T.Y.B.Sc.
(Computer Science)

(Revised Syllabus)

Paper IV

DBMS-II & Software Engineering

Syllabus

UNIT I

Decomposition: Functional dependency, Lossless-Join decomposition, Multi valued dependency and fourth normal form, Join dependency, Fifth normal form.

Concurrency Control: Concept of a transaction, ACID properties, Serial and serializable schedules, Conflict and View serializabilty, Precedence graphs and test for conflict seralizability.

Enforcing serializability by locks: Concept of locks, the locking scheduler, Two phase Locking, upgrading and down grading locks, Concept of dead locks, Concurrency control by time stamps, The Thomos Write rule.

Crash Recovery: ARIES algorithm. The log based recovery, recovery related structures like transaction and dirty page table, Write-ahead log protocol, check points, recovery from a system crash, Redo and Undo phases.

UNIT II

Sequences: creating sequences, referencing, altering and dropping a sequence.

Fundamentals of PL/SQL: Defining variables and constants, PL/SQL expressions and comparisons: Logical Operators, Boolean Expressions, CASE Expressions Handling, Null Values in Comparisons and Conditional Statements, PL/SQL Datatypes: Number Types, Character Types, Boolean Type, Datetime and Interval Types.

Overview of PL/SQL Control Structures: Conditional Control: IF and CASE Statements, IF-THEN Statement, IF-THEN-ELSE Statement, IF-THEN-ELSIF Statement, CASE Statement, Iterative Control: LOOP and EXIT Statements, WHILE-LOOP, FOR-LOOP, Sequential Control: GOTO and NULL Statements, Concept of nested tables.

Query evaluation: System Catalog, Evaluation of relational operators like selection, projection, join and set, introduction to query optimization.

Cursors: Concept of a cursor, types of cursors: implicit cursors; explicit cursor, Cursor for loops, Cursor variables, parameterized cursors,

Transactions in SQL: Defining a transaction, Making Changes Permanent with COMMIT, Undoing Changes with ROLLBACK, Undoing Partial Changes with SAVEPOINT and ROLLBACK, Defining read only transactions, explicit locks: transaction and system level, Choosing a Locking Strategy: ROW SHARE and ROW EXCLUSIVE Mode.

Unit III

Project management: Revision of Project Management Process, Role of Project Manager, Project Management Knowledge Areas, Managing Changes in requirements, Role of software Metrics

Size & Effort Estimation –Concepts of LOC & Estimation, Function Point, COCOMO Model, Concept of Effort Estimation & Uncertainty

Project Scheduling Building WBS, Use of Gantt & PERT/CPM chart, Staffing

Configuration Management Process & Functionality & Mechanism, Process Management, CMM & its levels, Risk Management & activities

Management of OO software Projects - Object oriented metrics, Use-Case Estimation Selecting development tools, Introduction to CASE

Changing Trends In Software Development - Unified Process, Its phases & disciplines, Agile Development – Principles & Practices, Extreme programming- Core values & Practices Frameworks, Components, Services, Introduction to Design Patterns, Open Source

Unit IV

Software Testing: Introduction to Quality Assurance, Six Sigma Testing Fundamentals, Common Terms(like Error, Fault, Failure, Bug, Crash) Objectives of testing, Challenges in Testing, Principles of Testing

Static Testing – Introduction & Principles

Types of Testing – Levels of testing such as Unit testing, Integration testing, System testing, Validation Testing, Acceptance testing,

Types of testing such as Black box, White Box, Functional, Performance, Regression, Acceptance, Volume, Stress, Alpha, Beta testing

Black Box Testing: Introduction, Equivalence partitioning, Boundary-value analysis, Robustness testing, Cause Effect Graph,

White Box Testing: Statement Coverage, Branch/Decision Coverage, Condition Coverage, Graph Matrix, Cyclomatic complexity, Mutation Testing

Object Oriented Testing & Web site testing – Object Oriented Testing Strategies, Testing methods, Overview of web site testing

Planning Software Testing – Test Plan, Test Plan Specification, Test Case Execution and Analysis, Defect logging and tracking

CONTENTS

1

FUNCTIONAL DEPENDENCY AND DECOMPOSITION

Unit Structure

1.0 OBJECTIVES

The objective of this chapter is

- To understand the basics functional dependency.
- To understand lossless join decomposition.
- To know multivalued dependency and fourth normal form.
- To understand join dependency and fifth normal form.

1.1 INTRODUCTION

An attribute B in a relational database is said to be functional dependent on some other attribute A if each value in column A determines one and only one value in column B. A relation can have certain anomalies due to some functional dependencies. By decomposing the relations into smaller relations we can remove these anomalies and bring the relation into what we call as normalized form.

1.2 FUNCTIONAL DEPENDENCY

Consider a relation R defined over a set of attributes (A_1, A_2 ….A_n) and let X and Y be subset of (A_1, A_2 ….A_n) then

$$X \rightarrow Y$$

It means that Y is functionally dependent on X if and only if, whenever two or more records in R agree on their X value i.e. if each X value in (A_1, A_2 ….A_n) has associated with it one Y value in (A_1, A_2 ….A_n).

Example: Consider the following relation:

J	K	L
X	1	2
X	1	3
Y	1	4
Y	1	3
Z	2	5
P	4	1

In above relation

$J \rightarrow K,\ L \rightarrow K$

$J \nrightarrow L, K \nrightarrow J$

For two same values of J, L have different values. Similarly for two same values of K, J have different values. So these are not functional dependencies.

Example: Consider the following supplier relation:

Supplier Relation

S#	P#	City	Qty
S_1	P_1	London	100
S_1	P_2	London	100
S_2	P_1	Paris	200
S_2	P_2	Paris	200
S_3	P_2	Paris	300
S_4	P_2	London	400
S_4	P_4	London	400
S_4	P_5	London	400

In the above relation we can observe:

- Supplier relation satisfies following functional dependency:

$$S\# \rightarrow city$$

In the above functional dependency every tuple with a given value of S# has same value for city.

- Similarly there is another functional dependency

$$S\#, P\# \rightarrow city$$

- **Trivial Functional Dependencies:**

A functional dependency of form

$$X \rightarrow Y$$

where $X \rightarrow Y$ is said to be trivial functional dependency if Y is subset of X.

Example: In above relation, the dependency $S\#, P\# \rightarrow S\#$ is trivial function dependency.

Example: For the following relation, list all the functional dependencies.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Functional dependencies are as follows

$$A \rightarrow C$$

$$AB \rightarrow D$$

$AB \rightarrow A$ (Trivial Functional dependency)

- **Candidate Key:** Consider a relation R on a set of n attributes, then a candidate key K is set of one or more fields if it satisfies following properties

  (i) Uniqueness: K uniquely identifies each tuple (record) i.e. no two tuples can have same value of K

  (ii) Non-redundancy: K is non redundant, i.e. no proper subset of K has uniqueness property.

  Note: There is no subset of key which has above properties and in general more than one candidate key may exist.

  Example: In the following relation

  Student (name, father-name, course, enrollment-no, grade)

  Candidate key assuming they give and unique identification

  {Name, father-name}, {Enroll-number}

  Non candidate key

  {Name, enrollment no} violates property 2

  {Course} violates property 1

- **Types of Attributes:**

  (a) Prime Attribute: A set of attributes that participates in the candidate key.

  (b) Non Prime Attribute: A set of attributes that do not participate in the candidate key

  For example:

  Student (name, father- name, enroll-number, grade)

  where key be enroll-number then

Prime attribute: enrollment-number

Non prime attribute: name, father-name, course, grade

- **Primary Key:** A designated candidate key is primary key. It has following properties

  (i) It is fully defined i.e. no unknown values.

  (ii) It cannot be null.

  Example: Assume that candidate key for student is enrollment number. Therefore enrollment number satisfies that every student must have enrollment number, it should be prime attribute and other name, course, father-name, and grades are non attribute then in relation

  Student (enrollment number, name, father-name, course, grade)

  Enrollment number is candidate key. Hence, enrollment number is primary key.

- **Alternate Key:** The candidate key which is not chosen as primary key is known as alternate key.

  In student table, father name is also candidate key but it is not chosen as primary key. Therefore it is alternate key and it can be NULL.

- **Super Key:** Collection of attributes which is uniquely identified but may not be minimal.

  For example, in student relation enrollment number and name together can be uniquely identified but it is not minimal.

1.3 LOSSLESS JOIN DECOMPOSITION

Let r is relation on relational schema R and let $R_i = \pi_{Ri}(R)$ for i=1, 2,…,n

and R = R_1 join R_2 …join R_3

The decomposition of relational schema R = {A_1, A_2,…, A_n} in its replacement by a set of relational schemas {R_1, R_2, R_3,…, R_n} such that R_1 join R_2 join R_3….join R_n = R

Decompositions are usually of two types which are as follows

- **Lossless Join Decomposition:** It is as follows

  R is relation and F is set of FD's. Let R be decomposed into R_1 and R_2. Decomposition is lossless

  If $R_1 \cap R_2 = R_1$

  OR

  $R_1 \cap R_2 = R_2$

- **Lossy Decomposition**: In this decomposition we are unable to obtain the original relation when we join them all together.

  Example: Supplier relation

S#	City	Status
53	Mumbai	30
55	Delhi	30

Let us consider the following decomposition

R_1

S#	Status
53	30
55	30

and R_2

City	Status
Mumbai	30
Delhi	30

On joining them R_1 join R_2

S#	City	Status
53	Mumbai	30
53	Delhi	30
55	Mumbai	30
55	Delhi	30

We cannot get original relation supplier. Therefore decomposition is lossy.

Now consider decomposition

R_1

S#	Status
53	30
55	30

R_2

S#	City
53	Mumbai
55	Delhi

Now R_1 join R_2 results

S#	City	Status
53	Mumbai	30
55	Delhi	30

The result is original relation supplier. Therefore decomposition is lossless.

Example:

Emp. Dpt. = (emp no., name, job, dept. no, dloc, dname)

$$F = \{deptno \rightarrow dname, deptno \rightarrow dloc, empno \rightarrow deptno, empno \rightarrow job\}$$

If we decompose Emp.Dpt. into two relations as following

Emp = {empno, name, job}

Dept = {deptno, dname, dloc}

Decomposition is lossless join $deptno \rightarrow dname, dloc (union)$

$deptno \rightarrow dname, dloc, deptno$ (augmentation)

$Emp \cap dept = \{deptno\} \rightarrow dept$

Therefore it lossless decomposition

Now other possibility of decomposition into two relations

$Emp = \{deptno, dloc, dname, job\}$

$Dept = \{deptno, dloc, dname, job\}$

$Emp \cap Dept = \{job\} \rightarrow$ Dept or Emp

There are above decomposition is lossy

1.4 MULTIVALUED DEPENDENCY AND FOURTH NORMAL FORM

Functional dependency relates a collection or attributes to a single value of others. Functional dependency is special case of multi-valued dependency. If a set determined by multi-valued dependency is restricted to singleton set, multi-valued dependency reduced to functional dependency.

Multi-valued dependency holds all instances of time.

Formally MVD (Multi-valued Dependency) can be defined as

$X \rightarrow\rightarrow Y$ holds if

$Yx_z = Yx_z$

Equivalently, if two tuples t_1 and t_2 exists in R such that $t_1[x] = t_2[x]$ then two tuples t_3 and t_4 must exists in R such that

$t_3[x] = t_4[x] = t_1[x] = t_2[x]$

$t_3[y] = t_1[y]$ and $t_4[y] = t_2[y]$

$t_3[z] = t_2[z]$ and $t_4[z] = t_1[z]$

where z denotes $(R - (x \cup x))$

- **Fourth Normal Form (4NF):**

A relation is in 4NF when a non-trivial multi-valued dependency $A \rightarrow\rightarrow B$ holds then A is super key.

A relation in 4NF is also in 3NF

Example: Consider the following relation

Course_ID	Instructor	Textbook
CS404	Clay	Korth
CS404	Clay	Date
CS404	Drake	Korth
CS404	Drake	Date

By replacing the multi-valued attributes in tables by themselves, we can convert the above table to 4NF.

Course_Inst (Course_ID, Instructor)

Course_Text (Course_ID, Textbook)

The above relations are in 4NF.

1.5 JOIN DEPENDENCY AND FIFTH NORMAL FORM

A join dependency (A, B, C, D) on R is implied by a candidate key of R if and only if each of R (A, B, C, D) is super key.

Consider following example where supplier relation apply additional constraints. There projections are as follow

(i) R_1 (SNAME, PARTNAME)

(ii) R_2 (SNAME, PROJNAME)

(iii) R_3 (PARTNAME, PROJNAME) of supply relation.

If the constraints hold for a tuple

R_1

SName	Name
Smith	Bolt
Smith	Nut
Adam	Bolt
Walton	Nut
Adam	Nail

R_2

SName	Proj Name
Smith	X
Smith	Y
Adam	Y
Walton	Z
Adam	X

R_3

Part Name	Project Name
Bolt	X
Nut	Y
Bolt	Y

Nut	Z
Nail	X

The R_1, R_2, R_3 are in 5NF because there is no join dependency. We can see it by applying natural join to all three relations.

- **Fifth Normal Form (5NF):** It is also known as projection join normal form (PJNF) is level of data base normalization design to reduce redundancy in relational data base recording multi-valued facts by isolating semantically related multiple relationships.

  A table (relation) is said to be in 5NF if and only if every join dependency in it is implied by candidate keys.

1.6 LET US SUM UP

We learnt about functional dependency in this chapter. Then we studied lossless join decomposition and the process involved in it. Multi valued dependency was then discussed followed by definition of fourth normal form. Finally we learnt about the join dependency and fifth normal form.

1.7 REFERENCES AND SUGGESTED READING

(1) Ramakrishnam, Gehrke, "Database Management Systems", McGraw- Hill.
(2) Elmasri and Navathe, "Fundamentals of Database Systems", Pearson Education.
(3) Peter Rob and Coronel, "Database Systems, Design, Implementation and Management", Thomson Learning

1.8 EXERCISE

1. Explain functional dependency.

2. Explain the closure of set of functional dependency. If closure of attribute contains all the attribute of set then it is super key explain.
3. Explain lossless join decomposition with example.
4. What is multi-valued dependency explain with example?
5. Why we need fourth normal form? Explain with example.
6. What is join dependency explain with example?
7. What is fifth normal form explain with example?

2

TRANSACTION MANAGEMENT AND CONCURRENCY CONTROL

Unit Structure

2.0 OBJECTIVES

The objective of this chapter is

- To understand the basics concept of transaction
- To understand ACID properties
- To understand serial and serializable schedule
- To understand conflict and view serializability
- To understand precedence graph and test for conflict serializabilty

2.1 INTRODUCTION

Transactions are means for simplifying the development of distributed multiuser enterprise applications. By enforcing strict rules on an application's ability to access and update data, transactions ensure data integrity. A transactional system ensures that a unit of work either fully completes or the work is fully rolled back. Transactions make an application programmer free from dealing with the complex issues of failure recovery and multiuser programming.

2.2 CONCEPT OF TRANSACTION

Transaction is unit of work in the real world. For example, promote employee and withdraw money from bank account.

Data base technology captures this in notion of database transaction. A database transaction is unit of program execution that operates on database and performs unit of work. Transactions are achieved through Data Manipulation Language. There are two types of DML which is required for transactions:

- **Non procedural DML**: It allows the user to specify what data is required without specifying how it is to be obtained.
- **Procedural DML**: It allows the user to specify what data is needed and how to obtain it.

2.3 ACID PROPERTIES

ACID (atomicity, consistency, isolation, durability) is a set of properties that guarantee the reliability of processing of database transactions. It is one of the oldest and most important concepts of database theory. It sets out the requirements for the database reliability.

The ACID properties allow safe sharing of data. It makes possible to achieve transactions using computer systems to be accurate without which the potential for inaccuracy would be huge.

The ACID properties are as follows:

- **Atomicity**: Either all operations are reflected to database or none are.
- **Consistency**: Each transaction starts and leaves the database in consistent state.
- **Isolation**: Execution one transaction is isolated from the others.
- **Durability**: If transactions commits then changes persists.

Example: Consider a bank where are a set of transactions that access and updated accounts. Let it is transaction that transfer 50 Rs from account A to Account B

The transactions are defined as follows:

T1: read (A);

A: = A-50;

Write (A)

Read (B)

B: = B+50;

Write (B);

Consider each of ACID property in above example.

- **Atomicity**: Let A = 100 Rs and B = 2000 Rs at prior transactions. During Ti if it fails after write (A), then A = 950, B = 2000 after transaction. There it violated ACID property. This is not desirable situation because there is a loss of 50 Rs.
- **Consistency**: This requires sum of (A, B) and should be same before transaction and after transaction. But in above case the sum before transaction is 3000 and after transaction is 2950. Therefore data base is not consistent.
- **Isolation**: Database becomes temporarily inconsistent while the transaction to transfer A executes, with deducted value written to A and increased B yet to be written in B.
  Read (A); A = A-50, write (A); user1
  Read (B); B = B+50; write (B); user2
  The two transaction if not running concurrently then we get the inconsistent data.

- **Durability**: It guarantees that when a transaction completed successfully then all updates that is carried on database persist; even if system failure after the transaction completed the execution.
  Read (A); A = A-50; write (A) system failure. Then data is inconsistence, hence system is not durable.

2.4 SERIAL AND SERIALIZABLE SCHEDULE

A schedule is order of execution of actions of the transactions executing concurrently. Consider two transactions T1 and T2.

- **Serial Schedule**: All action of T1 occurs before all actions of T2
- **Serializable Schedule**: Action T1 and T2 are inter leaved but schedule behave like a serial schedule.
- **Non Serializable Schedule**: It violates the isolation property of ACID.

For example, if two transactions T1 and T2

T1	T2
Read Bx	Read By
Bx = Bx-50	By = By-60
Update Bx	Update By
Read By	Read Bz
By = By+50	Bz = Bz+60
Update By	Update Bz

It is a serial schedule it can be written as follows:

T1: read Bx

T1: Bx = Bx-50

T1: update Bx

T1: read By

T1: By = By+50

T1: update By

T2: read By

T2: By = By-60

T2: update By

T2: read Bz

T2: Bz = Bz+60

T2: update Bz

A Serializable schedule as follows:

T1: read Bx

T1: Bx = Bx-50

T1: update Bx

T2: read By

T2: By = By-60

T2: update By

T1: read By

T1: By = By+50

T1: update By

T2: read Bz

T2: Bz = Bz+60

T2: update Bz

The above schedule is serializable.

Non Serializable Schedule: A schedule which neither serial nor serializable is called non serializable which is as follows

T1: read Bx

T1: Bx = Bx-50

T2: read By

T2: By = By-60

T1: update Bx

T1: read by

T1: By = By+50

T1: update By

T2: read Bz

T2: Bz = Bz+60

T2: update Bz

T2: update By

Above transaction is non-serializable.

2.5 CONFLICT AND VIEW SERIALIZABILITY

Conflict Serializibility: If schedule is serializable in conflict operations then it is conflict serializable.

View Serializibility: Suppose there are two schedules S1 and S2 of the same set of transactions and if for every read action in one of the schedules, its source is the same in the other schedule, we say that S1 and S2 are view-equivalent. Surely, view equivalent schedules are truly equivalent; they each do the same when executed on any one database state. If a schedule S is view-equivalent to a serial schedule, we say S is view serializable.

2.6 PRECEDENCE GRAPH

To check whether a transaction is serializable or not precedence graph is used. We can draw a precedence graph of the transactions.

A precedence graph contains two components:

- Nodes: represents the transaction
- Edges: An edge follows the following rules.

We can draw a precedence graph by using following steps:

- For each node transaction A that shares lock a resource find a transaction B that exclusively locks it, then draw an edge from B.
- For each transaction A that exclusively locks a resource, find a transaction B that exclusively locks it. Draw edge from A to B.
- Determine all transactions C that shared the resource after A unlocks exclusive lock. Draw an edge A to these C.

If graph contains cycle then schedule in non serializable some statements of transactions proceeds and other succeeds those in other transactions. Hence the schedule is non serial.
Examples of precedence graphs are given in the next section.

2.7 TEST FOR CONFLICT SERIALIZABILITY

We can check a transaction to be conflict serializable by using precedence graph that was explained in the previous section. Here are some examples to test for conflict serializability of transactions.

Example:

T_1: Lock Shared A

T_2: Lock exclusive B

T_1: Unlock A

T_3: Lock exclusive A

T_2: Unlock B

T_1: Lock Shared B

T_3: Lock Shared B

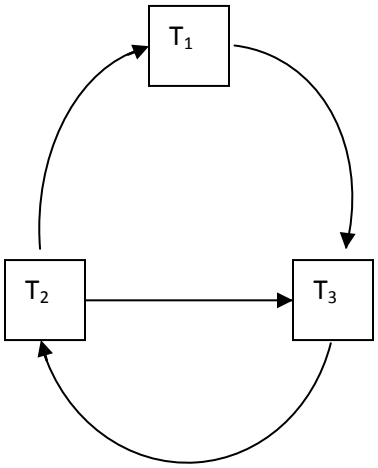
T_3: Unlock A

T_2: Lock exclusive A

T_2: Unlock A

T_1: Unlock B

T_3: Unlock B



There is a cycle in the graph, therefore if is not serializable.

