Cons:-

(i) Requires good expertise in risk management and project planning.
(ii) Projected cost is revisited and revised every round during planning.

(6) RAD (Rapid Application Development) Model:-

This model is ~~not~~ generally used when:-

(i) There are tight deadlines.
(ii) High pressure from customer.
(iii) Quick time to market.
(iv) There ~~any~~ are many functionalities
(v) Users have to be involved throughout.

→ Each major function is addressed by a seperate RAD team.

(7) Agile Methodology:-

→ Development team focuses on construction than design and documentation.

→ Intended to deliver software quickly.

www.ankurgupta.net

# Software Requirements Analysis and Specifications

→ Requirements describe the 'what' of a system, not the 'how'.

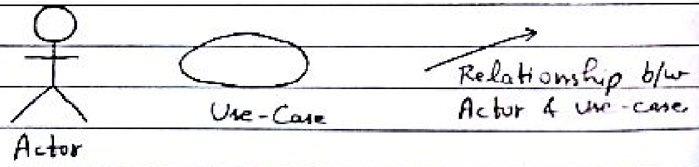
→ The input is the problem statement prepared by the customer.

## Crucial Process Steps:-

### (i) Requirements Elicitation (Gathering):-

Requirements are identified with the help of customer and existing system processes.

Use Case Diagrams, that use a combination of text and pictures, are used to improve the understanding of requirements.



### Types of Requirements:-

(i) Functional Requirements:-

Describe what the software has to do. They are often called product features.

(ii) Non-Functional Requirements:-

Describe how well the software does what it has to do.

### (ii) Requirements Analysis:-

Requirements are analysed in order to identify inconsistencies, defects and ambiguities.

Produces a structured requirements specification made of graphical notations.

(a) Data Flow Diagrams:-

DFDs show the flow of data through a system.

(b) Entity Relationship Diagrams:-

Its a detailed logical representation of the data for an organization.

(c) Data Dictionaries:-

Data dictonaries are repositories to store information about all data items defined in DFDs.

### (iii) Requirements Documentation:-

This is the end product of requirements elicitation and analysis. The document is known as Software Requirements Specification (SRS).

SRS should address the following:-

(a) Functionality:-

What the software is supposed to do?

(b) External Interfaces:-

How does the software interact with people, the system's hardware, other hardware and other software?

(c) Performance :-
What is the speed, availability, response time, recovery time, etc. of various software functions?

(d) Design Constraints imposed on an implementation:
Implementation language, operating environment etc.

Characteristics of a good SRS :-

(1) Correct, (2) Unambiguous, (3) Complete,
(4) Consistent, (5) Verifiable, (6) Modifiable,
(7) Traceable.

# Acceptance Test Plan and System Test Plan is ~~algo~~ also generated in requirements analysis phase.

# SRS should not describe any design or implementation details.

#

## Software Design

Design :-
In this phase, the designer plans "how" a software system should be developed in order to make it functional, reliable, understandable, modifyable and maintainable.

The purpose of design phase is to produce a solution to a problem given in SRS document.

Modularity :-
Desirable properties of a modular system include:-
(i) Each module is a well defined subsystem that is potentially useful in other applications.
(ii) Modules can be separately compiled and stored in a library.

Module Coupling :-
Coupling is the measure of the degree of interdependence between modules.

Loosely Coupled systems are made up of modules which are relatively independent.
Highly Coupled systems share a great deal of dependence between modules.
Uncoupled modules have no interconnections at all.

# A good design will have low coupling. Thus, interfaces should be carefully specified in order to keep low value of coupling.

Types of coupling :-

(i) Data Coupling :-
Module A and B communicate by only passing of data.

(ii) Stamp Coupling :-
Module A and B communicate by passing complete datastructure.

(iii) Control Coupling :-
Module A and B communicate by passing of control information, i.e. flags.

(iv) External Coupling :-
A module has a dependency to other module, external to the software being developed.

(v) Common Coupling :-
Module A and B have shared data.

(vi) Content Coupling :-
When control is passed from one module to the middle of another.

| | |
|---|---|
| Data Coupling | Best (Low) |
| Stamp Coupling | ↑ |
| Control Coupling | |
| External Coupling | |
| Common Coupling | |
| Content Coupling | Worst (High) |

Module Cohesion :-
Cohesion is a measure of the degree to which the elements of a module are functionally related.

An important design objective is to maximize the module cohesion and minimize the module coupling.

Types of Cohesion :-
Given a procedure that carries out operations X and Y, we can describe various forms of cohesion between X and Y :-

(i) Functional Cohesion :-
X and Y are part of a single functional task.

(ii) Sequential Cohesion :-
X outputs some data which forms the input to Y.

(iii) Communicational Cohesion :-
X and Y both operate on the same input data or contribute towards the same output data.