Example: Is following schedule is serial

T_1: Lock exclusive B

T_2: Lock exclusive A

T_3: Lock exclusive C

T_1: Unlock B

T_2: Lock exclusive B

T_2: Unlock A

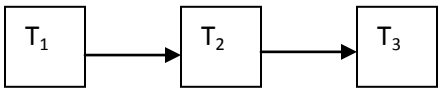
T_3: Lock exclusive A

T_2: Unlock B

T_3: Unlock C

T_3: Unlock A

The direct dependencies graphs follows



No cycle here, so it is serializable.

Example: Following schedule is serializable.

T_1: Lock shared A

T_2: Lock exclusive B

T_1: Unlock A

T_3: Lock exclusive A

T_2: Unlock B

T_1: Lock shared B

T_3: Lock shared B

T_3: Unlock A

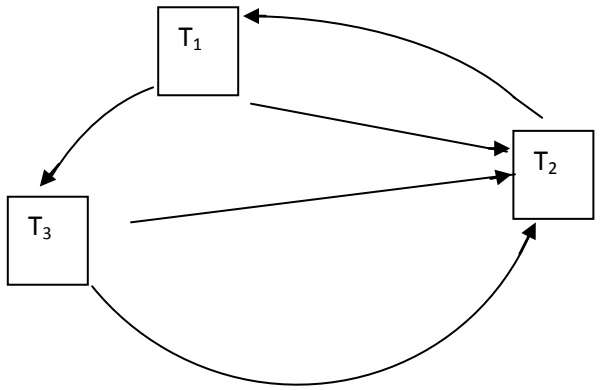
T_2: Lock exclusive A

T_2: Unlock A

T_1: Unlock B

T_3: Unlock B

The precedence graph as follows.



Here there is a cycle in this graph therefore it is non – serializable schedule.

2.8 LET US SUM UP

In this chapter we learnt the concept of transaction. The ACID properties and its importance for ensuring integrity were discussed. Then we studied about serial and serializable schedule in detail with examples. We also learnt about conflict and view serializability and the use of precedence graph. Finally we learnt to test for conflict serializability using precedence graph.

2.9 REFERENCES AND SUGGESTED READING

(1) Ramakrishnam, Gehrke, "Database Management Systems", McGraw- Hill.

(2) Elsmasri and Navathe, "Fundamentals of Database Systems", Pearson Education.

(3) Peter Rob and Coronel, "Database Systems, Design, Implementation and Management", Thomson Learning

.

2.10 EXERCISE

1 What do you mean by transactions in a database? Explain.

2. Explain ACID property. Why it is needed?

3. What is schedule? How many type of schedule? Explain with example.

4. What is conflict and view serializability? Explain with example.

5. Explain precedence graph and its significance.

3

CONCURRENCY CONTROL – LOCKS AND TIMESTAMPS

Unit Structure

3.0 OBJECTIVES

The objective of this chapter is

- To understand the concept of locks and the locking scheduler
- To understand two phase locking , its downgrading and upgrading
- To understand the concept of deadlocks
- To understand timestamps and its use in concurrency control
- To understand the Thomos write rule

3.1 INTRODUCTION

Concurrent execution of programs is essential for better performance of a database management system as it utilizes the system efficiently. But it should be ensured by the DBMS that no two transactions get into each other's way. If two transactions contend for a single data item and any of them changes it, then it is important that the data item should be consistent and should reflect the changes properly to all transactions. There are several mechanisms for concurrency control. Locking method and timestamps are one of the most used concurrency control protocols which will be discussed in this chapter.

3.2 CONCEPT OF LOCK

Locking is a commonly used technique in which data access is controlled to ensure serializability of transactions. The data item involved in transaction has a lock associated with it and when a transaction intends to access it, has to examine the associated lock first. If no other transaction holds the lock, the scheduler locks the data item for that transaction. A transaction may place a lock on the resource it requires in two modes:

- **Exclusive Lock**: If a transaction T1 has obtained on exclusive mode lock (X) on data (P), then T1 can both read and write (P). No other transaction can read or write (P) in this case. Following example illustrates the exclusive lock usage.

```
Lock-X (P);     (Exclusive Lock, P's value can be read and modified)
Read P;
P = P – 10;
Write P;
Unlock (P);     (Unlocking P after the modification is done)
Lock-X (P);     (Exclusive Lock, Q's value can be read and modified)
Read Q;
Q = Q + 10;
Write Q;
```

Unlock (Q); (Unlocking Q after the modification is done)

- **Shared Lock**: If a transaction T1 has obtained a shared mode lock (S) on data (P), then T1 cannot write P but can only read. Moreover another transaction T2 can also read (P). The following transaction shows the use of shared lock in transactions.

  Lock-S (P); (Shared Lock, P's value can be read only)
  Read P;
  R = P * 2;
  Unlock (P); (Unlocking P)
  Lock-X (Q); (Exclusive Lock, C's value can be read and modified)
  Read Q;
  Q = Q + R;
  Write Q;
  Unlock (Q); (Unlocking Q after the modification)

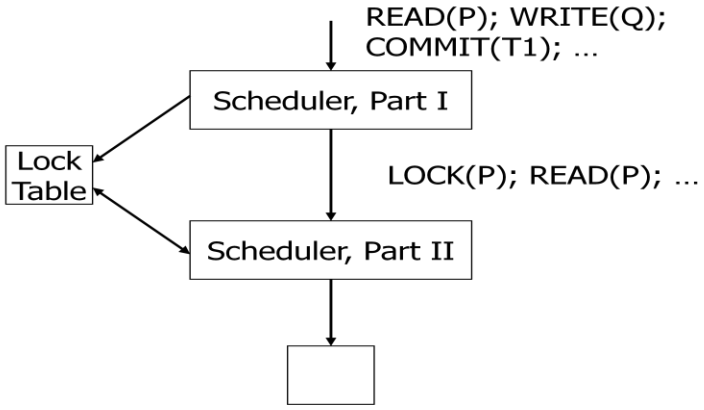
3.3 THE LOCKING SCHEDULER

The locking scheduler

- It is responsible for granting locks to the transactions.
- Keeps list of all current locks and requests for locks.
- Key idea behind locking scheduler
  - Each data item has a unique lock
  - Transactions must first acquire the lock before reading/writing the element.
  - If the lock is taken by another transaction then wait for it to be released.
  - The transactions must release the locks.
- A locking scheduler generally comprises of two parts
  - Part I: This part takes the stream of requests from the transactions and inserts lock actions ahead of all database-access operations.
  - Part II: It executes the sequence of actions passed to it by Part I. This part determines whether to delay the any transaction T because a lock has not been granted. If so, then add the action to the list of actions that must eventually be

performed for transaction T. If the action is a database access, it is transmitted to the database and executed and if the action is a lock request, examine the lock table to see if the lock can be granted.

The two parts are shown in the figure below



- Functioning of the scheduler
    - When a transaction T commits or aborts: Part I releases all locks held by T. If any transactions are waiting for any of these locks, Part I notifies Part II.
    - When Part II is notified that a lock on some database item P is available: It determines the next transaction or transactions that can now be given a lock on P. Those transactions are allowed to execute their delayed actions until they either complete or reach another lock request that cannot be granted.

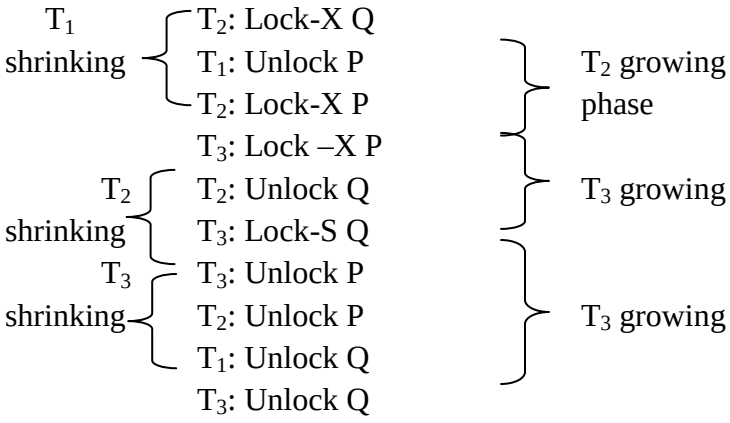
3.4 TWO PHASE LOCKING

This method ensures serializibility. It requires that each transaction issue lock and unlock request into two phases which is as follows.

- **Growing phase**: A transaction may obtain locks but may not release any locks.
- **Shrinking phase**: A transaction may release locks, but may not obtain anyhow locks.

The following example illustrates the growing and shrinking phase of two phase locking

T_1: Lock-S P
T_1: Lock-S Q
} T_1 is in growing phase



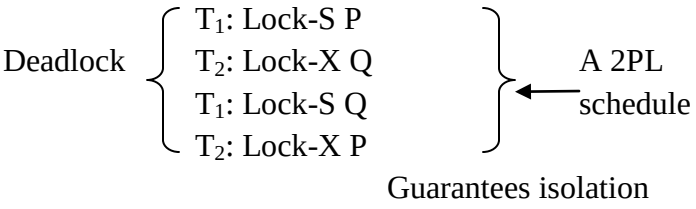
3.5 UPGRADING AND DOWNGRADING LOCKS

A different version of two phase locking protocol is that in which the locks can be converted. The two phases of this locking protocol is as follows:

- **First phase**:
  - A transaction can acquire lock-S on a data item.
  - A transaction can acquire lock-X on a data item.
  - Lock-S can be converted to lock-X. This process is known as upgrading.
- **Second phase**:
  - A transaction can release a lock-S.
  - A transaction can release a lock-X.
  - Lock-X can be converted into lock-S. This process is known as downgrading.

3.6 CONCEPT OF DEADLOCKS

Deadlock is operational problem. It does not violate the ACID property. For detecting deadlock firstly we draw direct dependency graph. If there is loop then there is deadlock otherwise not. The example as follows:

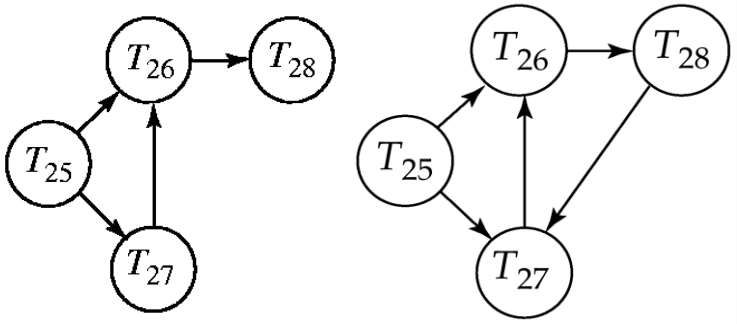


T_1: Unlock (P)
T_3: Lock-X P
T_2: Unlock P
T_3: Lock-S Q
T_3: Unlock P
T_2: Unlock Q
T_1: Unlock Q

- **Deadlocks: Necessary conditions**
  - (i) Mutual exclusion: Exclusive lock can be held by one process at time
  - (ii) Non-pre-emptive: Only greater of lock can be unlock it scheduling
  - (iii) Partial Allocation: Resources can be acquired piecemeal
  - (iv) Circular Waiting: Wait for each other to unlock resources need by other.

- **Deadlock Prevention**: Prevent the occurrence of any one necessary condition. To eliminate mutual exclusion is impossible.
  - (i) Non-pre-emptive scheduling: Create a super transaction that can pre-emptitively unlock.
  - (ii) Partial Allocation: Allocate the resources in one shot. Knowledge of resources needed must be available.
  - (iii) Circular Waiting: Impose ordering on resources. Acquisition is in this order only.
- **Deadlock detection**: Deadlocks can be described as a wait-for graph, which consists of a pair G = (V,E)
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair Ti $\rightarrow$ Tj.
  - If Ti $\rightarrow$ Tj is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.

- When Ti requests a data item currently being held by Tj, then the edge Ti Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle Wait-for graph with cycle

3.7 CONCURRENCY CONTROL BY TIMESTAMPS

In timestamps based concurrency control mechanism, each transaction is issued a timestamp when it enters the system. If an old transaction Ti has time-stamp TS (Ti), a new transaction Tj is assigned time-stamp TS (Tj) such that TS (Ti) < TS (Tj).

- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - W-timestamp (Q) is the largest time-stamp of any transaction that executed write (Q) successfully.
  - R-timestamp (Q) is the largest time-stamp of any transaction that executed read (Q) successfully.
- This protocol ensures that any conflicting read and write operations are executed in timestamp order.
- Suppose a transaction Ti issues a read (Q)
  - If TS (Ti) ≤ W-timestamp (Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.

- If TS (Ti) ≥ W-timestamp (Q), then the read operation is executed, and R-timestamp (Q) is set to the maximum of R-timestamp (Q) and TS (Ti).

- Suppose that transaction Ti issues write (Q).
  - If TS (Ti) < R-timestamp (Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and Ti is rolled back.
  - If TS (Ti) < W-timestamp (Q), then Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.
  - Otherwise, the write operation is executed, and W-timestamp (Q) is set to TS (Ti).

Following is an example of a partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read(X)
read(Y)	read(Y)			
		write(Y)		
		write(Z)		
				read(Z)
	read(X)			
read(X)	abort			
		write(Z)		
		abort		
				write(Y)
				write(Z)

3.8 THOMOS WRITE RULE

Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.

- If TS (Ti) < R-timestamp (Q) then the system reject the write operation and Ti is rolled back. Because at any transaction the first operation must be read operation.

- If TS (Ti) < W-timestamp (Q) then the operation rejected and Ti is rolled back. Because this write operation attempting to produce obsolete value.
- Otherwise this protocol is the same as the timestamp ordering protocol.

3.9 REFERENCES AND SUGGESTED READING

(1) R. Elmasri and S. B. Navathe, Fundamentals of Data Base Systems, 4th Edition, Pearson Education Asia, New Delhi, 2004.

(2) C. J. Date "An introduction to database systems (7th ed.)" – Pearson Education Asia, New Delhi,.

(3) Jeffery D Ulman, Jennifer Widom, "A first Course in Database Systems", Pearson Education.

(4) A Silberschtz, H. F. Korth, Sr Sudarshan, "Database System Completes," Fifth Edition, Mc Graw Hill,2005.

(5) P Benstein etal, "Concurrency Control and Recovery in Database Systems," Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1999.

3.10 LET US SUM UP

We learnt about the concept of locks and its growing and shrinking phase. The locking scheduler and its function were then discussed. Then we studied two phase locking for concurrency control. Further we learnt about upgrading and downgrading locks. Concept of deadlocks, with its causes and prevention was also discussed. Concurrency control by timestamps with an example was elaborated. At last, we learnt about Thomos write rule.

3.11 EXERCISE

1. Explain the locking scheme.

2. Explain the two phase locking with example.

3. Explain the upgrading and downgrading of locks.

4. Explain the concurrency control with timestamps.

5. What is Thomas Write Rule?

6. Explain the necessary and sufficient condition for deadlock.

7. Explain the concept of deadlock detection

4

CRASH RECOVERY METHODS

Unit Structure

4.0 OBJECTIVES

The objective of this chapter is

- To understand ARIES algorithm
- To understand log based recovery technique
- To understand dirty page table
- To understand write ahead protocol
- To understand checkpoints
- To understand system crash recovery

4.1 INTRODUCTION

The recovery system consists of recovery data which is the update history of transactions. Recovery utility is software that must run for recovering when database is in bad state, operation or database disallowed.

Basic Concepts in Recovering the System:

- **Recovery**: It is a process to restore database to a consistent state after it has met with a failure.
- **Failure**: It is a database inconsistency that is visible.
- **Transaction Recovery**: It is a process to restore that last consistent state of data items modified by failed transactions.
- **Transaction Log**: It maintains execution history of concurrent transactions in the form of following record;
  (transaction_id, operation, date item, before image, after image)
- **BFIM and AFIM**: The value of a database object before its update is called as before image (BFIM) and the value of that object after its update is called as after image (AFIM).
- **Transaction directories**: During the execution of a transaction two directories are maintained:
  - **Current directory:** The entries in this directory points to the most recent database pages on disk.
  - **Shadow directory:** It points to the old entries. It gets its entry by the current directory.
- **Recovery log entries**: There are two types of recovery log entries
  - Undo type log entry: It includes the BFIM of a data being updated. It is required for undo operation.
  - Redo type log entry: It includes the AFIM of a data item being updated. It is needed for the redo operation.
- **Recovery approaches: Steal/no – steal approach**

  - Steal: In this case updated pages are allowed to be written to disk before the transaction commits. It is a form of immediate update.
  - No steal: Updated pages cannot be written to the disk before the transaction commits. It is a kind of deferred update.
- **Recovery approaches: Force/no force approach**
  - Force: In this case transaction writes immediately all updated pages to the disk when the transaction commits.
  - No force: Pages updated by the transaction are not written immediately to the disk.
- **Recovery management**: Recovery management has two components
  - **Recovery manager**: It keeps track of transactions, handles commit and abort operations. It also takes care of system checkpoint and restart.
  - **Log manager**: It provides log services to the recovery manager and other components that may need its service.

Causes of Transaction Failure: The causes are as follows.

- Logical Errors: These are defined as fatal errors in transaction
- DBMS Error: It is due to deadlock detection and rollback system enters in bad state.
- System Crash: Power out, OS failure, H/W malfunction.
- I/O: It is like a disk failure.

4.2 ARIES ALGORITHM

Algorithms for Recovery and Isolation Exploiting Semantics, or ARIES is a recovery algorithm designed to work with a no-force, steal database approach; it is used by IBM DB2, Microsoft SQL Server and many other database systems.

Three main principles lie behind ARIES

- **Write ahead logging**: Any change to an object is first recorded in the log, and the log must be written to stable storage before changes to the object are written to disk.

- **Repeating history during Redo**: On restart after a crash, ARIES retraces the actions of a database before the crash and brings the system back to the exact state that it was in before the crash. Then it undoes the transactions still active at crash time.
- **Logging changes during Undo**: Changes made to the database while undoing transactions are logged to ensure such an action isn't repeated in the event of repeated restarts.

ARIES perform three steps after crash

- **Analysis**: Finds all pages that have not been written to disk (dirty pages) and all active transactions at the time of crash.
- **Redo**: Repeats all the statements in the log (at an appropriate point) and restore the database to a state similar to before crash has occurred.
- **Undo**: Undoes the operations of transactions those did not commit.

Information sources of ARIES recovery

- **Log record**: Each log record has a log sequence number (LSN) which is monotonically increasing. It indicates the address of the log record on the disk. There are different logging actions like write, commit, abort, undo and ending a transaction which are recorded in log record.
- **Transaction table**: It contains an entry for each active transaction. In recovery process it is rebuild.
- **Dirty page table**: It contains an entry for each dirty page in the buffer. It also includes the page ID and the LSN corresponding to the earliest update to that page.

ARIES **Compensation Log Record (CLR)**

- This record is written just before the change recorded in update log is undone.
- It describes the action taken to undo the actions recorded in the corresponding update record.
- It contains field undoNextLSN, the LSN of the next log record that is to be undone for the transaction that wrote the update record.
- It describes an action that will never be undone.
- CLR contains information needed to reapply or redo, but not to reverse it.

4.3 LOG BASED RECOVERY

Log file is a sequential file that contains a record of actions taken by an entity. A log is kept on stable storage. There are two log records used by log based recovery technique:

- Undo log records: It contains log entries of all write operations before update.
- Redo log records: It contains log entries of all write operations after update.

The algorithm for log based recovery is as follows:

- When transaction Ti starts, it registers itself by writing a <Ti start> log record
- Before Ti executes write (X), a log record < Ti, X, V1, V2> is written, where V1 is the value of X before the write, and V2 is the value to be written to X.
- When Ti finishes its last statement, the log record <Ti commit> is written
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- The BFIM is not overwritten by AFIM until all undo log records for the updating information is force written to the disk.
- The commit operation of a transaction cannot be completed until all the redo and undo log are force written to the disk.
- Two approaches using logs
    - Deferred database modification: This scheme records all modifications to the log, but defers all the writes to after partial commit.
    - Immediate database modifications: This scheme allows database updates of an uncommitted transaction to be made as the writes are issued.

4.4 TRANSACTION AND DIRTY PAGE TABLE

Dirty page table is used to represent information about dirty buffer pages during normal processing. It is also used during restart recovery. It is implemented using hashing or via the deferred- writes queue mechanism. Each entry in the table consists of two fields:

- PageID and
- RecLSN

During normal processing , when a non-dirty page is being fixed in the buffers with the intention to modify , the buffer manager records in the buffer pool (BP) dirty-pages table , as RecLSN , the current end-of-log LSN , which will be the LSN of the next log record to be written. The value of RecLSN indicates from what point in the log there may be updates. Whenever pages are written back to nonvolatile storage, the corresponding entries in the BP dirty-page table are removed. The contents of this table are included in the checkpoint record that is written during normal processing. The restart dirty-pages table is initialized from the latest checkpoint's record and is modified during the analysis of the other records during the analysis pass. The minimum RecLSN value in the table gives the starting point for the redo pass during restart recovery.

4.5 WRITE AHEAD LOG PROTOCOL

Write-ahead logging (WAL) is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems.

The Write-Ahead Logging Protocol:

- Must force the log record for an update before the corresponding data page gets to disk.
- Must write all log records for a exact before commit.
- Guarantees Atomicity.
- Guarantees Durability.

WAL allows updates of a database to be done in-place. Another way to implement atomic updates is with shadow paging, which is not in-place. The main advantage of doing updates in-place is that it reduces the need to modify indexes and block lists.

ARIES is a popular algorithm in the WAL family.

4.6 CHECKPOINTS

Checkpoint mechanism copies the state of a process into nonvolatile storage. Restore mechanism copies the last known checkpointed state of the process back into memory and continues processing. This mechanism is especially useful for application which may run for long periods of time before reaching a solution.

Checkpoint-Recovery gives an application or system the ability to save its state, and tolerate failures by enabling a failed executive to recover to an earlier safe state.

Key ideas

- Saves executive state
- Provides recovery mechanism in the presence of a fault
- Can allow tolerance of any non-apocalyptic failure
- Provides mechanism for process migration in distributed systems for fault tolerance reasons or load balancing

During the execution of the transaction, periodically perform checkpointing. This includes

- Output the log buffers to the log.
- Force – write the database buffers to the disk.
- Output an entry < checkpoint > on the log.

During the recovery process, the following two steps are performed

- Undo all the transactions that have not committed.
- Redo all transactions that have committed after the checkpoint.

Demerits of technique:

- Insufficient in context of large databases
- It requires transactions to execute serially

4.7 RECOVERY FROM A SYSTEM CRASH

Sometimes when there is power failure or some hardware or software failure occurs, it causes the system to crash. The following actions are taken when recovering from system crash

Scan log forward from last <checkpoint> record

- Repeat history by physically redoing all updates of all transactions.
- Create an undo-list during scan as follows
    - Undo-list is set to L initially
    - Whenever <Ti start> is found Ti is added to undo-list
    - Whenever <Ti commit> or <Ti abort> is found, Ti is deleted from undo-list

This brings database to state as of crash, with committed as well as uncommitted transactions having been redone. Now undo-list contains transactions that are incomplete, that is, have neither committed nor been fully rolled back.

Scan log backwards, performing undo on log records of transactions found in undo-list.

- Transactions are rolled back as described earlier.
- When <Ti start> is found for a transaction Ti in undo-list, write a <Ti abort> log record.
- Stop scan when <Ti start> records have been found for all Ti in undo-list

This undoes the effects of incomplete transactions (those with neither commit nor abort log records). Recovery is now complete.

4.8 REDO AND UNDO PHASES

The recovery algorithms like ARIES have two phases: Undo and Redo phases

(i) Undo (Ti): Restore the value of all data items updated by Ti to old values. The log is scanned backwards and the operations of transactions that were active at the time of the crash are undone in reverse order.

(ii) Redo (Ti): Sets the value of data items updated by transaction Ti to new values. It actually reapplies updates from the log to the database. Generally the Redo operation is applied to only committed transactions. If a transaction was aborted before the crash and its updates were undone, as indicated by CLRs, the actions described in CLRs are also reapplied.

4.9 REFERENCES AND SUGGESTED READING

(1) R.Elmasri and S.B.Navathe, Fundamentals of Data Base Systems, 4th Edition, Pearson Education Asia, New Delhi, 2004.

(2) C. J. Date "An introduction to database systems (7th ed.)" – Pearson Education Asia, New Delhi,.

(3) Jeffery D Ulman, Jennifer Widom, "A first Course in Database Systems", Pearson Education.

(4) A Silberschtz, H. F. Korth, Sr Sudarshan, "Database System Completes," Fifth Edition, Mc Graw Hill,2005.

(5) P Benstein, Etal, "Concurrency Control and Recovery in Database Systems," Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1999.

4.10 LET US SUM UP

We learnt about the recovery system and causes of failures. The ARIES algorithm was then discussed. Then we studied log based recovery algorithm for recovery process. Further we learnt about dirty page table. Write – ahead protocol for recovery process was also discussed. Check points and its use in recovery process were elaborated. Recovery from system crash was also learnt in the later part of the chapter. At last, we learnt about redo and undo phases of recovery process.

4.11 EXERCISE

1. Explain the ARIES algorithm

2. Explain the log based recovery

3. Explain the concept of dirty page table

4. Explain the concept of shadow paging

5. Explain the concept of check points

6. Explain the redo and undo

7. Explain redo-undo algorithm

8. What is recovery? Why it is needed?

9. What types of recovery schemes?

10. Explain write ahead-log-protocol.

5

FUNDAMENTALS OF PL/SQL

Unit Structure

5.0 OBJECTIVES

The objective of this chapter is

- To understand variables and constants in PL/SQL
- To understand PL/SQL expressions and comparisons

- To understand PL/SQL datatypes

5.1 INTRODUCTION

SQL is the natural language of the DBA, but it suffers from various inherent disadvantages, when used as a conventional programming language. SQL does not have any procedural capabilities i.e. SQL does not provide the programming techniques or condition checking, looping and branching that is vital for data testing before its permanent storage. SQL statement is executed when a call is made to the engines resource. This adds to the traffic on the network, thereby decreasing the speed of data processing, especially in multi-user environment. While processing an SQL sentence if an error occurs, the Oracle engine displays its own error messages. SQL has no facility for programmed handling of errors that arise during the manipulation of data.

Oracle provides PL/SQL. As the name suggests, PL/SQL is SQL. PL/SQL is a block-structured language that enables developers to combine the power to SQL with procedural statements. PL/SQL bridges the gap between database technology and procedural programming languages.

Advantages of PL/SQL: PL/SQL is development tool that only supports SQL data manipulation but also provides facilities of conditional checking, branching and looping. PL/SQL sends an entire block of SQL statements to the Oracle engine all in one go. Communication between the program block and the Oracle engine reduces considerably, reducing network traffic. Since the Oracle engine got the SQL statements as a single block, it processes this code much faster than if it got the code one sentence of a time. There is a define improvement in the performance time of the Oracle engine. As an entire block of SQL code is passed to the Oracle engine at one time for execution, all changes made to the data in the table are done or undone, in one go. PL/SQL also permits dealing with errors as required, and facilitates displaying user-friendly messages, when errors are encountered. PL/SQL allows declaration and use of variables in blocks of code. These variables can be used to store intermediate result of a query for later processing or calculate values and insert them in to an Oracle table later. PL/SQL variable can be used anywhere either in SQL statements or in PL/SQL block via PL/SQL block engine. This considerably improves transaction performance applications system, where Oracle is operational. Hence, PL/SQL code

blocks written for a DOS version of Oracle will run on its Linux/Unix version, without any modifications at all

The Generic PL/SQL Block: Every programming environment allows the creation of structured, logical blocks of code that describe processes, which have to be applied to data. Once these blocks are passed to the environment, the processes described are applied to data, suitable data manipulation takes place and useful output is obtained. PL/SQL permits the creation at structured logical blocks of code that describe processes, which have to be applied to data. A single PL/SQL code block consist a set of SQL statements, clubbed together, and passed to the Oracle engine entirely. The sections of a PL/SQL block are:

- **The Declare Section**: This code block starts with a declaration section in which memory variables and other Oracle objectives can be declared, and if required initialized. Once declared, they can be used in SQL statements for data manipulation.
- **The Begin section**: It consists of a set of SQL and PL/SQL statements, which describe processes that have to be applied to table data. Actual data manipulation, retrieval, looping and branching constructs are specified in this section.

The syntax of declare and begin section are given below:

```
declare
  variable declarations
begin
  sql statements
end;
```

Example:

```
DECLARE
        var_salary number(6);
        var_emp_id number(6) = 1116;
BEGIN
        SELECT salary
        INTO var_salary
        FROM employee
        WHERE emp_id = var_emp_id;
```

```
        dbms_output.put_line(var_salary);
        dbms_output.put_line('The employee '
            || var_emp_id || ' has  salary  ' || var_salary);
    END;
```

The above program will get the salary of an employee with id '1116' and display it on the screen.

5.2 DEFINING VARIABLES AND CONSTANTS

These are placeholders that store the values that can change through the PL/SQL Block. The general syntax to declare a variable is:

variable_name datatype [NOT NULL := value];

- variable_name is the name of the variable.
- datatype is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable.
- value or DEFAULT valueis also an optional specification, where we can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if we want to store the current salary of an employee, we can use a variable.

DECLARE

salary number (6);

* “salary” is a variable of datatype number and of length 6.

PL/SQL Constants: As the name implies a constant is a value used in a PL/SQL block that remains unchanged throughout the program. A constant is a user-defined literal value. We can declare a constant and use it instead of actual value.

For example if we want to write a program which will increase the salary of the employees by 25%, we can declare a constant and use it throughout the program. Next time when we want to

increase the salary again we can change the value of the constant which will be easier than changing the actual value throughout the program.

The general syntax to declare a constant is:

constant_name CONSTANT datatype := VALUE;

- constant_name is the name of the constant i.e. similar to a variable name.
- The word CONSTANT is a reserved word and ensures that the value does not change.
- VALUE - It is a value which must be assigned to a constant when it is declared. We cannot assign a value later.

For example, to declare salary_increase, we can write code as follows:

```
DECLARE
salary_increase CONSTANT number (3) := 10;
```

We must assign a value to a constant at the time we declare it. If we do not assign a value to a constant while declaring it and try to assign a value in the execution section, we will get an error.

5.3 PL/SQL EXPRESSIONS AND COMPARISONS

Expressions are constructed using operands and operators. An operand is a variable, constant, literal, or function call that contributes a value to an expression. An example of a simple arithmetic expression follows:

```
-X / 2 + 3
```

Unary operators such as the negation operator (-) operate on one operand; binary operators such as the division operator (/) operate on two operands. PL/SQL has no ternary operators.

The simplest expressions consist of a single variable, which yields a value directly. PL/SQL evaluates an expression by combining the values of the operands in ways specified by the

operators. An expression always returns a single value. PL/SQL determines the datatype of this value by examining the expression and the context in which it appears.

5.3.1 Logical operators

The logical operators AND, OR, and NOT follow the tri-state logic shown in the table below. AND and OR are binary operators; NOT is a unary operator.

X	**Y**	**x AND y**	**x OR y**	**NOT x**
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL	NULL

As the truth table shows, AND returns TRUE only if both its operands are true. On the other hand, OR returns TRUE if either of its operands is true. NOT returns FALSE for TRUE and TRUE for FALSE value.

5.3.2 Boolean operations: In a SQL statement, BOOLEAN expressions let us specify the rows in a table that are affected by some statements. In a procedural statement, BOOLEAN expressions are the basis for conditional control. There are three kinds of BOOLEAN expressions: arithmetic, character, and date.

- **BOOLEAN Arithmetic Expressions:** We can use the relational operators to compare numbers for equality or inequality. Comparisons are quantitative; that is, one number is greater than another if it represents a larger quantity. For example, given the assignments

number1 := 75;

number2 := 70;

The following expression is true:

number1 > number2

- **BOOLEAN Character Expressions**: We can compare character values for equality or inequality. By default, comparisons are based on the binary values of each byte in the string. For example, given the assignments

  string1 := 'Kathy';

  string2 := 'Kathleen';

  The following expression is true:

  string1 > string2

- **BOOLEAN Date Expressions**: We can also compare dates. Comparisons are chronological; that is, one date is greater than another if it is more recent. For example, given the assignments

  date1 := '01-JAN-91';

  date2 := '31-DEC-90';

  The following expression is true:

  date1 > date2

5.3.3 CASE Expression: There are two types of expressions used in CASE statements: simple and searched. These expressions correspond to the type of CASE statement in which they are used.

- **Simple CASE expression**: A simple CASE expression selects a result from one or more alternatives, and returns the result. Although it contains a block that might stretch over several lines, it really is an expression that forms part of a larger statement, such as an

assignment or a procedure call. The CASE expression uses a selector, an expression whose value determines which alternative to return.

Example:

```
DECLARE
        deptno     NUMBER := 20;
        dept_desc  VARCHAR2(20);
BEGIN
        dept_desc := CASE deptno
                WHEN 10 THEN 'Accounting'
                WHEN 20 THEN 'Research'
                WHEN 30 THEN 'Sales'
                WHEN 40 THEN 'Operations'
                ELSE 'Unknown'
               END;
        DBMS_OUTPUT.PUT_LINE(dept_desc);
END;
```

- **Searched CASE Expression**: A searched CASE expression lets us test different conditions instead of comparing a single expression to various values. A searched CASE expression has no selector. Each WHEN clause contains a search condition that yields a BOOLEAN value, so we can test different variables or multiple conditions in a single WHEN clause.

```
DECLARE
        sal NUMBER := 2000;
        sal_desc  VARCHAR2(20);
BEGIN
 sal_desc := CASE
          WHEN sal < 1000 THEN 'Low'
          WHEN sal BETWEEN 1000 AND 3000 THEN 'Medium'
          WHEN sal > 3000 THEN 'High'
          ELSE 'N/A'
```

```
        END;
    DBMS_OUTPUT.PUT_LINE (sal_desc);
END;
```

5.3.4 NULL Value in Comparisons: When working with nulls, we can avoid some common mistakes by keeping in mind the following rules:

- Comparisons involving nulls always yield NULL
- Applying the logical operator NOT to a null yields NULL
- In conditional control statements, if the condition yields NULL, its associated sequence of statements is not executed
- If the expression in a simple CASE statement or CASE expression yields NULL, it cannot be matched by using WHEN NULL. In this case, we would need to use the searched case syntax and test WHEN expression IS NULL.

Example:

```
IF rating > 90 THEN
        compute_bonus(emp_id);
ELSE
        NULL;
END IF;
```

5.4 PL/SQL DATATYPES

5.4.1 Number type: The NUMBER datatype is used to store fixed-point or floating-point numbers. Its magnitude range is 1E-130 .. 10E125. If the value of an expression falls outside this range, we get a numeric overflow or underflow error. We can specify precision, which is the total number of digits, and scale, which is the number of digits to the right of the decimal point. The syntax follows:

```
NUMBER[(precision,scale)]
```

To declare fixed-point numbers, for which we must specify scale:

```
NUMBER(precision,scale)
```

5.4.2 Character type: Character types let us store alphanumeric data, represent words and text, and manipulate character strings.

- **CHAR**: The CHAR datatype is used to store fixed-length character data. How the data is represented internally depends on the database character set. The CHAR datatype takes an optional parameter that lets us specify a maximum size up to 32767 bytes. We can specify the size in terms of bytes or characters, where each character contains one or more bytes, depending on the character set encoding. The syntax follows:
  CHAR [(maximum_size [CHAR | BYTE])]

- **VARCHAR2**: The VARCHAR2 datatype is used to store variable-length character data. The VARCHAR2 datatype takes a required parameter that specifies a maximum size up to 32767 bytes. The syntax follows:
  VARCHAR2(maximum_size [CHAR | BYTE])
  We cannot use a symbolic constant or variable to specify the maximum size; we must use an integer literal in the range 1 .. 32767.

5.4.3 Boolean type: BOOLEAN datatype is used to store the logical values TRUE, FALSE, and NULL (which stand for a missing, unknown, or inapplicable value). Only logical operations are allowed on BOOLEAN variables.
The BOOLEAN datatype takes no parameters. Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable. We cannot insert the values TRUE and FALSE into a database column. Also, we cannot select or fetch column values into a BOOLEAN variable.
5.4.4 Datetime type: The Datetime datatypes lets us store and manipulate dates, times, and intervals (periods of time). A variable that has a date/time datatype holds values called datetimes; a variable that has an interval datatype holds values called intervals. A datetime or interval consists of fields, which determine its value.
We use the DATE datatype to store fixed-length datetimes, which include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the

time portion defaults to midnight. The date function SYSDATE returns the current date and time.

5.5 LET US SUM UP

In this chapter first, we defined how variables and constants used in PL/SQL. Then we saw PL/SQL expressions and comparisons which include logical operations, Boolean expressions, CASE expressions, NULL values in Comparison and Conditional Statements. Then we learnt about PL/SQL datatypes and its types which include Number types, Character types, Boolean type, Datetime and Interval types.

5.6 REFERENCES AND SUGGESTED READING

(1) Ramakrishnam, Gehrke, "Database Management Systems", McGraw- Hill.

(2) Ivan Bayross, "SQL,PL/SQL -The Programming language of Oracle", B.P.B. Publications, 3rd Revised Edition

(3) Michael Abbey, Michael J. Corey, Ian Abramson, Oracle 8i – A Beginner's Guide, Tata McGraw-Hill.

5.7 EXERCISE

1. Explain variables and constants in PL/SQL.
2. Describe PL/SQL Expressions and Comparisons
3. Explain the use of NULL Values in Comparison and Conditional Statements.
4. Differentiate between simple CASE and searched CASE expression.
5. Describe various datatypes in PL/SQL.

6

CONTROL STRUCTURES IN PL/SQL

Unit Structure

6.0 OBJECTIVES

The objective of this chapter is

- To understand conditional control statements

- To understand iterative control statements
- To understand sequential control statements

6.1 INTRODUCTION

Control structures are meant to control the flow of program in a programming language. There are three types of control structures:

- Conditional control
- Iterative control
- Sequential control

We will discuss all of the above mentioned control structures in detail with reference to PL/SQL in this chapter.

6.2 CONDITIONAL CONTROL

'IF' statement in the PL/SQL can be used to control the execution of a block of code. There are three three formats of IF statement:

6.2.1 IF – THEN Statement: The syntax for IF – THEN statement is

```
IF< condition > THEN
        < commands >
END IF;
```

Example: The following PL/SQL block of code illustrates the use of IF – THEN statement

```
CREATE OR REPLACE
PROCEDURE insY (pp IN NUMBER)
AS
qq NUMBER := 10;
BEGIN
```

```
        IF pp > qq THEN
                INSERT INTO Y VALUES (pp, 1);
        END IF;
END;
```

6.2.2 IF – THEN – ELSE Statement: The syntax for IF – THEN – ELSE statement is

```
IF< condition > THEN
        < commands >
ELSE
        < commands >
END IF;
```

Example: The PL/SQL block of code below illustrates the use of IF – THEN – ELSE statement

```
CREATE OR REPLACE
PROCEDURE insY (pp IN NUMBER)
AS
qq NUMBER := 10;
BEGIN
        IF pp > qq THEN
                INSERT INTO Y VALUES (pp, 1);
        ELSE
                INSERT INTO Y VALUES (qq, 2);
        END IF;
END;
```

6.2.3 IF – THEN – ELSIF Statement: The syntax for IF – THEN – ELSIF statement is

```
IF < condition > THEN
```

```
        < commands >
ELSIF < condition > THEN
        < commands >
ELSE
        < commands >
END IF;
```

Example: The following PL/SQL block of code illustrates the use of IF – THEN – ELSIF statement

```
CREATE OR REPLACE
PROCEDURE insY (pp IN NUMBER)
AS
qq NUMBER := 10;
BEGIN
        IF pp > qq THEN
                INSERT INTO Y VALUES (pp, 1);
        ELSIF pp = qq THEN
                INSERT INTO Y VALUES (pp, 2);
        ELSE
                INSERT INTO Y VALUES (qq, 3);
        END IF;
END;
```

6.2.4 CASE statement: The PL/SQL CASE statement allows you to execute a sequence of statements based on a selector variable. The syntax for CASE statement is

```
CASE < selector variable >
        WHEN < expression 1 > THEN
        < commands >
```

```
        WHEN < expression 1 > THEN
        < commands >
        .
        .
        .
        [ELSE
        < commands >]
    END CASE
```

Example:

```
BEGIN
    CASE pp
    WHEN 1 THEN
        INSERT INTO Y VALUES (pp, 1);
    WHEN 2 THEN
        INSERT INTO Y VALUES (pp, 2);
    WHEN 3 THEN
        INSERT INTO Y VALUES (pp, 3);
    ELSE
        INSERT INTO Y VALUES (qq, 0);
    END CASE;
END;
```

6.3 ITERATIVE CONTROL

Iterative control statements are meant for a sequence of statements that has to be repeated. In PL/SQL there are three ways for iterative statements :

6.3.1 LOOP and EXIT: The syntax for LOOP and EXIT statement is as follows:

```
LOOP
        < statements >
        [EXIT WHEN < condition >;]
END LOOP;
```

Example: The following example shows the use of LOOP and EXIT commands

```
DECLARE
pp NUMBER := 0;
BEGIN
        LOOP
                INSERT INTO y VALUES (pp, 1);
                pp := pp + 1;
                EXIT WHEN pp = 6;
        END LOOP;
END;
```

6.3.2 WHILE LOOP: The syntax for WHILE LOOP is as follows:

```
WHILE < condition >
LOOP
        < commands >
END LOOP;
```

Example: The following example illustrates the use of WHILE LOOP

```
DECLARE
pp NUMBER := 0;
BEGIN
        WHILE pp < 6
```

```
        LOOP
            INSERT INTO y VALUES (pp, 1);
            pp := pp + 1;
        END LOOP;
    END;
```

6.3.3 FOR LOOP: The syntax for FOR LOOP is as follows:

```
    FOR variable_name IN [REVERSE] start..end
    LOOP
        < commands >
    END LOOP;
```

Example: The following example illustrates the use of FOR LOOP

```
    DECLARE
    pp NUMBER;
    BEGIN
        FOR pp IN 0..4
        LOOP
            INSERT INTO y VALUES (pp, 1);
        END LOOP;
    END;
```

6.4 SEQUENTIAL CONTROL

6.4.1 GOTO and NULL Statements: These statements change the flow of control within a PL/SQL block of code. GOTO and NULL statements are used for this purpose. The syntax of GOTO statement is

GOTO < codeblock name >

Example:

```
CREATE OR REPLACE
PROCEDURE insY (pp IN NUMBER)
AS
qq NUMBER := 10;
BEGIN
    IF pp > qq THEN
        GOTO label1;
    ELSE
        INSERT INTO Y VALUES (qq, 2);
    END IF;
END;
<<label1>>
        INSERT INTO Y VALUES (pp, 1);
```

NULL Statement: The NULL statement is used in PL/SQL block when we have to do nothing. The NULL statement can act as a placeholder whenever an executable statement is requisite, but no operation is required; for example, within a branch of the IF-THEN-ELSE statement.