(iv) Procedural Cohesion :-
It occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

(v) Temporal Cohesion :-

X and Y both must perform around the same time.

(vi) Logical Cohesion :-

X and Y perform logically similiar operations.

(vii) Coincidental Cohesion :-

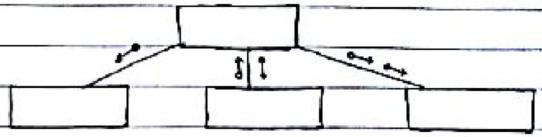
X and Y have no conceptual relationship other than shared code.

| | |
|---|---|
| Functional Cohesion | Best (high) |
| Sequential Cohesion | ↑ |
| Communicational Cohesion | |
| Procedural Cohesion | |
| Temporal Cohesion | |
| Logical Cohesion | |
| Coincidental Cohesion | Worst (Low) |

Structure Chart :-

The structure chart is one of the most commonly used method for system design. It partitions a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design.

Here black boxes are arranged in hierarchical format as shown :-



# Software Project Planning

Software planning begins before technical work starts, continues as the software evolves and culminates only when the software is retired.

The project planning must incorporate the major issues like :-

(i) Size and Cost Estimation
(ii) Scheduling
(iii) Project Monitoring
(iv) Resources Requirement
(v) Risk Management.

## Size Estimation :-

### (1) Lines of Code (LOC) :-

A line of code is any line of a program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declarations, and executable and non-executable statements.

### (2) Function Count :-

It measures functionality from the users point of view, that is, on the basis of what the user requests and receives in return from the system.

It is independent of the language and tools used for implementation.

$$\boxed{FP = UFP \times CAF}$$

where, FP = Function Point
UFP = Unadjusted Function Point
CAF = Complexity Adjustment Factor.

## Cost Estimation :-

### The Constructive Cost Model (COCOMO) :-

$$\boxed{E = a\,(KLOC)^{b}}$$

$$\boxed{D = c\,(E)^{d}}$$

E ⇒ Effort in Person-Months.
D ⇒ Development time in months.

$$\boxed{\text{Average staff size} = \frac{E}{D} \text{ Persons}}$$

$$\boxed{\text{Productivity} = \frac{KLOC}{E} \text{ KLOC/P.M}}$$

## Risk Estimation :-

Q:- Consider a project that has 0.5% probability of an undetected fault that would cost the company Rs. 1,00,000 in fine. What would be the risk exposure?

Solution :-

$$\text{Risk exposure} = 1,00,000 \times \frac{0.5}{100}$$

$$= 500 \text{ Rs. } \underline{\text{Ans}}$$

Q:- In a software, mean time to failure is equal to mean time to repair. What is the availability of the software?

Solution :-

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

$$= \frac{x}{(x+x)} \times 100\% \rightarrow 50\% \ \underline{\text{Ans}}$$

Q:- A software was tested using the error seeding strategy in which 10 errors were seeded in the code. When the code was tested using the complete test suite, 4 of the seeded errors were detected. The same test suit also detected 100 non-seeded errors. What is the estimated number of undetected errors in the code after this testing?

Solution :-

$$\text{Total no. of errors} = 100 \times \frac{10}{4}$$

$$= 250$$

$$\text{Undetected errors} = \text{Total errors} - \text{Detected Errors}$$

$$= 250 - 100$$

$$= 150 \ \underline{\text{Ans}}$$

Mean Time Between Failure (MTBF) = MTTF + MTTR

$$\text{Availability} = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$$

# Software Testing

Testing is the process of executing a program with the intent of finding errors.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

~~Complete or exhaustive~~

Complete or exhaustive testing is just not possible.

## Verification and Validation :-

(i) Verification :-

Checking the software with respect to specifications.

(ii) Validation :-

Checking the software with respect to customer's expectations.

## Acceptance Testing :-

→ Used when the software is developed for a specific customer.

→ Testing is conducted by the customer to validate all requirements.

## Alpha and Beta Testing :-

→ Used when the software is developed as a product for anonymous customers.

→ The alpha tests are conducted at the developer's site by a customer.

→ The beta tests are conducted by the customer at their site.

### Unit Testing :-

Tests the functionality within the module.

### Integration Testing :-

Tests are focused more on interaction between modules.

### System Testing :-

Tests the software as part of the bigger system for which it was created.

### Regression Testing :-

It consists of running the corrected system against tests which the program had already passed successfully. to ensure that in process of modifying the existing system, the original functionality of the system was not disturbed.

### Performance Testing :-

Tests the non-functional requirements of the system.

(a) Load Testing :-

Testing with many users accessing the system at the same time.

(b) Stress Testing :-

Testing to identify the number of users the system can handle at a time before breaking down.

(c) Endurance Testing :-

Testing for a long time for reliability.