Syntax

```
NULL;
```

Example:The following PL/SQL block illustrates the use of NULL statement.

```
CREATE OR REPLACE
PROCEDURE insY (pp IN NUMBER)
AS
```

```
qq NUMBER := 10;
BEGIN
        IF pp > qq THEN
                NULL;
        ELSE
                INSERT INTO Y VALUES (qq, 2);
        END IF;
END;
```

6.5 CONCEPT OF NESTED TABLE

A nested table is an unordered set of any number of elements, all of the same data type. It has a single column and the type of that column may be a built-in database type or an object type. If the column in a nested table is an object type, the table can also be viewed as a multicolumn table, with a column for each attribute of the object type. We can insert, update, and delete individual elements in a nested table.

Syntax for nested table :

TYPE type_name IS TABLE OF element_type [NOT NULL];

where type_name is a type specifier used later to declare collections. For nested tables declared within PL/SQL, element_type is any PL/SQL datatype except:

REF CURSOR

Example:

```
CREATE TYPE CourseList AS TABLE OF VARCHAR2(10)  -- define type
/
CREATE TYPE Student AS OBJECT (  -- create object
  id_num  INTEGER(4),
  name    VARCHAR2(25),
```

```
  address VARCHAR2(35),
  status  CHAR(2),
  courses CourseList)  -- declare nested table as attribute
/
```

The above script shows how you might declare a nested table in SQL, and use it as an attribute of an object type.

6.6 LET US SUM UP

We learnt about the three types of control structures in this chapter. The conditional control structure which involve: IF – THEN Statement, IF – THEN – ELSE Statement, IF – THEN – ELSIF Statement, CASE Statement. Then we learnt about the iterative control structure which entail LOOP and EXIT Statement, WHILE LOOP and FOR LOOP structure. Finally the Sequential Control using GOTO and NULL statements.

6.7 REFERENCES AND SUGGESTED READING

(1) Ramakrishnam Gehrke, "Database Management Systems", McGraw- Hill.

(2) Ivan Bayross, "SQL, PL/SQL - The Programming language of Oracle", B.P.B. Publications, 3rd Revised Edition

(3) Michael Abbey, Michael J. Corey, Ian Abramson, Oracle 8i – A Beginner's Guide, Tata McGraw-Hill.

6.8 EXERCISE

1. Explain 'IF'conditional control statement and its various forms.
2. Describe use of CASE expression with example.
3. What do you mean by iterative control?
4. Describe sequential control in detail.
5. Explain Nested table with an example.

7

QUERY EVALUATION AND CURSORS IN PL/SQL

Unit Structure

7.0 OBJECTIVES

The objective of this chapter is

- To understand the query evaluation and its techniques
- To understand cursors and its types

7.1 INTRODUCTION

Query optimization is one of the most important tasks of a relational DBMS. The optimizer generates alternative plans and chooses the plan with the least estimated cost. Query optimization uses the concept of query evaluation which will be discussed in this chapter.

In the second part of the chapter we will learn about cursors. Cursors form an important part of PL/SQL and are used to handle various kinds of operations in the database.

7.2 QUERY EVALUATION

There are alternative way of evaluating a given query which is done through evaluating the expressions and defining algorithms for each operation. Query evaluation can be achieved with a query evaluation plan. Query evaluation plan can be represented as trees of relational operators, with labels identifying the algorithm to use at each node. The process of finding a good evaluation plan is called query optimization. Query Optimization is basically the process of choosing the most efficient way to execute a SQL statement.

7.2.1 System Catalog: A database system should have a meta-database of information on the schemata which it contains. It includes, for each schema, at least the following:

- Relation names in the schema.
- Column names of each relation.
- Data type of each column.
- The integrity constraints on the relations.
- Information about indices on the relations.
- The privileges for the elements of the schema.

This database is commonly called as the system catalog. It can be defined as the collection of files corresponding to user's tables and indexes representing data in the database. It helps in finding the best way to evaluate a query.

7.2.2 Evaluation of relational operators: Algorithms for evaluating relational operators use simple ideas extensively:

- **Indexing**: It can use WHERE conditions to retrieve small set of tuples (selections, joins)
- **Iteration**: Sometimes, it is faster to scan all tuples even if there is an index. (sometimes data entries can be scanned in an index instead of the table itself.)
- **Partitioning**: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

Transformation of Relational Expression:

Two relational algebra expressions are said to be equivalent if on every legal database instance the two expressions generate the same set of the tuples. The order of tuples is irrelevant.

Equivalent Rules: The equivalent rules are as follows:

- Conjunctive selection operation can be deconstructed into a sequence of individual selections.

$$\sigma_{Q1,Q2}(E) = \sigma_{Q1}(\sigma_{Q2}(E))$$

- Selection operations are cumulative.

$$\sigma_{Q1,Q2}(E) = \sigma_{Q2}(\sigma_{Q1}(E))$$

- Only last in sequence of projection operation is needed the others can be omitted.

$$\pi_{2_1}(\pi_{2_2}(....\pi_{L_n}(E))...) = \pi_{2_1}(E)$$

- Selection can be combined with Cartesian products and theta joins.

$$\sigma_Q = (E_1 \times E_2) = E_1 \bowtie_Q E_2$$

$$\sigma_{Q1}(E_1 \bowtie_{Q2} E_2) = E_1 \bowtie_{Q1 \wedge Q2} E_2$$

- Theta – join operations (and natural join) are cumulative

$$E_1 \bowtie_Q E_2 = E_2 \bowtie_Q E_1$$

- Natural join operations are associative

$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$

- Theta join associative in following manner

$$(E_1 \bowtie_{Q1} E_2) \bowtie_{Q1 \wedge Q2} E_3 = E_1 \bowtie_{Q1 \wedge Q3} (E_2 \bowtie_{Q2} E_3)$$

where Q_2 involves attributes from only E_2 and E_3

- Selection operation is distributes over the theta join operation under the following two conditions.
- When all attributes in Q_O involves only attributes of one of the expression (E_1) being joined.

$$\sigma_{Q0}(E_1 \bowtie_Q) = (\sigma_{Q0}(E_1)) \bowtie_Q E_2$$

- When Q_1 involves only the attributes of E_1 and Q_2 involves only the attribute of E_2

$$\sigma_{Q1 \wedge Q2}(E_1 \bowtie E_2) = \left(\sigma_{Q1}(E_1)\right) \bowtie (\sigma_{Q2}(E_2))$$

- Projection operation distributes over Q join operation.
- Set operations union and intersection are cumulative.
- Set operations union and intersection are associative.
- Selection operation is distributed over union intersection and difference.
- Projection operation is distributed over union.

7.2.3 Introduction to Query optimization: Query Optimization is the process of choosing the most efficient way to execute a SQL statement.

The optimizer feature allows us to create statistics collection, selectivity, and cost functions that are used by the optimizer in choosing a query plan. The optimizer cost model is extended to integrate information supplied by the user to assess CPU and the I/O cost, where CPU cost

is the number of machine instructions used, and I/O cost is the number of data blocks fetched. We can perform the following

- Associate cost functions and default costs with domain indexes (partitioned or unpartitioned), index types, packages, and standalone functions. The optimizer can obtain the cost of scanning a single partition of a domain index, multiple domain index partitions, or an entire index.
- Associate selectivity functions and default selectivity with methods of object types, package functions, and standalone functions. The optimizer can estimate user-defined selectivity for a single partition, multiple partitions, or the entire table involved in a query.
- Associate statistics collection functions with domain indexes and columns of tables. The optimizer can collect user-defined statistics at both the partition level and the object level for a domain index or a table.
- Order predicates with functions based on cost.
- Select a user-defined access method (domain index) for a table based on access cost.
- Use new data dictionary views to include information about the statistics collection, cost, or selectivity functions associated with columns, domain indexes, indextypes or functions.
- Add a hint to preserve the order of evaluation for function predicates.

7.3 CURSORS

A cursor is a named control structure used by an application program to point to and select a row of data from a result set. Instead of executing a query all at once, you can use a cursor to read and process the query result set one row at a time.

7.3.1 Types of cursors:

Implicit cursors: These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

Attributes of Implicit cursors:

- %ISOPEN: It always returns FALSE, because the database closes the SQL cursor automatically after executing its associated SQL statement.
- %FOUND: It evaluates to TRUE, if an insert, update or delete affects one or more rows, or a single-row select returned one or more rows. Otherwise it returns false.
- %NOTFOUND: It is logical opposite of %FOUND. The value returned is just opposite to that of %FOUND.
- %ROWCOUNT: Returns the number of rows affected by an insert, update or delete or select into statement.

Example: Following is a PL/SQL block for illustrating an implicit cursor for number of rows affected by an update.

SQL%ROWCOUNT returns numbers of rows updated. It can be used as follows:

```
BEGIN
        UPDATE Customers
        SET Cust_name = 'B'
        WHERE Cust_name LIKE 'B%';
        DBMS_OUTPUT.PUT_LINE (SQL%ROWCOUNT);
END;
```

Explicit cursors: They must be created when we execute a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When we fetch a row the current row position moves to next row.

An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT statement which returns more than one row. We can provide a suitable name for the cursor.

The general syntax for creating a cursor is as given below:

CURSOR cursor_name IS select_statement;

- cursor_name – A suitable name for the cursor.
- select_statement – A select query which returns multiple rows.

There are four steps in using an Explicit Cursor.

- DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
- CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

7.3.2 Cursor for loops: There are three types of cursors for loops namely SIMPLE LOOP, WHILE LOOP and FOR LOOP. These loops can be used to process multiple rows in the cursor

Example: The following example illustrates cursor with simple loop

```
DECLARE
  CURSOR emp_cur IS
   SELECT first_name, last_name, salary FROM emp_tbl;
   emp_rec emp_cur%rowtype;
BEGIN
        IF NOT sales_cur%ISOPEN THEN
            OPEN sales_cur;
        END IF;
        LOOP
                FETCH emp_cur INTO emp_rec;
```

```
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(emp_cur.first_name || ' '
    ||emp_cur.last_name
        || ' ' ||emp_cur.salary);
    END LOOP;
END;
```

7.3.3 Cursor variables: A cursor variable is a cursor that contains a pointer to a query result set. It holds the memory location (address) of some item instead of the item itself.

Cursor variables let you:

- Associate a cursor variable with different queries at different times in your program execution. In other words, a single cursor variable can be used to fetch from different result sets.
- Pass a cursor variable as an argument to a procedure or function. We can, in essence, share the results of a cursor by passing the reference to that result set.
- Employ the full functionality of static PL/SQL cursors for cursor variables. We can OPEN, CLOSE, and FETCH with cursor variables within PL/SQL programs. We can reference the standard cursor attributes -- %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT -- for cursor variables.
- Assign the contents of one cursor (and its result set) to another cursor variable. Because the cursor variable is a variable, it can be used in assignment operations. There are, however, restrictions on referencing this kind of variable.

7.3.4 Parameterized cursors: Parameterized cursors are static cursors that can accept passed-in parameter values when they are opened. The following example includes a parameterized cursor. The cursor displays the name and salary of each employee in the EMP table whose salary is less than that specified by a passed-in parameter value.

```
DECLARE
  my_record  emp%ROWTYPE;
  CURSOR c1 (max_wage NUMBER) IS
```

```
    SELECT * FROM emp WHERE sal < max_wage;
BEGIN
  OPEN c1 (2000);
  LOOP
    FETCH c1 INTO my_record;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
      || my_record.sal);
  END LOOP;
  CLOSE c1;
END;
```

7.4 LET US SUM UP

We learnt about query evaluation and its techniques in this chapter. The techniques involved system catalogue and evaluation of relational operators. We also saw the basic definition of query optimization. Then we learnt about cursors in PL/SQL. We saw the types of cursors which are implicit and explicit cursors. Further we learnt about cursor for loops and parameterized cursors.

7.5 REFERENCES AND SUGGESTED READING

(1) Ramakrishnam Gehrke, “Database Management Systems”, McGraw- Hill.

(2) Ivan Bayross, “SQL,PL/SQL -The Programming language of Oracle”, B.P.B. Publications, 3rd Revised Edition

(3) Michael Abbey, Michael J. Corey, Ian Abramson, Oracle 8i – A Beginner’s Guide, Tata McGraw-Hill.

7.6 EXERCISE

1. Explain query evaluation and query optimization.

2. What is system catalog?

3. Explain cursors and its types.

4. Differentiate between implicit and explicit cursors.

5. Discuss cursor for loops with an example.

6. Define parameterized cursors and its advantage.

8

TRANSACTIONS IN SQL

Unit Structure

8.0 OBJECTIVES

The objective of this chapter is

- To understand the way of defining transaction
- To understand COMMIT, ROLLBACK and SAVEPOINT command
- To understand read only transactions
- To understand explicit locks
- To understand locking strategy

- To understand sequences in PL/SQL

8.1 INTRODUCTION

Transactions are a mechanism for simplifying the development of distributed multiuser enterprise applications. By enforcing strict rules on an application's ability to access and update data, transactions ensure data integrity. A transactional system ensures that a unit of work either fully completes or the work is fully rolled back. Transactions free an application programmer from dealing with the complex issues of failure recovery and multiuser programming. In this chapter we will study the transactions and the mechanisms involved in its management.

8.2 DEFINING A TRANSACTION

Transactions are logical units of work you use to split up your database activities. A transaction has both a beginning and an end.

A transaction begins when one of the following events occurs:

- Connection to the database takes place and the first DML statement is performed.
- A previous transaction ends and some other DML statement is entered.

A transaction ends when one of the following events occurs:

- A COMMIT or a ROLLBACK statement is performed.
- A DDL statement, such as a CREATE TABLE statement is performed, in which case a COMMIT is automatically performed.
- A DCL statement, such as a GRANT statement is performed, in which case a COMMIT is automatically performed.
- Disconnection from the database takes place.
- By entering the EXIT command, a COMMIT is automatically performed.
- If termination occurs abnormally, a ROLLBACK is automatically performed.

- A DML statement that fails is performed, in which case a ROLLBACK is automatically performed for that individual DML statement.

8.3 MAKING CHANGES PERMANENT WITH COMMIT

The COMMIT statement ends the current transaction, making any changes made during that transaction permanent, and visible to other users.

Syntax:

```
Commit;
```

Example:

```
BEGIN
    UPDATE emp_information SET emp_dept = 'Web Developer'
    WHERE emp_name = 'Jawahar';
  COMMIT;
END;
```

8.4 UNDOING CHANGES WITH ROLLBACK

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

Syntax:

```
Rollback;
```

Example:

```
DECLARE
  emp_id  emp.empno%TYPE;
        BEGIN
              SAVEPOINT dup_found;
              UPDATE emp SET eno=1
              WHERE empname = 'Ajay'
        EXCEPTION
        WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO dup_found;
END;
```

8.5 UNDOING PARTIAL CHANGES WITH SAVEPOINT AND ROLLBACK

SAVEPOINT names and marks the current point in the processing of a transaction. Savepoints let you roll back part of a transaction instead of the whole transaction.

- A simple rollback or commit erases all savepoints. When a savepoint is rolled back, any savepoints marked after that savepoint are erased. The savepoint to which we roll back remains.
- We can reuse savepoint names within a transaction. The savepoint moves from its old position to the current point in the transaction.
- If a savepoint is marked within a recursive subprogram, new instances of the SAVEPOINT statement are executed at each level in the recursive descent. It can be only rolled back to the most recently marked savepoint.
- An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction; if the statement raises an unhandled exception, the host environment (such as SQL*Plus) determines what is rolled back.

Syntax:

```
SAVEPOINT SAVEPOINT_NAME;
```

Example: The PL/SQL block below illustrates the use of SAVEPOINT statement

```
DECLARE
  emp_id  emp.empno%TYPE;
      BEGIN
            SAVEPOINT dup_found;
            UPDATE emp SET eno=1
           WHERE empname = 'Ajay'
      EXCEPTION
            WHEN DUP_VAL_ON_INDEX THEN
           ROLLBACK TO dup_found;
END;
```

8.6 DEFINING READ ONLY TRANSACTIONS

Read-only transactions are useful for running multiple queries against one or more tables while other users update the same tables. A transaction can be defined as read only by using SET TRANSACTION statement.

Syntax:

```
SET TRANSACTION READ ONLY;
```

Example:

```
COMMIT;  -- end previous transaction
SET TRANSACTION READ ONLY;
SELECT ... FROM emp WHERE ...
```

```
SELECT ... FROM dept WHERE ...
SELECT ... FROM emp WHERE ...
COMMIT;  -- end read-only transaction
```

8.7 EXPLICIT LOCKS – TRANSACTION AND SYSTEM LEVEL

A LOCK is a mechanism that prevents destructive interaction between two simultaneous transactions or sessions trying to access the same database object. A LOCK can be achieved in two ways: Implicit locking or Explicit Locking. The session remains in a waiting state until one of the sessions is either committed or rolled back.

If a server implicitly creates a deadlock situation if a transaction is done on the same table in different sessions. This default locking mechanism is called **implicit** or **automatic locking**.

With **Explicit Locking**, a table or partition can be locked using the LOCK TABLE statement in one of the specified modes. The available lock modes are ROW EXCLUSIVE, SHARE UPDATE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, NOWAIT and WAIT. The syntax for locking command is as follows:

```
LOCK TABLE [TABLE NAME] IN [LOCK MODE] [WAIT | NOWAIT]
```

8.8 LOCKING STRATEGY: ROW SHARE AND ROW EXCLUSIVE MODE

ROW SHARE LOCKS:

- Row-level shared locks allow multiple users to read data, but do not allow any users to change that data.
- Table-level shared locks allow multiple users to perform read and write operations on the table, but do not allow any users to perform DDL operations.
- Multiple users can hold shared locks simultaneously.

ROW EXCLUSIVE LOCKS: An exclusive lock allows only one user/connection to update a particular piece of data (insert, update, and delete). When one user has an exclusive lock on a row or table, no other lock of any type may be placed on it.

The following statements shows the use of LOC TABLE command in the above mentioned two modes.

```
LOCK TABLE emp IN ROW EXCLUSIVE MODE;

LOCK TABLE emp, dept IN SHARE MODE NOWAIT;
```

In the LOCK TABLE statement, we can also indicate how long we want to wait for the table lock:

- For no waiting, specify either NOWAIT or WAIT 0.
- We can acquire the table lock only if it is immediately available; otherwise, an error notifies that the lock is not available now.
- To wait up to n seconds to acquire the table lock, specify WAIT n, where n is greater than 0 and less than or equal to 100000.
- If the table lock is still unavailable after n seconds, an error notifies you that the lock is not available now.
- To wait indefinitely to acquire the lock, specify neither NOWAIT nor WAIT.
- The database waits indefinitely until the table is available, locks it, and returns control to you. When the database is running DDL statements concurrently with DML statements, a timeout or deadlock can sometimes result. The database detects such timeouts and deadlocks and returns an error.