(d) Spike Testing :-

The system is stressed for a short duration.

### Functional Testing :-

Functional testing refers to testing, which involves only observation of the output for certain input values. There is no attempt to analyse the code, which produces the output.

It is also referred to as black box testing.

### (i) Boundary Value Analysis :-

The basic idea of boundary value analysis is to use input variables values at their :-

(i) minimum,
(ii) just above minimum,
(iii) a nominal value,
(iv). just below their maximum,
(v) maximum.

### (ii) Equivalence Class Testing :-

In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, that the test of a representative value of each class is equivalent to a test of any other value.

That is, if one test case in a class detects an error, all other test cases in the class would be expected to find same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the class would find an error.

→ The idea is to choose at least one element from each equivalent class.

→ We should not forget to have equivalent classes for invalid inputs.

→ Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domains.

Example :- Consider the program for the determination of nature of roots of a quadratic equation. Identify the equivalence class test cases for output domain.

Solution :- Output domain equivalence class test cases can be identified as follows :-

$O_1 = \{<a, b, c> : \text{Not a quadratic equation if } a = 0\}$

$O_2 = \{<a, b, c> : \text{Real roots if } (b^2 - 4ac) > 0\}$

$O_3 = \{<a, b, c> : \text{Imaginary roots if } (b^2 - 4ac) < 0\}$

$O_4 = \{<a, b, c> : \text{Equal roots if } (b^2 - 4ac) = 0\}$

| Test Case | a | b | c | Expected Output |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not a quadratic equation |
| 2 | 1 | 50 | 50 | Real Roots |
| 3 | 50 | 50 | 50 | Imaginary Roots |
| 4 | 50 | 100 | 50 | Equal Roots |

(iii) Decision Table Based Testing :-

Decision tables are useful for describing situations in which a number of combinations of actions are taken under varying set of conditions. These are used to represent and analyze complex logical relationships.

| | | Entry | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Condition Stub | $c_1$ | True | | | | False | | |
| | $c_2$ | True | | False | | True | | False |
| | $c_3$ | True | False | True | False | True | False | — |
| | $a_1$ | x | x | | | x | | |
| Action Stub | $a_2$ | x | | x | | | x | |
| | $a_3$ | | x | | | x | | |
| | $a_4$ | | | | x | | x | x |

When conditions $c_1$, $c_2$ and $c_3$ are all true, action $a_1$ and $a_2$ occur. When conditions $c_1$ & $c_2$ are true and $c_3$ is false, actions $a_1$ and $a_3$ occur.

Thus decision table is used to generate test cases.

### Structural Testing :-

A complementary approach to functional testing is called structural/white box testing. It permits us to examine the internal structure of the program.

### Static White Box Testing :-

If we want to test the program without running it.

### Dynamic White Box Testing :-

If we want to test the program by running it.

### Path Testing :-

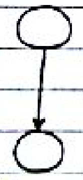
This type of testing involves :-

(i) Generating a set of paths that will cover every branch in the program.

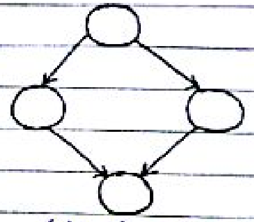
(ii) Finding a set of test cases that will execute every path in this set of program paths.

### Flow Graph :-

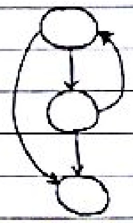
The flow graph is a directed graph in which nodes are statements and edges represent flow of control.



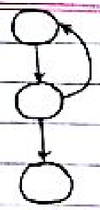
(Sequence)



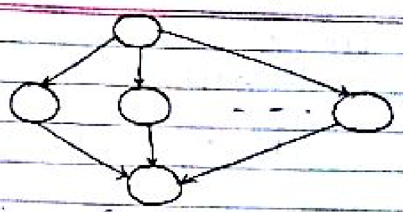
(If-then-else)



(While loop)



(Do-while loop)



(Switch Statement)

### DD Path Graph :-

The DD Path graph is known as decision to decision path graph. Here, we concentrate only on decision nodes. The nodes of flow graph which are in sequence are combined into a single node.

Hence, DD Path graph is a directed graph in which nodes are sequences of statements and edges ~~are~~ represent control flow between nodes.

### Independent Paths :-

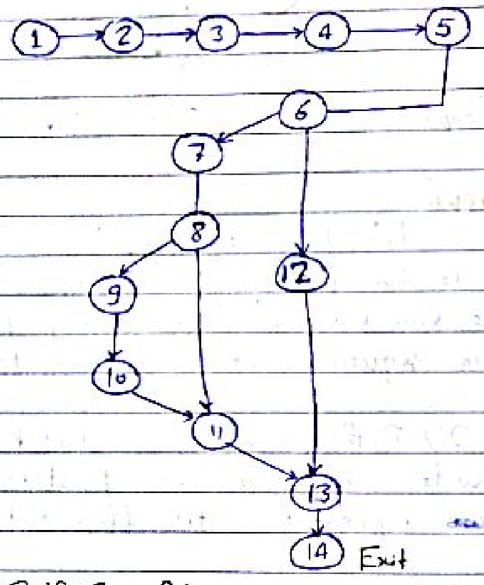
An independent path is any path through the DD Path graph that introduces at least one new set of processing statements or new conditions. Therefore an independent path must move along at least one edge that has not been traversed before the path is defined.

We are interested to execute all independent paths at least once during path testing.

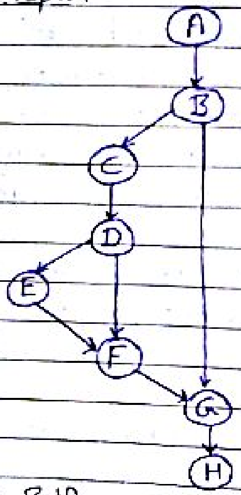
www.ankurgupta.net

Example :-

Flow Graph :-



DD Path Graph :-



Independent Paths are :-

(1) ABGH, (2) ABCDFGH, (3) ABCDEFGH

It provides a quantitative measure of the logical complexity of a program.

Cyclomatic Complexity :-

It is also known as structural complexity. This approach is used to find the number of independent paths through a program.

(1) For a graph G with n-vertices, e edges, and k connected components :-

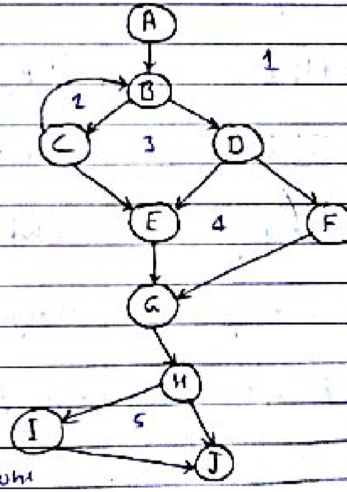
Cyclomatic Complexity $V(G) = e - n + 2k$

(2) Cyclomatic complexity of a flow graph G is equal to the number of predicate (decision) nodes plus one.

$$\boxed{V(G) = \pi + 1}$$

(3) Cyclomatic complexity is equal to the number of regions of the flow graph

Example :-



(1) $V(G) = e - n + 2k$
$= 13 - 10 + 2 \times 1$
$= 5$

(2) $V(G) = \pi + 1$
$= 4 + 1$
$= 5$

(3) V(G) = Number of regions
= 5

Independent Paths are :-

(1) ABCEGHJ, (2) ABCBCEGHJ, (3) ABDEGHJ, (4) ABDFGHJ, (5) ABDFGHIJ

## Defect Detection using Reviews :-

Examination of software work products by the author's peers to identify defects and areas where changes are required.

Pros :-

(1) 100% code coverage

(2) 2 to 5 times more defects.

(3) Mix of errors like maintainability can be detected.

## Testing and the life cycle :-

| Phase | Deliverables |
|---|---|
| (1) Requirement Analysis | User acceptance plan & System test plan |
| (2) HLD | Integration Test Plan |
| (3) DLD | Unit test plan |
| (4) Coding | Unit tested code |
| (5) Testing | Integration & System tested code |
| (6) Acceptance | Sign-off by the customer. |

# It is better to prevent the injection of defects rather than finding them and fixing them.

## Integration Testing :-

(1) Top Down :-

→ Top level modules are developed and tested first.

→ Dummy bottom level module called "stub" is required.

(2) Bottom Up :-

→ Bottom level modules are developed and tested first.

→ Dummy top level module called "Driver" is required.

(3) Sandwitch :-

→ Combines both bottom-up and top-down integration.

→ A layer is identified in between.

→ Above this layer top-down approach is followed and below this layer bottom up is followed.

(4) Big Bang :-

→ Test all modules independently and then integrate the modules to form the software.

→ The integrated software is tested as a whole.

# Testing phase requires largest manpower.

## Maintenance :-

Software maintenance is a broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization.

Any work done to change the software after it is in operation is considered to be maintenance work.

## Types of Maintenance :-

### (1) Corrective Maintenance :-

This refers to modifications initiated by defects in the software.

### (2) Adaptive Maintenance :-

It includes modifying the software to match changes in the ever-changing environment.

### (3) Perfective Maintenance :-

It means improving processing efficiency or performance, or restructuring the software to improve changeability.

### (4) Preventive Maintenance :-

There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software. The work is required to be done to maintain it or to reduce it. This work may be named as preventive maintenance.

## Software Configuration Management :-

A Software Configuration Management tool helps in maintaining different versions of the configurable items.
Example - VSS.