8.9 SEQUENCE

A sequence is a database object that is used to generate sequential number.

- **Creating a sequence**: We can create a sequence using the CREATE SEQUENCE statement, which has the following syntax:

```
CREATE SEQUENCE sequence_name
```

[START WITH start_num]
[INCREMENT BY increment_num]
[{ MAXVALUE maximum_num | NOMAXVALUE }]
[{ MINVALUE minimum_num | NOMINVALUE }]
[{ CYCLE | NOCYCLE }]
[{ CACHE cache_num | NOCACHE }]
[{ ORDER | NOORDER }];

where INCREMENT BY specifies the interval between sequence numbers. MINVALUE specify the minimum value, NOMINVALUE specifies a minimum value of 1 for an ascending sequence and –(10)^26 for a descending sequence. MAXVALUE specify the maximum value, NOMAXVALUE specifies a maximum value of (10)^26 for an ascending sequence and 1 for a descending sequence. START WITH specifies the first sequence number to be generated. CYCLE specifies that the sequence continues to generate repeat values after reaching either maximum or minimum value. NOCYCLE specifies that there will be no cycle. CACHE specifies how many values of a sequence Oracle pre-allocates and keeps in memory for faster access. NOCACHE specifies that values of a sequence are not pre-allocated.

- **Referencing a sequence**: Once a sequence is created in SQL, it can be used to view the values held in its cache. To view the sequence value we can use a SELECT statement as described below :
  SELECT sequence_name. NextVal FROM DUAL.

- **Altering a sequence**: We can use the ALTER SEQUENCE statement to change the increment, minimum and maximum values, cached numbers, and behavior of an existing sequence. This statement affects only future sequence numbers. The syntax is as follows

  ALTER SEQUENCE seq_cache NOCACHE;.
  ALTER SEQUENCE seq_cache INCREMENT BY xx

- **Dropping a sequence**: The DROP SEQUENCE statement is used to remove a sequence from the database. The syntax for this statement is:

  DROP SEQUENCE sequence_name

8.10 LET US SUM UP

In this chapter we learnt about the COMMIT command and its use in making changes permanent. Similarly we learnt about ROLLBACK command and its use in undoing changes. Use of SAVEPOINT and ROLLBACK for undoing partial changes was also studied. We also learnt to define a transaction as read only. Then we learnt about explicit locks. We learnt about locking strategies and ROW SHARE and ROW EXCLUSIVE modes of locking. Finally we learnt about sequence, its creation, referencing, alteration and deletion.

8.11 REFERENCES AND SUGGESTED READING

(1) Ramakrishnam Gehrke, "Database Management Systems", McGraw- Hill.

(2) Ivan Bayross, "SQL,PL/SQL -The Programming language of Oracle", B.P.B. Publications, 3rd Revised Edition

(3) Michael Abbey, Michael J. Corey, Ian Abramson, Oracle 8i – A Beginner's Guide, Tata McGraw-Hill.

8.12 EXERCISE

1. Explain transactions.
2. Explain Commit command with an example.
3. What is the significance of Rollback command?
4. Describe SAVEPOINT command with a suitable example.
5. How will you define a read only transaction?
6. Explain different types of locking commands used for locking table.
7. Explain sequences with and example.

9

PROJECT MANAGEMENT AND SCHEDULING

Unit Structure

9.0 OBJECTIVES

The objective of this chapter is

- To understand the project management process
- To understand role of project manager
- To understand project management knowledge areas
- To understand role of software metrics
- To understand the process of building WBS
- To understand the use of Gantt and PERT/CPM chart
- To understand the concept of Staffing

9.1 INTRODUCTION

Good project management and good engineering are essential for successful project. For project management, project planning is required. Software project planning includes size effort estimation, and project scheduling. In past several projects have failed due to project management.

Scheduling the software project involves identifying all the tasks necessary to complete the project. In order to schedule the project activities a software project manager needs to do following:

- Identify all the tasks needs to compete the project.
- Break down large tasks into small activity
- Determine the dependency among different activities.
- Establish most suitable estimate for time durations necessary to complete the activities.
- Allocate resources to activities.
- Plan the starting and ending dates for various activities.
- Determine the critical path.

9.2 REVISION OF PROJECT MANAGEMENT PROCESS

The primary goal of software project management is to enable a group of software engineers to work efficiently towards successful completion of the project. In case of small projects software engineers assumes responsibilities of project manager in addition. But in large projects have a full time project manager.

Project management in a software system is the process of defining, planning, organizing, leading and controlling the development of a software project. The goal of project management is to deliver software product that is acceptable to users and is developed on time and within budget. There are four phases of project management process:

- **Initiating the project**: This is first phase of project management process in which activities are performed to assess the size, scope, and complexity of the project and to establish procedures to support later project activities.
- **Planning the project**: The Project planning provides an overall framework for managing project costs and schedules. It takes place at the beginning and at the end of each project phase. Project planning involves defining clear, discrete activities or tasks and the work needed to complete a project.
- **Executing the project**: The third phase in Project management process in which the plans created in the prior project phases are put to action. If you develop a high quality project plan, it is much more likely that the project will be successfully executed.
- **Closing down the project**: This is the final phase of project management process which focuses on bringing a project to an end. Closedown is a very important activity since a project is not complete until it is closed and it is at closedown that projects are deemed a success or failure.

9.3 ROLE OF PROJECT MANAGER

The job responsibilities of project manager range from invisible activities like building up team morale to highly visible customer presentations. Roles of project manager are project proposal writing, project cost estimation, scheduling, project staffing, software processes tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, and so on. The project planning activity is undertaken before development start. Project monitoring and control is undertaken after begun of project. The changes to plan whenever required are made to continuously cope with situations.
On the basis of above points, we can say that the project manager does the following

- Manage the project taking into account integration across all areas.
- Develop Project Plan.
- Direct project resources.
- Monitor and manage the project schedule.
- Monitor and manage the project budget.
- Monitor and manage the project risk.

- Deal with operational issues.
- Ensure project meets requirements and objectives .
- Manage project team members.
- Negotiate and resolve issues as they arise across areas of the project and where they impact on other activities, systems and projects.
- Look after the interests of the project team.
- Maintain project documentation.

9.4 PROJECT MANAGEMENT KNOWLEDGE AREAS

Knowledge about different project management techniques is necessary to become successful project manager. Effective software project management requires good qualitative judgment and decision-making capabilities. Project management knowledge areas are the latest software project management techniques such as cost estimation, risk management, configuration management. Project managers need good communication and interpersonal skills, so that it is useful to get work done. However, tracking and controlling the progress of the project customer interaction, managerial presentation, and team building are largely acquired through experience.

Project management knowledge areas have been structures into nine areas:

- **Integration Management**: It is related to fitting everything together, planning and project changes.
- **Project Scope Management**: It is responsible for clear scope statement and to prevent scope creep.
- **Project Time Management**: It is meant planning and management of time and schedule.
- **Project Cost Management**: It deals with management of costs which is out of our control and for competing projects.
- **Project Quality Management**: It is responsible for planning quality, enforcing quality and checking quality control.
- **Project Human Resource Management**: It deals with organizational planning, staff acquisition and formation of a team.
- **Project Communications Management**: It is related and responsible for communication plan.

- **Project Risk Management**: It deals with risk management plans.
- **Project Procurement Management**: It is responsible for acquisition and contract management.

9.5 MANAGING CHANGES IN REQUIREMENT

One of the challenging issues is to manage changes in requirements. It can occur at any time during life of a project. Management of changes in requirement becomes more severe with going farther down in life cycle. When there are some new requirements or changes to existing requirement are made, the management process defines a set of activities to be performed. The steps that should be followed are

- Log the changes
- Perform impact analysis on the work products
- Estimate impact on effort and schedule
- Review impact with concerned stakeholders
- Rework work products.

9.6 ROLE OF SOFTWARE METRICS

A quantitative measurement of degree to which the system component or process possesses a given attribute is termed as software metric. Software metric plays important role in the effort, cost and duration estimation. In order to accurately estimate the project size we need to define appropriate metric or unit in terms of which we can express the project size. The role of software metrics can be summarized under following points

- Estimate the cost & schedule of future projects.
- Evaluate the productivity impacts of new tools and techniques.
- Establish productivity trends over time.
- Improve software quality.
- Forecast future staffing needs.
- Anticipate and reduce future maintenance needs.

9.7 PROJECT SCHEDULING

The management of large projects requires analytical tools for scheduling activities and allocating resources. Project scheduling is one of the key process in project management. The main objectives of project scheduling are following:

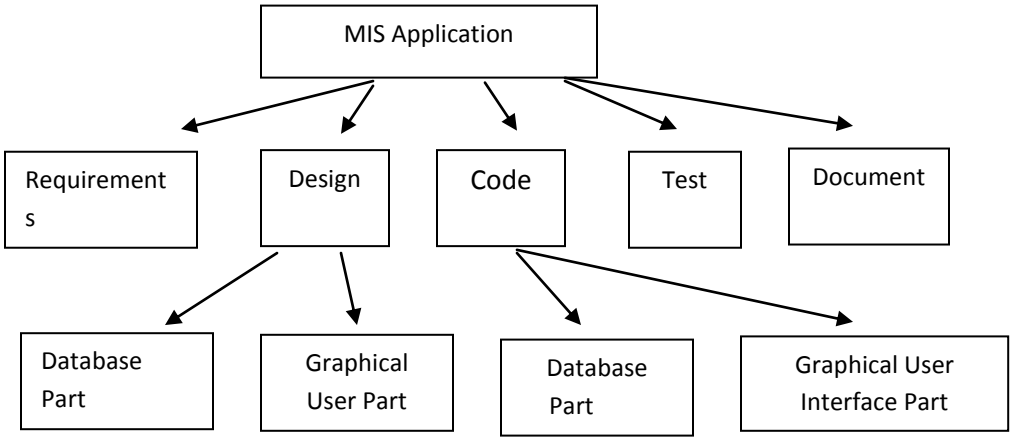
- Completing the project as early as possible by determining the earliest start and finish of each activity.
- Calculating the likelihood a project will be completed within a certain time period.
- Finding the minimum cost schedule needed to complete the project by a certain date.
- Investigating the results of possible delays in activity's completion time.
- Progress control.
- Smoothing out resource allocation over the duration of the project.

There are various models of project scheduling which will be discussed in this chapter.

9.7 BUILDING WBS

Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides notion of major task needed to be carried out in order to solve a problem. The root node of tree is named as problem. Each node of tree is broken down into small activities that are made at leaf level, while breaking down a large task into decisions. If task is broken into a large number of very small activities, these can be less efficient.

Work Break down Structure of a MIS Problem



There are several advantages of WBS; in short we can say that WBS helps to:

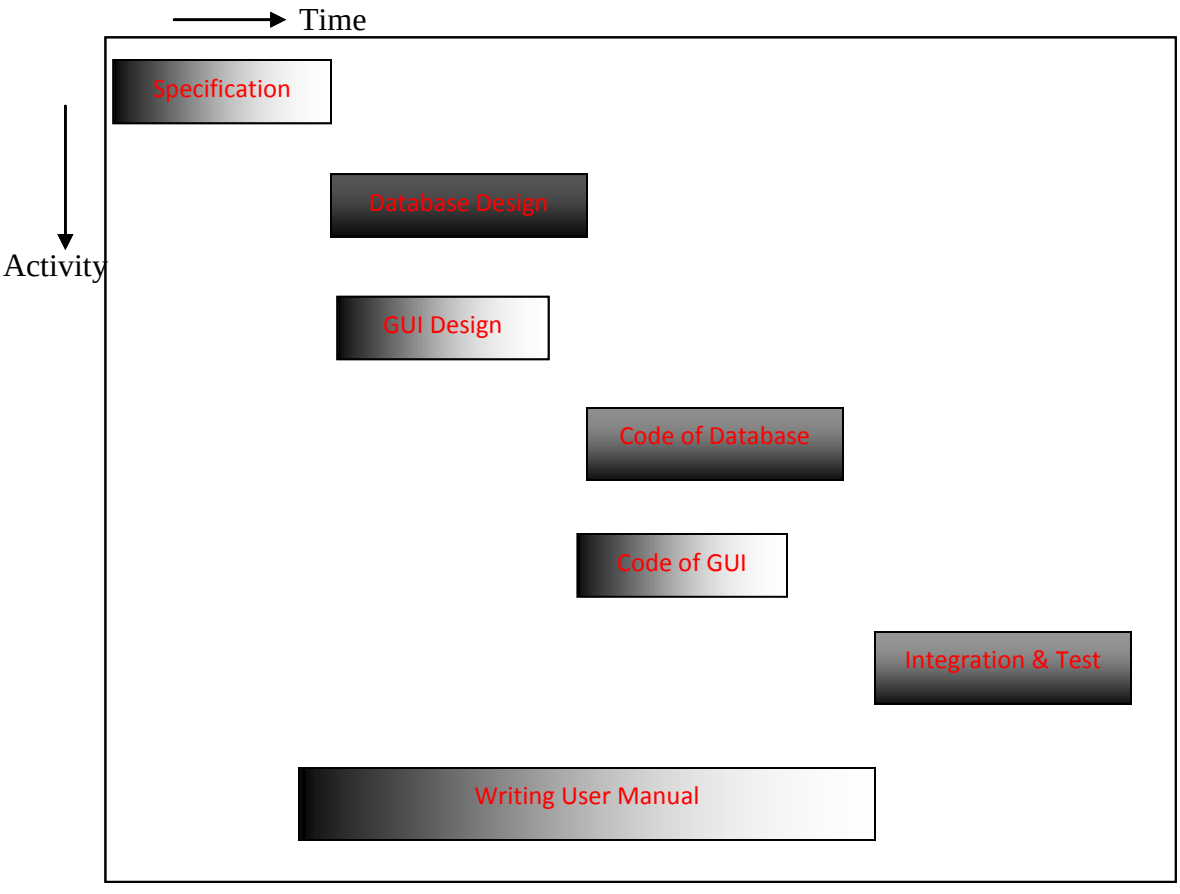
- Identify all work needing to be done.
- Logically organize work so that is can be scheduled.
- Assign work to team members.
- Identify resources needed.
- Communicate what has to be done.
- Organize work using milestones.

9.8 USE OF GANTT AND PERT/CPM CHART

GANTT Chart:

GANTT chart is basically a graphical representation of a project that shows each task as a horizontal bar whose length is proportional to its time for completion. It is mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware and software. Following figure shows an example of Gantt chart

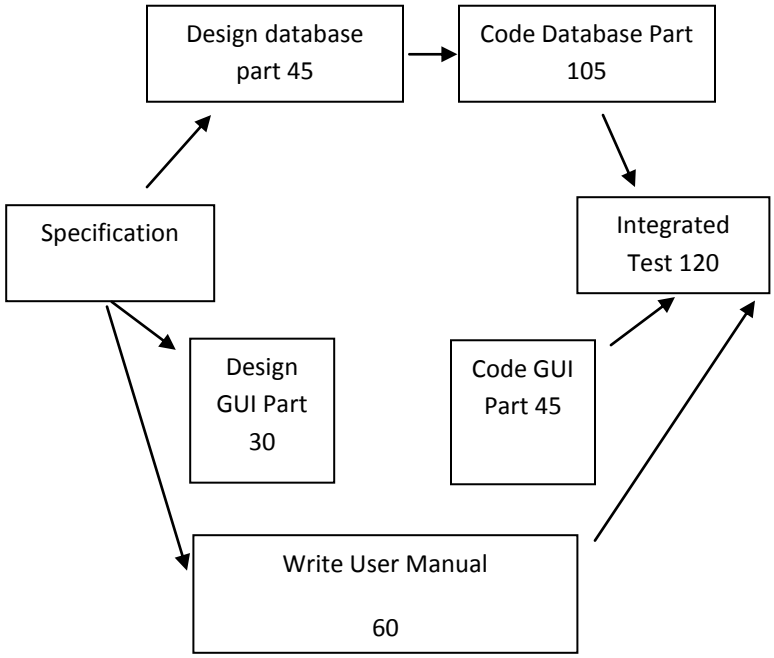
Gantt chart representation of MIS problem



Gantt chart is special type of bar chart where each bar represents an activity. The bars are drawn along time line. The length of each bar is proportional to duration. In Gant chart each bar consist white part and shaded part. The shaded part shows length of time each task has taken and the remaining portion is the latest time by which task must be finished.

CPM Chart: CPM (Critical Path Method) is used in critical path. A critical path is the chain of activities that determines the duration of the project. Activity networks of MIS problem is shown in figure below

Activity network representation of MIS problem



In activity network each activity is represented by rectangular node and duration of activity shown in alongside.

From activity network we can say about (i) Minimum time (MT) to complete the project finish; (ii) Earliest Start (ES) time of all paths from start to this task. (iii) The latest start (IS) time is difference between MT and maximum of all path from this task to finish; (iv) Earliest Finish Time (EFT) of task is the sum of earliest start time of task and duration of task (v) Latest Finish (LF) time of task can be obtained by subtracting maximum of all paths from this task to finish from MT. The Slack Time (ST) is LS – EF and equivalently can be written as LF – EF. The

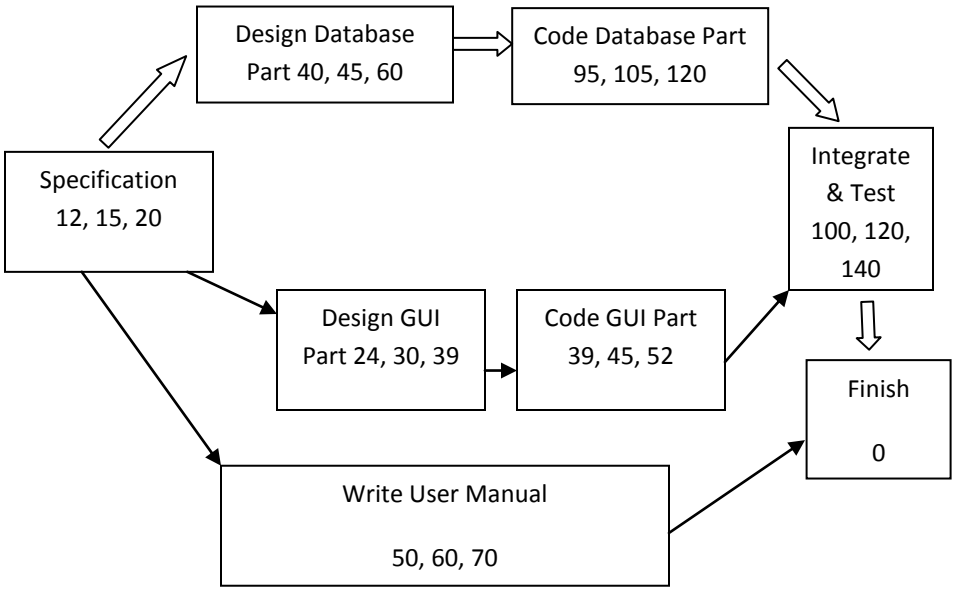
Slack Time or float time is total time for which a task may be delayed before it would affect the schedule.

A path from start node to the finish node contains only critical tasks called a critical path. The different tasks for MIS problem are shown in table below

Task	ES	EF	LS	LF	ST
Specified Part	0	15	0	15	0
Database Design Part	15	60	15	60	0
Design GUI Part	15	45	90	120	75
Code Database Part	60	165	60	165	0
Code GUI Part	45	90	12	165	75
Integrated Test	165	295	165	295	0
Write User Manual	15	75	225	295	210

PERT Chart: PERT (Project Evaluation and Review Technique) chart consist of a network of boxes and arrows. The boxes represent task dependencies. This chart represents statistical variations in the project estimation as normal distribution. Thus in this chart we make pessimistic likely and optimistic estimate. Since all possible completion every task has to be considered. There are many critical paths, depending on permutation of estimate for each task. This makes critical path analysis in PERT charts very complex. A critical path is shown figure below by using double line arrow. PERT charts are more complex form of estimate task durations is represented. Since actual utility of activity diagram is limited. PERT chart is useful for monitoring the timely progress of activities. It is easier to identity parallel activities using PERT chart. Scheduling in project for assignment to different engineers is shown by PERT chart. The PERT chart representation of MIS problem is shown in figure below

PERT Chart Representation of MIS Problem



9.9 STAFFING

A staff is required in order to execute work tasks and activities. If you are a project manager, you need to have an adequate staff for executing your project activities. Staffing function is one of the most important mangerial act along with planning, organizing, directing and controlling. The operations of these four functions depend upon the manpower which is available through staffing function.

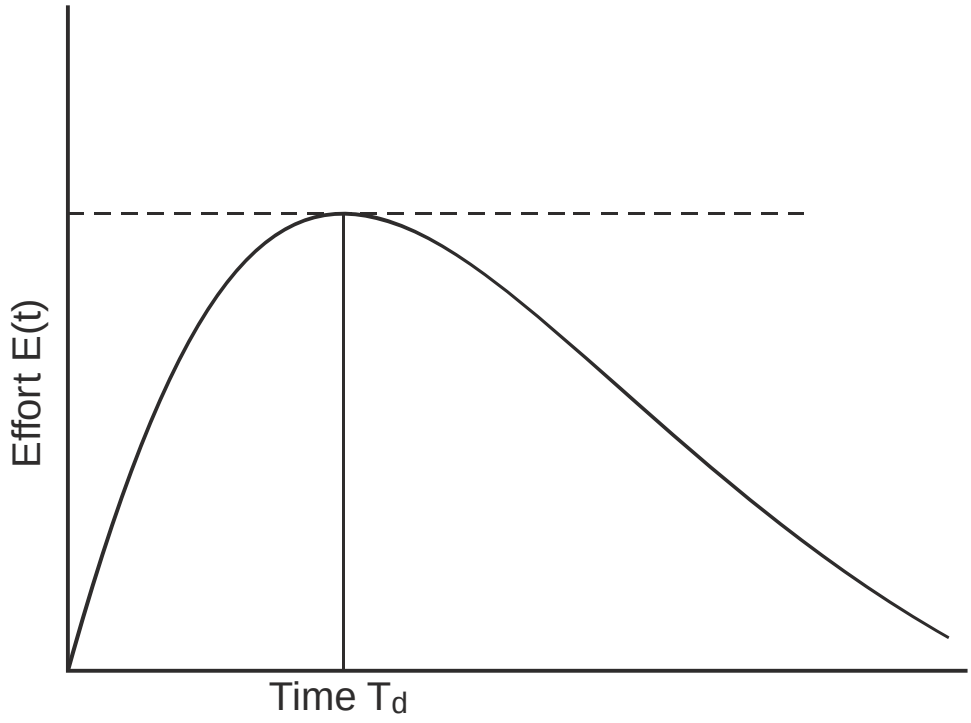
When effort required to develop to software is determined, it is necessary staffing requirement for project. To understand staffing requirement we should study Norden's and Putnam's results.

- **Norden's Work:** Norden found that staffing can be approximated by Rayleigh distribution curve. Norden represented Rayleigh curve by following equation

$$E(t) = \frac{k}{t_{d^2}} \times t\alpha\, \bar{e}\, t^2 \Big/ 2td^2$$

where E is the effort required at time t. k is area under curve and t_d is time which curve attains maximum value

Rayleigh Curve



- **Putnam's Work:** By analyzing large number of army projects Putnam derived following expression $L = C_k K^{1/3} t_d^{4/3}$, where K is total effort required in product development and L is the product size in K_{LOC}, t_d corresponds to time of system integration and testing. Therefore t_d can be approximately considered as the time required to develop the software. C_k is state of technology constant. The value of C_k = 2 corresponds to poor development environment (no methodology, poor documentation and review), C_k = 9 corresponds to good development environment (software engineering principles are used in efficient manner). C_k =11 corresponds to an excellent environment in which with software engineering principles automated tool are used. If we examine Rayleigh distribution curve, then approximately we can see that 4% area is to the left of t_d and 60% to the right of t_d. This implies that ratio of effort required is 40:60.

  According to Putnam's Law $\frac{k_1}{k_2} = \left(\frac{t_{d^2}}{t_{d^1}}\right)^4$.

  The Putnam's estimation model works reasonably well for very large system but seriously over estimate for small projects.

- **Jensen's Model:** Jensen, 1904 model is similar to Putnam model. $L = C_{te} t_d k^{1/2}$. Then, $\frac{k_1}{k_2} = \left(\frac{t_{d_2}}{t_{d_1}} \right)^2$, where C_{te} is effective technology constant, t_d is the time to develop software, and k is effort needed to develop software. When the project duration is compressed. Jensen's model gives the increase in effort (and cost) requirement proportional to the square of degree of compression.

9.10 LET US SUM UP

We revised the project management process first. Then we learnt the role of project manager. Project management knowledge areas followed by management of changes in requirement were then studied. We then saw role of software metrics. We also learnt the process of building WBS. Use of Gantt and PERT/CPM chart was explained. At last we learnt about the staffing process.

9.11 REFERENCES AND SUGGESTED READING

(1) Software Engineering, Practitioner Approach, 7th Edition, by R.S. Pressman, Tata McGraw Hill, India, 2009.
(2) Integrated Approach to Software Engineering , Pankaj Jalote, Narosa Publications, 2003.
(3) Introduction to Software engineering by Rajiv Mall – PHI, 2000.
(4) Software Engineering by I. Somerville, 7th Edition, Pearson Education, India, 2006.

9.12 EXERCISE

1. Explain project management process.
2. What is the role of project manager?
3. Describe project management knowledge areas.
4. What is the role of software metrics?
5. Explain the process of building WBS.
6. Describe the use of Gantt & PERT/CPM Chart.
7. Explain staffing.

10

SIZE AND EFFORT ESTIMATION

Unit Structure

10.0 OBJECTIVES

The objective of this chapter is

- To understand the basics concept of LOC and estimation
- To understand function point
- To understand COCOMO model
- To understand concept of effort estimation and uncertainty

10.1 INTRODUCTION

Efforts estimation for software development is the process of forecasting the most realistic use of effort required to develop or maintain software based on incomplete, uncertain and/or

noisy input. Effort estimates can be used as input to project plans, iteration plans, budgets, and investment analyses, pricing processes and bidding rounds.

The ability to accurately estimate the time and cost for a project to come to its successful completion has been a major problem for software engineers. In recent years, the use of repeatable, clearly defined and well understood software development process has been the most effective method for gaining useful historical data that can be used for statistical estimation. The act of sampling particularly, coupled with the loosening of constraints between parts of a project, has allowed more accurate estimation and more rapid development times.

Metrics are quantifiable measures that could be used to measure characteristics of a software system or the software development process. E.g., Number of errors found per person hours expended.

- Software metrics is a term that embraces many activities, all of which involve some degree of software measurement:
    - cost and effort estimation
    - productivity measures and models
    - data collection
    - quality models and measures
    - reliability models
    - performance evaluation and model

- We can specify software metric in 12 steps :
    - Step 1 - Identify Metrics Customers
    - Step 2 - Target Goals
    - Step 3 - Ask Questions
    - Step 4 - Select Metrics
    - Step 5 - Standardize Definitions
    - Step 6 - Choose a Model
    - Step 7 - Establish Counting Criteria
    - Step 8 - Decide On Decision Criteria
    - Step 9 - Define Reporting Mechanisms

- Step 10 - Determine Additional Qualifiers
- Step 11 - Collect Data
- Step 12 - Consider Human Factors

The software metrics can be product oriented or process oriented

- **Process Metrics**
  - Insights of process paradigm, software engineering tasks, work product, or milestones
  - Lead to long term process improvement

- **Product Metrics**
  - Assesses the state of the project
  - Track potential risks
  - Uncover problem areas
  - Adjust workflow or tasks
  - Evaluate teams ability to control quality

There are three type of metrics based on component level

- Cohesion (internal interaction) - a function of data objects.
- Coupling (external interaction) - a function of input and output parameters, global variables, and called modules.
- Complexity of program flow - hundreds have been proposed (e.g., cyclomatic complexity).

10.2 CONCEPT OF LOC AND ESTIMATION

The LOC code can be described by following points:

- LOC (Lines of Code) is equal code, excluding comment and documentation.
- It is simple.

- At the time of counting of number of instructions comments and header lines are ignored.
- LOC gives numerical value of the problem size.
- It is programming language and programming style dependent.
- We cannot determine LOC at the beginning.
- LOC metric penalizes use of high-level programming languages, code reuse, etc.
- It also penalizes metric measures the lexical complexity of program and does not address issues of logical or structural complexities between two programs with equal penalizes count, a program having complex logic would required much more effort to develop than very simple logic.
- It is very difficult to accurately estimate LOC in the final product from the problem specified.

10.3 FUNCTION POINT

The function point method was originally developed by Allan J. Albrecht. A function point is an approximate estimate of a unit of delivered functionality of a software project. Function points (FP) measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an unadjusted function point count (UFP). Counts are made for the following categories

- **Number of user inputs**: user inputs that provides distinct application oriented data to the software is counted.
- **Number of outputs**: The output of each user that provides application oriented information to the user is counted. "Output" in this context refers to reports, screens, error messages, etc. The individual data items contained in a report are not counted separately.
- **Number of user inquiries**: It is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. All distinct inquiries are counted.
- **Number of files**: All logical master file is counted.

- **Number of external interfaces**: Each machine-readable interface that is used to transmit information to another system is counted.

There are 14 technical complexity factors which are also used in determining function point metric. The table of calculating UFP (Unadjusted Function Point) is given below:

Weighing Factor W1 ⟶

Measurement Parameter	Count	Simple	Average	Complex
Number Inputs	C1	3	4	6
Number of Outputs	C2	4	5	7
Number of Inquiries	C3	3	4	6
Number of Files	C4	7	10	15
Number of External Interfaces	C5	5	7	10

Count total = $\sum_{C=1}^{5} CiWi$

UFP = Count Total

The TCF (Technical complexity factor) depends on 14 factors which is shown in the table below

F1	Reliable back-up and recovery
F2	Data communications
F3	Distributed functions
F4	Performance
F5	Heavily used configuration

F6	Online data entry
F7	Operational ease
F8	Online update
F9	Complex interface
F10	Complex processing
F11	Reusability
F12	Installation ease
F13	Multiple sites
F14	Facilitate change

$TCF = 0.65 + 0.01 \times \sum F_i$

where F_i is the value of i^{th} factor.

For all F_i's, the minimum value is 0 means that it is not required and maximum value of is 5 means that it is essential.

The function point metric can now be calculated as

$FP = UFP \times TCF$

Example: Calculate the function point value for project having, number of user inputs =32, number of user outputs =60, number of user inquiries =24, number of files =8, and number of interfaces =2. Assume that all complexities are average. In this case each F_i is 3

$$UFP = 32 \times 4 + 60 \times 5 + 24 \times 4 + 8 \times 10 + 2 \times 7 = 618$$

$$TCF = 0.65 + 0.01 \times \sum_{i=1}^{14} 3 = 0.65 + 0.01 \times 42 = 1.07$$

$$FP = UFP \times TCF = 618 \times 1.07 = 661.326 \approx 661$$

10.4 COCOMO MODEL

COCOMO stands for Constructive Cost Model and is a heuristic estimation technique proposed by W.H. Barry Bohem in 1981. Bohem saw that ratios of relative levels given by

Brooks in 1975 are not good. Because product development complexity for application utility and system is 1:3:9. She has proposed three COCOMO models. First is Basic COCOMO in which she considered three types of application organic, semidetached and embedded. Organic type deals with well understood application with experience people. Semidetached type is mixture of experience and inexperience people. Embedded types are those which deals strongly coupled complex applications.

Basic COCOMO equation takes the form

$$\text{Effort}(E) = a_b(\text{KLOC})^{b_b}\ \text{PM}(\text{Person} - \text{months})$$

$\text{Time of Development}(\text{Tdev}) = C_d(E)^{d_b}$, where coefficients a_b, b_b, c_b and d_d are given below

Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detach	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Example: Size of organic type software product has been estimated to be 32, 000 lines of source code. Average salary of software engineers is 10, 000$ per month. Calculate effort required to develop software product and nominal development time

$$Effort = (2.4) \times (32)^{1.05} = 91\ PM$$

$$Nominal\ Development\ time = 2.5(91)^{0.38} = 14\ months$$

$$Cost\ Required\ to\ develop\ product = 14 \times 10{,}000 = 14{,}000\$$$

Intermediate COCOMO: Equations of effort and time of development is given below:

$$E = a_i\ (\text{KLOC})^{b_i} \times \text{EAF}$$

$$Tdev = c_i(E)d_i$$

where, KLOC = Kilo Line of Code

EAF = Effort Adjustment Factor which is change from 0.9 to 1.4 and a_i, b_i, c_i and d_i are given in following table. The cost depends on 14 cost drivers.

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Complete COCOMO or **detail COCOMO** is combination of subsystem. Assume that sub-system has database part which lies in organic type. Graphical user interface part lies in semidetached and communicational part is of embedded type.
There are five phases of detailed COCOMO which are

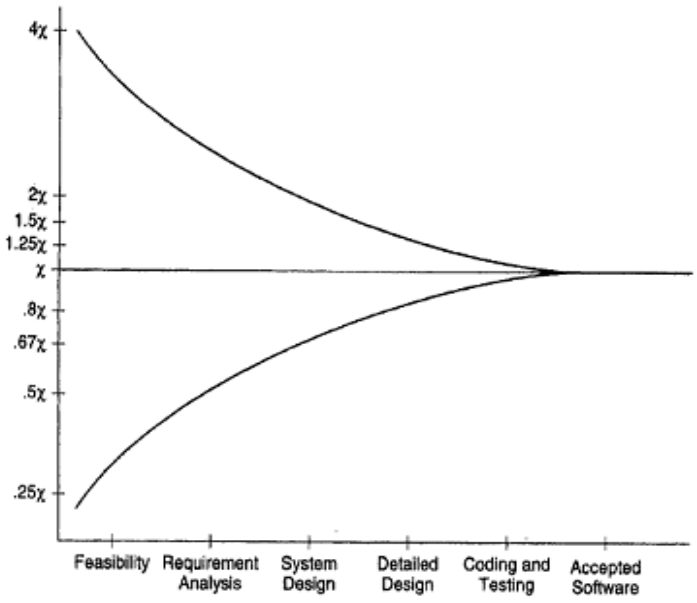
- plan and requirement.
- system design.
- detailed design.
- module code and test.
- integration and test.

There are two features of COCOMO model:

- COCOMO estimates are more objective and repeatable than estimates made by methods relying on proprietary models
- COCOMO can be calibrated to reflect your software development environment, and to produce more accurate estimates

10.5 CONCEPT OF EFFORT ESTIMATION AND UNCERTAINTY

- The accuracy of the estimate at any point depends on the amount of reliable information about the final product.
- The effort can be accurately determined when the product is delivered, as all the data about the project and the resources spent can be fully known by then.
- At the time when the project is being initiated, we have only some idea of the classes of data the system will get and produce and the major functionality of the system.
- The actual specifications of the system contain a great deal of uncertainty.
- Specifications with uncertainty represent a range of possible final products, not one precisely defined product.
- Estimates at this phase of the project can be off by as much as a factor of four from the actual final effort.



- The estimates can be made still more accurately when the design is complete.
- The figure above simply specifies the limitations of effort estimating strategies.
- Estimation models or procedures have to be developed for actual effort estimation.

- Estimation models have matured considerably despite the limitations and give fairly accurate estimates in general. For example, when the COCOMO model was checked with data from some projects, it was found that the estimates were within 20% of the actual effort 68% of the time.
- If the estimate is within 20% of the actual, the effect of this inaccuracy will not even be reflected in the final cost and schedule.

10.6 LET US SUM UP

We learnt in this chapter the concept LOC and estimation. Then we studied the Function Point metric for software estimation. The COCOMO model and its variants were also studied. At last we learnt about the concept of effort estimation and uncertainty.

10.7 REFERENCES AND SUGGESTED READING

(1) Software Engineering, Practitioner Approach, 7th Edition, by R.S. Pressman, Tata McGraw Hill, India, 2009.
(2) Integrated Approach to Software Engineering, Pankaj Jalote, Narosa Publications, 2003.
(3) Introduction to Software engineering by Rajiv Mall – PHI, 2000.
(4) Software Engineering by I. Somerville, 7th Edition, Pearson Education, India, 2006.
(5) Software Testing Techniques by B. Bezier PHI, India, 2001.

10.8 EXERCISE

1. Explain the concept LOC and estimation.
2. Describe Function Point.

3. What is the difference between product and process metrics?
4. What is COCOMO Model?
5. Explain the concept of effort estimation and uncertainty.

11

CONFIGURATION MANAGEMENT AND OO SOFTWARE MANAGEMENT

Unit Structure

11.0 OBJECTIVES

The objective of this chapter is

- To understand configuration management
- To understand process management
- To understand CMM and its levels
- To understand risk management and its activities

- To understand object oriented metrics
- To understand use case estimation
- To understand the concept of development tools
- To understand the basics of CASE

11.1 INTRODUCTION

A software program can be considered as configuration of software components. These software components are released in the form of executable code. Configuration management refers to the means by which the process of software development and maintenance is controlled.

One of the mechanisms for software management is object oriented approach which has several advantages over other approaches. In this approach information systems are viewed as collection of interacting objects that work together to accomplish tasks. The major advantage of using object oriented approach is its naturalness and reuse. In this chapter we will learn basics of object oriented approach.

11.2 CONFIGURATION MANAGEMENT PROCESS

The results of large software development effort typically consist of large number of objects e.g. source code, design document, SRS document, test document and user interface. These documents are usually modified in phases of SDLC. A new version of software is created when there is change in significant functionality usability etc. There are several reasons for configuration management. But, possibly the most important reason for configuration management to control the access the different deliverable objects. For configuration management we need to identify the configuration items first then we design control mechanisms for that item.

- Configuration identification involves deciding which part of system should be kept track of. In configuration identification objects associates with software development into three main categories controlled, pre-controlled, and un-controlled. Control objects are which are already put under the configuration control. Typical control object includes

requirement specification document, design document, tolls used to build system, such as compilers, linker lexical and module, test case and problem report.

- Configuration control ensures that changes to a system happen smoothly. It is the process of managing changes to controlled objects. It affects day to day life of developers. Configuration management tools allow only one person reverse module at any time. The engineer needing change a module first obtains a private copy of module through a reverse operation. Then, he carries out all necessary changes on his private copy.

The configuration management process can be divided into following set of activities

1) Identify configuration items, including customer supplied and purchase items, 2) Define a naming scheme for configuration items, 3) Define the directory structure needed for configuration management, 4) Define access restrictions, 5) Define change control procedures, 6) Identify and define responsibility of configuration personnel, 7) Identify points at procedure and re-configuration procedure , if needed, 9) Define a release procedure.

11.3 PROCESS MANAGEMENT

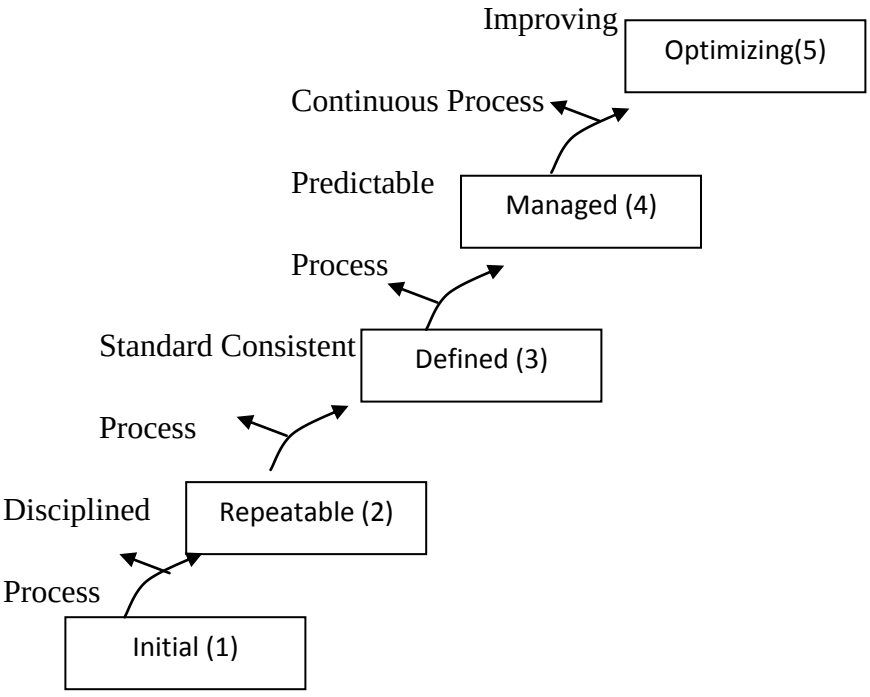
Software changes with respect time therefore, it is not static. It has to change to improve so that products produced using processes are of higher quality and less costly. Improving quality and productivity is key issues of software engineering. To achieve key issues software process must continuously be improved, as quality and productivity determined to a great extent by process. It should be emphasized that process management is quite different from project management. In process management we improve quality and productivity. Project management considers executing current project and assuring of objective met. Project management works with duration while process management works on much larger time scale as each project is viewed as providing data point for process. To improve software process organization of current status and development plan to improve process.

11.4 CMM AND ITS LEVELS

CMM is proposed by SEI (Software Engineering Institute) of Carnegie Mellon University, USA. It is basically

- The application of process management and quality improvement concepts to software development and maintenance.
- A guide for evolving toward a culture of engineering excellence.
- A model for organizational improvement.

CMM focuses on practices that are under control of the software group. It presents a minimum set of recommended practices that have been shown to enhance a software development and maintenance capability. There are five stages of CMM which is shown in the figure below:



Process name level is shown in box level are described below:

Level 1 – Initial: It is characterized by adhoc activities. In this very few ar ho process are defined. Therefore different engineer follow their own approach. Hence result of development; becomes chaotic (it is also known as chaotic level). In this development depend on individual hence when engineers leaves project management is not allowed, therefore shortcut are used, produced low quality software.

Level 2- Repeatable: In this tracking cost and schedule are established. Size and cost estimation techniques are used. Necessary process is in place to repeat earlier success on projects. With similar application, opportunity to repeat process exist only when company produce a family of products.

Level 3- Defined: In this level processes for both management and development are defined and documented. There are common roles, and responsibilities. The processes and product quality one not measured.

Level 4 – Managed: In this, we focus on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of product being developed. Process metric reflect the effectiveness of process being used. Product metrics are size, reliability time complexity, understandability, etc. process metrics are average defect correction time, number of failure detected for testing per LOC, etc. Qualitative quality goals are set for product.

Level 5 – Optimizing: In this process and product measurement data are analyzed for continuous process improvement. Continuous process improvement is achieved by carefully analyzing data and quantitative feedback form process. For this best software engineering practices and innovations are used.

Except level 1, each maturity level is characterized by key process Areas (KPAs) that indicate an organization should focus on for quality improvement. The KPA's of repeatable are software project planning & software configuration management. KPA's of defined are process definition, training program, and peer reviews. Managed has quantitative process metric and software quality management as KPA's. The KPA's of optimizing are defect prevention, process change management and technology change management. The components of people definition of process in level 3.

11.5 RISK MANAGEMENT AND ACTIVITIES

Risk is an unfavourable event or circumstance that can occur while a project is under way. Management of risk aims at reducing the impact of all kinds of risks. Three essential activities are risk identification, risk assessment and risk containment.

Risk Identification: To identify risk as soon as possible so the impact of the risks can be minimized by making effective risk management plans. So early risk identification is important issue. Some people may worry about vendors ability to complete their work on time as per company's standard. All risk should be identified and listed. A project gets affected by various risks. To identify important risks, classify risks indifferent classes.

There are three important classes of risks.

1. **Project risks** threaten the project plan. That is, if risks become real, it is likely that project schedule will slip and that cost will increase. Project risk identifies potential budgetary, schedule, personnel (staffing and organisation), resource customer, and requirement problem and their impact on software project. The size, project complexity and degree of structural uncertainty were also defined as project (and estimation) risk factors.

2. **Technical risks** threaten the quality and timeliness of software to be produced. If technical risk becomes reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problem. In addition, specification ambiguity, technical uncertainty, technical obsolescence and leading edge technology are also risks.

3. Business risks threaten the viability of software to be built. Candidates top five business risks are (1) building a product that no longer fits into overall business strategy for company (2) building a product that is no longer fit into overall business strategy, (3) building a product that sales if project is available. (4) Due to change in focus or change people known as management risk, (5) losing budgetary or personnel commitment (budget risks).

Another types of risks are proposed by Charette. Known risks are those that can be uncovered after careful evaluation of project plan. Business and technical environment in which project is being developed. Dictable risks are extrapolated from past project experience (e.g. staff turnover, poor communication with customer, dilution of staff effort as ongoing maintenance request are serviced). Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Risk item checklist is given below:

(i) **Product Size**: Risk associated with overall size of software built or modified.

(ii) **Business Impact**: Risk associated with constrained imposed by management or the market place

(iii) **Customer characteristics**: Risk associated with sophistication of customer and developer ability to communicate with customer in a timely manner.

(iv) **Technology to be built**: Risk associated with degree to which software process has been defined and followed by development organization.

(v) **Development Environment**: Risk associated with availability and quality of tools to used to build the product.

(vi) **Technology to be built**: Risk associated with complexity of system to be built and ??? of technology that is packaged by system

(vii) **Staff size and experience**: Risks associated with overall technical and project experience of software engineer who will do work.

Risk component and drivers are suggested in US Air Force pamphlet. US Air Force require engineer to identify risk driven affect software risk components. US Air Force focuses performance risk, cost risk, support risk and schedule risk.

11.6 OBJECT ORIENTED METRICS

The metrics for object oriented software are described below:

- **Lines of code**: It is the simplest metric and is measured in terms of number of lines of code written by the programmer.
- **Uncommented lines of code**: In this case instead of counting all the lines, the lines containing actual source codes are counted.
- **Percentage of lines with comments**: If a code has lots of informative comments, it is said to be more maintainable code. The desirable value of this metric is about 50 percent.
- **Number of classes**: This is good indicator of the object oriented design. It depends from programmer to programmer.
- **Number of methods per class**: A class is said to be too complex if it has very large number of methods.
- **Number of public methods per class**: As the name suggests, it denotes the number of public methods in a class and should be low.
- **Number of public instance variables per class**: In the ideal case it should be zero.
- **Number of parameters per method**: It should be low, may be zero or one.
- **Number of lines of code per method**: A more lines of code for a method is considered better.
- **Depth of the inheritance hierarchy**: It should be neither very complex nor zero. Complex hierarchies can be difficult to maintain while no inheritance limits the opportunities for reuse.
- **Number of overridden methods per class**: It should not be too high.

11.7 USE CASE ESTIMATION

The steps for use case based estimation are

1. Find use cases to the correct level of granularity.
2. Allocate points: For each transactions represented by use cases allocate points. The points allocation may vary between 5 to 15 depending on the complexity of the use case. Suppose there are four use cases and we assign 10 points to each of them then the total points will be 4 * 10 = 40 points.
3. Estimate effort: It depends upon the time to deliver a point. A standard value of this is 20 man hours for one point. Hence for 40 points it will be 40 * 20 = 800 man hours.

11.8 SELECTING DEVELOPMENT TOOLS

An analyst has to consider and select development tools. The development environment consists of

- Programming language(s)
- CASE (Computer Assisted Software Engineering) tool(s)
- and other softwares

Analysts can choose from various available choices of tools according to the requirement.

11.9 INTRODUCTION TO CASE

CASE stands for Computer Aided Software Engineering. It is basically a tool that provides automated assistance for software development, maintenance or project management activities. A CASE tool contains a database of information about the project, known as repository. The repository stores information about the system, including models, descriptions and references that link the various models together. The CASE tool can be used to verify the model and can also be used to check one model against the other. The advantages of CASE include increased productivity, restructuring of poorly written code, decrease of application development time and aid in project management. CASE tools may assist in

- Corporate planning of info systems
- Creating specification requirements
- Creating design specifications
- Code generation tools
- Information repository
- Development methodology

11.10 LET US SUM UP

In this chapter we learnt about configuration management process. Then we learnt about the process management. We then studied CMM and its levels. Risk management and its activities

were then discussed. We also learnt about the object oriented metrics and use case estimation. We also studied about development tools and its selection. Finally we learnt about the CASE tool and its application.

11.11 REFERENCES AND SUGGESTED READING

(1) Software Engineering, Practitioner Approach, 7th Edition, by R.S. Pressman, Tata McGraw Hill, India, 2009.

(2) Integrated Approach to Software Engineering, Pankaj Jalote, Narosa Publications, 2003.

(3) Introduction to Software engineering by Rajiv Mall – PHI, 2000.

(4) Software Engineering by I. Somerville, 7th Edition, Pearson Education, India, 2006.

(5) Software Testing Techniques by B.Bezier PHI, India, 2001.

(6) Stephen R Schach, Object Oriented and Classical Software Engineering , 5/e, TMH, 2010.

11.12 EXERCISE

1. Explain configuration management process.
2. Describe process management.
3. Explain CMM and its levels.
4. Explain Risk Management and Activities.
5. Describe the object oriented and web testing.
6. Explain the planning of software testing.
7. Explain the principles of static testing

12

CHANGING TRENDS IN SOFTWARE DEVELOPMENT

Unit Structure

12.0 OBJECTIVES

The objective of this chapter is

- To understand unified process
- To understand agile development
- To understand extreme programming
- To understand the concept of frameworks, components and services
- To understand the basics of design pattern

12.1 INTRODUCTION

The main goal of this chapter is to study new techniques that are advancing the way systems are being developed. We will focus here on two prime areas: the current trends in modeling and development process and some current tools and techniques that support these trends.

12.2 UNIFIED PROCESS – PHASES AND DISCIPLINES

The Unified Process (UP) tells how widely recognized is a standard system development methodology for object-oriented development, and many variations are in use. The original reason of UP defined an elaborate set of activities and deliverable for every step of the development and deliverables, simplifying the methodology.

UP phases: A phase in the UP can be thought of as a goal or major emphasis for a particular portion of the project. The four phases of the UP life cycle are named inception, elaboration, construction, and transition.

- **Inception**: Develop an approximate vision of the system make the business case, define the scope, and produce rough estimates for cost and schedule.
- **Elaboration**: Refine the vision, identify and describe all requirements, finalize the scope, design and implement the care architecture and function revolve high risks, and produce realistic estimate for cost and schedule.
- **Construction**: Iteratively implement the remaining lower risk, predictable, and easier elements and prepare for deployment.
- **Transition**: Complete the beta test and deployment and the system is made ready for operation.

UP Disciplines: A discipline is a set of functionally related activities that together contribute to one aspect of the development project. UP disciplines include business modeling, requirements, design, implementation, testing, deployment, configuration and change management, project management and environment. Each iteration usually involves activities from all disciplines.

During the inception phase iteration, the project manager might complete a model showing some aspect of the system environment (the business modeling discipline). The scope of the system is delineated by defining many of the key requirements of the system and listing use cases (the requirement discipline). To prove technological feasibility, some technical aspect of the system might be designed (the design discipline), programmed (the implementation discipline), and tested to make sure it will work as planned (the testing discipline). In addition, the project manager makes plan for handling changes to the project (the configuration and change management discipline), working on a schedule and cost/benefit analysis (the project management discipline), and tailoring the UP phases, iterations, deliverables, and tools to match the needs of the project (the environment discipline).

The elaboration phase includes several iterations. In the first iteration, the team works on the details of the domain classes and use cases addressed in the iteration (the business modeling and requirement disciplines). At the same time they might complete the description of all use cases to finalize the scope (the requirement discipline). The use cases addressed in the iteration are designed by creating design class diagrams and interaction diagrams (the design discipline), programmed using Java or Visual Basic. NET (the implementation discipline), and fully tested (the testing discipline). The project manager works on the plan for the next iteration (the project management discipline), and all team members continue to receive training on the UP activities they are completing and the system development tools they are using (the environment discipline)

12.3 AGILE DEVELOPMENT – PRINCIPLES AND PRACTICES

The main goal of Agile Software Development is to identify four basic values, which represent the core philosophy of the agile movement. The four values emphasize as follows:

- Responding to change over following a plan
- Individuals and interaction over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation.

Agile Modeling Principles: Agile Modeling (AM) is not about doing less modeling but about doing the right kind of modeling at the right level of detail for the right purposes. We can identify two primary reasons to built models: (1) to understand what you are building and (2) to communicate important aspects of the solution system.

Agile Modeling Practices: The following practices support the AM principles just expressed. The heart of AM is its practices, which give the practitioner specific modeling technique.

- **Iterative and Incremental Modeling:** Modeling is a support activity, not the end result of software development. As a developer, we should create small models frequency to help you understand or solve a problem. New developers sometimes have difficulty deciding which models to select. We should continue to learn about models and expand your repertoire.
- **Teamwork:** AM supports various development methodologies. One of the tenants in all of these methodologies is that developers work together in small teams of two to four members. In addition, users should be integrally involved in modeling exercises.
- **Simplicity:** During modeling sessions, the team can begin to write code for the solutions already conceived so that they can validate the models. Simplicity support frequent validation. Too many or complex models are not created until the simple ones have been validated with code.
- **Documentation:** Several models are temporary working documents that are developed to solve a particular problem. These models quickly become obsolete as the code evolves and improves. It is discarded if they were posted to a repository; it should be dated so that everyone can keep track of it.
- **Motivation:** The basic objective of modeling is to build a model if it helps you understand a process or solve a problem or if you understand a process or solve a problem or if you need to record and communicate something. For example the team members in a design session might make, some design decisions. To communicate these decisions, the team posts an simple model to make it public. The model can be a very effective tool to document the decisions and ensure that all have a common understanding and reference point.

12.4 EXTREME PROGRAMMING – CORE VALUES AND PRACTICES

Extreme Programming (XP) is an adaptive, agile development methodology that was created in the mid-1990s. It is an attempt to take the best practices of software development and extend them "to the extreme." Extreme programming – task proven industry best practices and focuses on them intensely.

XP Core Values: The four core value of XP – communications, simplicity, feedback, and courage – drive its practices and project activities.

- **Communication:** One of the major causes of project failure has been a fact of open communication with the right players at the right time and at the right level. Effective communication involves not only documentation but also open verbal discussion. The practices and methods of XP are designed to ensure that open, frequent communication takes place.
- **Simplicity:** Developers have always advocated keeping solutions simple, they do not always follow their own advice. XP includes techniques to reinforce this principle and make it a standard way of developing systems.
- **Feedback:** Getting frequent and meaningful feedback is recognized as a best practice of software development. Feedback on functionality and requirements should come from the users, feedback on designs and code should come from other developers, and feedback on satisfying a business need should come from the client XP integrates feedback into every aspect of development.
- **Courage:** Developers always need courage to face the harsh choice of doing things right of throwing away bad code and starting over. But all too frequently they have not had the courage to stand up to a too right schedule, resulting in bad mistakes. XP practices are designed to make it easier to give developers the courage to "do it right."

XP Practices: XP's 11 practices embody the basic values just presented.

- **Planning:** The planning practice focuses on making a rough plan quickly and then refining it as things become clearer.

- **Testing:** XP intensifies testing by requiring that the tests for each story be written first-before the solution is programmed. There are two major types of tests, unit tests which test the correctness of a small piece of code, and acceptance tests, which test the business function.
- **Pair Programming:** Pair programming divides up the coding work. First, one programmer might focus more on design and double-checking the algorithm while the other writes the code. Then they switch roles so that both think about design, coding, and testing.
- **Simple Design:** XP considers that design should be done continually but is small chunks. As with everything else, the design must be verified immediately by reviewing if along with coding and testing.
- **Code Refactoring:** It is the technique of improving the code without changing its function. XP programmers continually refactor their code. Refactoring produces high-quality robust code.
- **Owning the Code Collectively:** In XP everyone is responsible for the code. Collective ownership allows anyone to modify any piece of code.
- **Continuous Integration:** This practice embodies XP's idea or "growing" the software. Small pieces of code - which have passed the unit tests – are integrated in to the system daily and even more often. Continuous integration highlights errors rapidly and keeps the project moving ahead.
- **On-Site Customer:** XP project requires continual involvement of users who can make business decisions about functionality and scope. If the customer is not ready to commit resource to the project, the project will not be very successful.
- **System Metaphor:** This practice is XP's unique and interesting approach to defining an architectural vision. It answers the questions, "How does the system work? What are its major components?" To answer these questions, developers identify a metaphor for the system.
- **Small Releases:** Consistent with the entire philosophy of growing the software, small and frequent releases provide upgraded solution to the users and keep them involved in the project. They also facilitates, such as immediate feedback and continual integration.

- **Forty-Hour Week and Coding Standards:** These final two practices set the tone fur hours the developers should work. The exact number of hours a developer works is not the issue. The issue is that the project should not be a death march that burns out every number of the team. Neither is the project a haphazard coding exercise. Developers should follow standards for coding and documentation. XP was just the engineering principles that are appropriate for an adaptive process based on empirical controls.

12.5 FRAMEWORKS, COMPONENTS, SERVICES

Reusing software to implement such common functions is a decades old development practice. But such reuse was awkward can cumbersome with older programming languages and before ubiquitous networks. Object orientation includes two powerful techniques, frameworks and components that support software reuse.

Object Frameworks: An object framework is a set of classes that are specifically designed to be reused in a wide variety of programs. The object framework is supplied to a developer as a precompiled library or as program source code that can be included or modified in new programs. The classes within an object framework are sometimes called foundation classes.

Object frameworks have been developed for a variety of programming needs. Examples include the following:

- **User-interface Classes**: Classes for commonly used objects within a graphical user interface, such as windows, menus, toolbars, and file open and save dialog boxes.
- **Generic Data Structure Classes**: Classes for commonly used data structure such as linked lists, indicates, and binary trees and related processing operations such as searching, sorting and inserting and deleting elements.
- **Rational Data Base Interface Classes**: Classes that allow programs to create database tables, add data to a table, delete data from a table, or query the data content of one or more table.
- **Classes Specific to an Application Area**: Classes specifically designed for use in application areas such as bunking, payroll, inventory control, and shipping.

The process of developing a system using one or more object frameworks is essentially one of adaptation. The frameworks supply a template for program construction and a set of classes that provide generic capabilities. Systems designers adapt the generic classes to the specific requirements of the new system.

Components: A component is a software module that is fully assembled and tested is ready to use, and has well-defined interfaces to convert it to clients or other components. Component can be single executable objects or groups of interacting objects. A component can also be a non-00 program or system "wrapped" in non-00 interface. Components implemented with non-00 technologies must still implement object like behavior. In other words, they must implement a public interface, respond to messages and hide their implementation details.

Components are executable objects that advertise a public interface that is, a set of methods and messages and hide (encapsulate) the implementation of their methods from other components.

Components provide an inherently flexible approach to systems design and construction. Developers can design and construct many parts of a new system simply by acquiring and plugging in an appropriate set of components. They can also make newly developed functions, programs and systems, and systems more flexible by designing and implementing them as collection of components. Component based design and construction have been the norm in the manufacturing of physical goods (such as cars, TV, and computer hardware) for decades. However, it has only recently become available approach to designing and implementing information systems.

Services: An application interacts with a service via the Internet or a private network during execution. Service standards have evolved from distributed object standards such as CORBA and EJBs to include standards such as SOAP, .NET and J2WS.

- **Simple Object Access Protocol (SOAP)**: This service standard is based on existing Internet protocols including HTTP and XML.
- **Microsoft .NET**: It is a service standard based on SOAP. The .NET applications and services communicate using SOAP protocols and XML messages.

- **Java 2 Web Services (J2WS)**: is a service standard for implementing applications and services in Java. It extends SOAP and several other standards to define Java-specific method of implementing communication among applications and servers.

12.6 INTRODUCTION TO DESIGN PATTERNS

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a "good" design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and iteration diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern, etc.

- **Expert Pattern:** The expert pattern expresses the common intuition that objects do things related to the information they have.
- **Creator Pattern:** The class that is be responsible for creating new instance of class C2, if one or more of the following are true:

  C1 is an aggregation of object of type C2

  C1 contains object of type C2

  C1 closely uses objects of type C2

  C1 has the data that would be required to initialize the objects of type C2, when they are created.

- **Controller Pattern:** For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out of sequence actor requests e.g. whether voucher request is received before arrange payment request.

- **Façade Pattern:** Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS. A class (such as DB façade) can be created which provides a common interface to the services of the package.

12.7 LET US SUM UP

We saw in this chapter the phases and disciplines of unified process. Then we learnt about the agile development in terms of its phases and disciplines. The core values and practices of extreme programming were then studied. We also learnt about framework, components and services. Finally we tried to understand the concept of design patterns.

12.8 REFERENCES AND SUGGESTED READING

(1) System Analysis & Design – Satzinger, Jackson,Burd,Cengage Learning, India Ed.
(2) Software Engineering- A Practitioner's Approach, 7th Edition, McGraw Hill Int.
(3) Integrated Approach to Software Engineering - Pankaj Jalote (Narosa), 3rd Edition
(4) Design Patterns – Elements of Reusable Object-Oriented Software, Pearson – Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

12.9 EXERCISE

1. Explain USE CASE estimation.

2. Explain unified process, its phases and disciplines.

3. Describe Agile Development. Discuss principles & practices.

4. Explain Extreme Programming. Describe core values and practices.

5. Explain framework and its components and services.

13

SOFTWARE TESTING – FUNDAMENTALS AND PLANNING

Unit Structure

13.0 OBJECTIVES

The objective of this chapter is

- To understand the basics quality assurance
- To understand six sigma

- To understand testing and related terms
- To understand objective, challenges and principles of testing
- To understand the concept of test plan
- To understand test plan specification
- To understand test case execution and analysis
- To understand defect logging and tracking

13.1 INTRODUCTION

The aim of the software development program is to develop software that has no error or very few errors. The effort to detect errors is less if it is detected as soon as they are introduced. At the end of verification activity such as review errors should be detected. However, these verification activities in the early phases of software development are based on human evaluation and cannot detect all the errors. This unreliability of the quality assurance activities in the early part of the development cycle places a very high responsibility on testing. In the other words, as testing is the last activity before the final software is delivered. It has the enormous responsibility of detecting any type of errors that may be in the software.

Furthermore, we know that software typically undergoes change even after it has been delivered. And to validate that a change has not affected some old functionality of the system, regression testing is done. In regression testing, old test cases are executed with the expectation that the some old results will be produced need for regression testing places additional requirements on the testing phase; it must provide the "old" test cases and their outputs.

Levels of Testing: Testing usually depends upon the detection of the faults remaining from earlier stages, in addition to the faults introduced during coding itself. Due to this, different levels of testing are used in the testing, process; each level of testing aims to test different aspects of the system.

The basic levels are unit testing, integration testing, and system and acceptance testing. These different levels of testing attempt to detect different types of faults.

The foremost level of testing is unit testing. Different modules are tested against the specifications produced during design for the modules in this level. Unit testing is essentially for verification of the code produced during the coding phase, and hence the goal is to test the internal logic of the module and is considered for integration and use by other only after it has been unit tested satisfactorily.

13.2 INTRODUCTION TO SOFTWARE QUALITY ASSURANCE (SQA)

The main objective of SQA is to help an organization develop high quality software products in a repeatable manner. A software development organization is the one where the software development process is person independent. In a non repeatable software development organization, a software product development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm as played by certain individuals. Thus, in a non repeatable software development organization, the chances of successful completion of a software project are to a great extent depends on the team members. For this reason, the successful development of one product by such an organization does not automatically imply that the next product development will be successful. In this context, we will see that one of the primary objectives of qualities assurance is repeatable software development. Besides, the quality of the developed software and the cost of development are important considerations.

13.3 SIX SIGMA

The purpose of Six Sigma is to improve process in order to do things better, tastes, and at lower cost. It can be used to improve every facet of business, from production to human resources, to order entry, for technical support. Six Sigma can be used for any activity that is concerned with cost, timeliness and quality of results. Therefore, it is applicable for risk to every industry.

For many organizations Six Sigma simply mean striking for perfection. It is a discipline, data-driven approach to eliminate defects in many processes from manufacturing to transactional and from product to service. The statistical representation of Six Sigma describes quantitatively

how a process is performing. To achieve Six Sigma, a process must not produce more than 3-4 defects per million opportunities. A Six Sigma defect is defined as any system behavior that is not as per customer specifications. The total number of Six Sigma opportunities is then the total number of chances for a defect. Process Sigma can easily be calculated using a Six Sigma calculator.

The fundamental objective of Six Sigma methodology is the implementation of a measurement based strategy that focuses on process improvement and variation reduction through the application Six Sigma improvement projects.

The next level of testing is often called integration testing. Unit tested modules are combined into subsystems, which are then tested at this level. The purpose here is to see if the modules can be integrated properly. Hence, the emphasis is on testing interfaces between modules. This testing activity can be considered testing the design.

System testing and acceptance testing comes at the next level. The entire software system is tested in this level. The requirements document acts as the reference document for this purpose, and the aim is to see if the software meets its requirements. This is essentially a testing is sometimes performed with realistic data of the client to demonstrate that the software is working satisfactorily. Testing here focuses on the external behaviors of the system; the internal logic of the program is not emphasized. Consequently, mostly functional testing is performed at these levels.

13.4 TESTING FUNDAMENTALS AND COMMON TERMS

The aim of the testing process is to identify all defects existing in a software product. However, for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, we should not underestimate the importance of testing.

- **Error**: The difference between expected result and actual result is error.
- **Bug**: If that error comes at the development stage before production then it is bug.
- **Defect**: If that error comes after production then we say it is defect.
- **Fault**: It can be defined as the deviation from the requirements either implied or stated. It is the condition that can cause failure of the system.
- **Failure**: It refers to the inability of the system to meet the requirements in terms of performance or outcome. It occurs whenever there is an attempt to execute a fault. A failure is a manifestation of an error (or defect) but the mere presence of a error may not necessarily lead to a failure. A test case is the triplet [I, S, O], where I is the data input to the system, S is the state of the system of which the data is input, and O is the expected output of the system.
- A **test suite** is the set of all test cases with which a given software product is to be tested.
- **Verification and Validation:** Verification determines whether the output of the present phase of software development conforms to that of its previous phase, whereas validation determines whether a fully developed system conforms to its requirements specification. Verification is thus concerned with phase containment of errors and the aim of validation is that the final product be error free.

13.5 OBJECTIVES OF TESTING

The main objectives of testing are

- To confirm that the system meets the requirements
- To find all the bugs and defects in the software product
- To verify the product i.e., to check whether the product is developed appropriately
- To increase user's confidence in the product.

13.6 CHALLENGES IN TESTING

The fundamental challenges in the testing process are:

- It is almost impossible to design a test that can completely test software.
- Testers often misallocate resources because they fall for the company's process myths.
- Test groups often involve in conflict situation when they operate under multiple missions.
- Test groups often lack skilled programmers and a vision of appropriate projects that would keep programming testers challenged.

13.7 PRINCIPLES OF TESTING

Testing principles suggest general rules common for all testing which aids us in performing testing effectively and efficiently. Principles for software testing are:

- Test a program to make it fail: It means we should try to make the software program to fail by planning test plans in different ways.
- Start testing earlier: The testing process should run parallel with its development.
- Testing should be context dependent: It should be different for different types of product.
- Test plan should be defined.
- Design of test plans should be effective, means it should be able to check the product effectively.
- Testing design should be made to check valid as well as invalid inputs.
- Test cases should be regularly reviewed.

13.8 TEST PLAN

In general, testing process begins with a test plan and concludes with acceptance testing. A test plan is a document for the entire project that defines the scope, approach to be taken, and the schedule at testing. It also identifies the test items for the entire testing process and the personnel responsible for the different activities of testing. The test planning can be done well before the actual testing commences and can be done in parallel with the coding the design activities. The inputs for forming the test plan are:

- Project plan: It is needed to make sure that the test plan is consistent with the overall quality plan for the project and the testing schedule matches with that of the project plan.
- Requirements document
- System design document

The requirements document and the design document are the basic documents used for selecting the test units and deciding the approaches to be used during testing.

A test plan should contain the following:

- Test unit specification
- Features to be tested

Approach for Testing, Test Deliverables, Schedule and Task Allocation: One of the most important activities of test plan is to identify the test units. A test unit is a set of one or more modules, together with associated data, that are from a single computer program and that are the object of testing. A test unit can occur at any level and can contain from a single module to the entire system. Thus, a test unit may be a module, a few modules, or a complete system.

13.9 TEST PLAN SPECIFICATION

The test plan aims at

- Progress of testing process for the project
- Units that will be tested
- Approaches to be used during the various stages of testing.

It does not deal with the details of testing a unit, also it does not specify which test cases are to be used.

Each unit should have its own test case specification. For this purpose first the feature of the unit to be tested is identified on the basis of approach specified for the test plan. The overall approach stated in the plan is refined in to specific test techniques that should be followed and

into the criteria to be used for evaluation. On the basis of this, the test cases are specified for testing the unit.

13.10 TEST CASE EXECUTION AND ANALYSIS

Test case specification defines the test cases for the unit to be tested while execution of test cases may require driver modules or stubs.

Test case execution involves

- Creation of test stubs or test drivers.
- Setting up test environment.
- Data collection or generation.

The outputs generated from test execution entails

- Test log (test case specification document itself can be used as test log)
- Test summary report gives summary of test case execution, e.g. no. of test cases, no. of errors, effort metric (if any)
- Error report lists and categorizes errors. It is also used for tracking the defects.

After testing is complete, efficiency of various defect removal techniques can be compared. Testing effort is another good measure for judging the effectiveness of testing.

13.11 DEFECT LOGGING AND TRACKING

A software project can have thousands of defects which are identified at different stages by different people. The person who fixes the defect may be not the same one who detected it. In such a scenario, defect reporting and closing cannot be done informally. The use of informal mechanisms may lead to defects being found but later forgotten, resulting in defects not getting removed or in extra effort in finding the defect again. Hence, defects found must be properly logged in a system and their closure tracked. Defect logging and tracking is considered one of the best practices for managing a project, and is followed by most software organization.

Defect Analysis and Prevention: We have seen that defects are introduced during development and are removed by the various quality control tasks in the process. Whereas the

focus of the quality control tasks it to identify and remove the defects, the aim of defect prevention is to learn from defects found so far on the project and prevent defects from getting injected in the rest of the project. Some forms of defects prevention are naturally practiced and in a sense the foal of all standards, methodologies, and rules is basically to prevent defects. However, when actual data is available, more effective defect prevention is possible through defect prevention is possible through defect data analysis.

13.12 LET US SUM UP

We first learnt about quality assurance. Then we saw six sigma technology for software development. We then studied about testing fundamentals and learnt some common terms related to testing. Further we discussed the main objectives of testing. Also we saw the challenges faced in testing process. We also described the principle of testing. We learnt about the basics of test plan. Then we saw the specification of test plan. Test case execution and its analysis were then discussed. Finally we studied about defect logging and tracking.

13.13 REFERENCES AND SUGGESTED READING

(1) Software Engineering, Practitioner Approach, 7th Edition, by R.S. Pressman, Tata McGraw Hill, India, 2009.

(2) Integrated Approach to Software Engineering by, Pankaj Jalote, Narosa Publications, 2003.

(3) Introduction to Software engineering by Rajiv Mall – PHI, 2000.

(4) Software Engineering by I. Somerville, 7th Edition, Pearson Education, India, 2006.

(5) Software Testing Techniques by B. Bezier PHI, India, 2001.

13.14 EXERCISE

1. What do we mean by quality assurance?
2. Explain Six Sigma technique.
3. Explain the terms fault, failure, defect, bug and error.
4. What are the objectives of testing?

5. What are the challenges that we face in testing process?

6. Discuss the principles of testing.

7. Explain Planning of testing.

8. What is Defect Logging and Tracking?

9. Explain Defect Analysis and Prevention.

10. Describe Test Case Specifications.

.

14

SOFTWARE TESTING TYPES I

Unit Structure

14.0 OBJECTIVES

The objective of this chapter is

- To understand different levels of testing
- To understand types of testing like black box and white box testing
- To understand function and performance testing
- To understand testing like regression, volume, stress, alpha and beta testing
- To understand static testing
- To understand the basic concept of object oriented testing
- To understand the basics of website testing

14.1 INTRODUCTION

The aim of software testing is to measure the quality of software in terms of number of defects found in the software, the number of tests run and the system covered by the tests. When bugs or defects are found with the help of testing, the bug is logged and the developers team fixes the bug.

We will discuss various types of testing in this chapter. The levels of testing and other types of testing like black box, white box, performance, regression, volume, stress , alpha and beta testing.

14.2 LEVELS OF TESTING

14.2.1 Unit Testing: This is the process of taking a module and run it in separation from the rest of the software product by using prepared test cases and comparing actual result with expected output. The small size of the module makes it easy to locate errors.

The most common approach to unit testing requires drivers and stubs to be written. The driver simulates a calling unit and the stub simulates a called unit. The investment of developer time in this activity sometimes results in demoting unit testing to a lower level of priority and that is almost always a mistake. Even though the drivers and stubs cost time and money, unit testing provides some undeniable advantages. It allows for automation of the testing process, reduces difficulties of discovering errors contained in more complex pieces

of the application, and test coverage is often enhanced because attention is given to each unit.

For example, if you have two units and decide it would be more cost effective to glue them together and initially test them as an integrated unit, an error could occur in a variety of places:

- Is the error due to a defect in unit 1?
- Is the error due to a defect in unit 2?
- Is the error due to defects in both units?
- Is the error due to a defect in the interface between the units?
- Is the error due to a defect in the test?

Finding the error (or errors) in the integrated module is much more complicated than first isolating the units, testing each, then integrating them and testing the whole.

14.2.2 Integration Testing: The objective of integration testing is the interface: whether parameters match on both sides as to type, permissible ranges, meaning and utilization. In other words the goal is to take unit tested modules and test the overall software structure that has been dictated by design. There are three common strategies to perform integration testing:

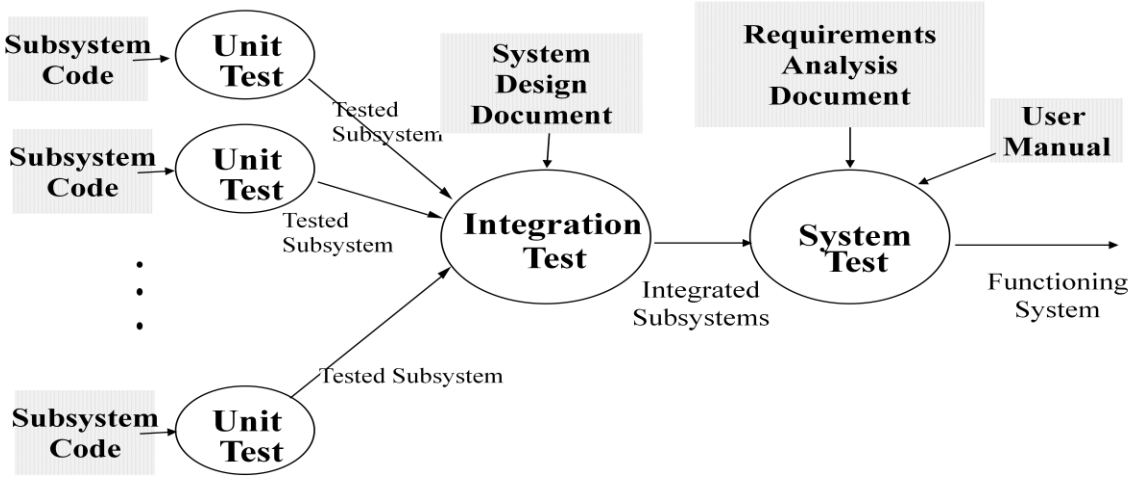
- The top-down approach to integration testing requires the highest-level modules be test and integrated first. This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers. However, the need for stubs complicates test management and low-level utilities are tested relatively late in the development cycle. Another disadvantage of top-down integration testing is its poor support for early release of limited functionality.
- The bottom-up approach requires the lowest-level units be tested and integrated first. These units are frequently referred to as utility modules. By using this approach, utility modules are tested early in the development process and the need for stubs is minimized. The downside, however, is that the need for drivers complicates test management and high-level logic and data flow are tested late. Like the top-down

approach, the bottom-up approach also provides poor support for early release of limited functionality.

- The third approach, sometimes referred to as the umbrella approach, requires testing along functional data and control-flow paths. First, the inputs for functions are integrated in the bottom-up pattern discussed above. The outputs for each function are then integrated in the top-down manner. The primary advantage of this approach is the degree of support for early release of limited functionality. It also helps minimize the need for stubs and drivers. The potential weaknesses of this approach are significant, however, in that it can be less systematic than the other two approaches, leading to the need for more regression testing.

14.2.3 System Testing: System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System tests are designed to validate a fully developed system to assure that it meets its requirements. It is of three types: alpha testing, beta testing and acceptance testing. We will discuss these tests later in this chapter.

Following diagram illustrates the three levels of testing



14.2.4 Validation Testing: These tests to determine whether an implemented system fulfills its requirements.

14.2.5 Acceptance Testing: Acceptance testing is performed by users or on behalf of the users to ensure that the software functions in accordance with the software requirement specification. It focuses on the following aspects

- All functional requirements are satisfied
- All performance requirements are achieved
- Other requirements like transportability, compatibility, error recovery etc. are satisfied
- Acceptance criteria specified by the user is met

14.3 FUNCTION TESTING AND PERFORMANCE TESTING

Function testing: A function test checks that the integrated system performs its function as specified in the requirement. It is accomplished by a test team independent of the designers and programmers. Bothe valid and invalid inputs are tested. Functional testing typically involves five steps:

- The identification of functions that the software is expected to perform
- The creation of input data based on the function's specifications
- The determination of output based on the function's specifications
- The execution of the test case
- The comparison of actual and expected outputs

Performance testing: Performance testing is carried out to check whether the system meets the non functional requirements identified in the SRS document. There are several types of performance testing and depends on the different non functional requirements of the system documented in the SRS document. All performance tests can be considered as black-box tests.

14.4 REGRESSION TESTING, VOLUME TESTING AND STRESS TESTING

Regression Testing: Regression testing is basically a separator dimension to the three testing methods - unit testing, integration testing, and system testing. It is the practice of running an old test suite after each change to the system or after each bug fix. It is ensured in this testing that no new bug has been introduced as a result of this change made or bug fixed. However, if only a few statements are changed, then the entire test suite need not be run-only those test cases that test the functions which are likely to be affected by the change need to be run.

Volume Testing: It has to be ensured that the data structures like arrays, queues, stacks, etc. have been designed successfully for unexpected situations. For example, a compiler might be tested to check whether the symbol table over flow, when a very large program is compiled.

Stress Testing: It is also known as endurance testing and basically evaluates system performance when it is stressed for short periods of time. Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity. For example, suppose an operating system is supposed to support 15 multi programmed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real time system might be tested to determine to effect of simultaneous arrival of high priority interrupts.

14.5 ALPHA AND BETA TESTING

Alpha testing: Alpha testing refers to the system testing carried out by the test team within the developing organization. A virtual user environment can be created for this type of testing. Testing is done at the end of development. Still minor design changes may be made as a result of such testing.

Beta testing: Beta test is the final test that is performed before releasing application for commercial purpose. It is basically a system test performed by a selected group of friendly customers.

14.6 ROBUSTNESS TESTING AND MUTATION TESTING

Robustness testing: Robustness is the degree to which a system can operate properly in the presence of unexpected inputs or stressful environmental conditions. The aim of robustness testing is to plan test cases and test environments where the system's robustness can be checked. It is basically a quality assurance technique focused on testing the robustness of software. It is also used to describe the process of verifying the robustness of test cases in a test process.

Mutation testing: Mutation Testing involves running slightly corrupted versions of your target program through your test suite to see if any test cases flag the variations as defects. Mutation testing involves deliberately altering a program's code, then re-running a suite of valid unit tests against the mutated program. A good unit test will detect the change in the program and fail accordingly.

14.7 STATIC TESTING

Static testing is a type of testing that does not entail execution of the application to be tested.

- It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily syntax checking of the code or and manually reading of the code or document to find errors
- This type of testing can be used by the developer who wrote the code, in isolation. Code reviews, inspections and walkthroughs are also used.
- This is the verification portion of Verification and Validation.
- There are three main types of static testing that are performed.
    - Desk checking

- Code walkthrough
- Code inspection

While desk-checking is performed by the author of the code who reviews his/her portion of code, the other two techniques of walkthrough and inspection involve a group of people apart from the author of the code performing the review.

14.8 OBJECT ORIENTED TESTING STRATEGIES

In general testing computer software begins with "testing in the small" and works outward to inward to "testing in the large." We start with unit testing, then progress toward integration testing and culminate with validation and system testing. In conventional applications, unit testing focuses on the smallest compliable program unit the subprogram (e.g. module, subroutine, procedure, and component). Once each of these units has been tested individually, it is integrated into a program structure which a series at regression tests are run to uncover errors due to interfacing between the modules and side effects caused by the addition of new units. Finally, the system as a whole is tested to insure that errors in requirements are uncovered.

- **Unit Testing in the 00 Context:** In the case of object oriented software, the concept of the unit changes. The definition of classes and objects is driven by encapsulation, this means that each class and each instance of a class (object) packages attribute (data) and the operations (also known as methods or services) that manipulate these data rather than testing an individual module. Since, a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.
- **Integration Testing in the 00 Context:** Object oriented software does not have a hierarchy of control structure, hence conventional top-down and bottom-up integration strategies have little meaning.
- **Validation Testing in an 00 Context:** The details of class connection disappear at the validation or system level. Similar to conventional validation, the validation of 00 software focuses on user visible actions and user-recognizable output of the

system. To assist in the deviation of validation tests, the tester should draw upon the use cases that are part of the analysis model. The use case provides scenario that has a high like hood of uncovered errors in user interaction requirements.

14.9 OVERVIEW OF WEBSITE TESTING

Web-based systems and applications reside on a network and interoperate with many different operating systems, browsers, hardware platforms, and communications protocols. Finding an error on web systems is a significant challenge for Web engineers

The following steps are followed for website testing:

- The **content model** for the Web application (WebApp) is reviewed to uncover errors. This "testing" activity is similar in many respects to copy editing in written document. In fact, a large web-site might enlist the services of a professional copy editor to uncover typographical errors, grammatical mistakes, errors in content consistency, errors in graphical representations and cross-referencing errors.
- To uncover navigation errors, **the design model** for the WebApp is reviewed. Use cases, derived as part of the analysis activity, allow a web engineer to exercise each usage scenario against the architectural and navigation design. In essence, these non executable tests help to uncover errors in navigation.
- Selected processing components and web pages are unit tested. In the case of WebApps, the concept of the unit changes. Each Web page encapsulates content, navigation links, and processing elements like forms, scripts, applets, etc. It is not always possible or practical to test each of these characteristics individually. In many cases, the smallest testable unit is the Web Page.
- The architecture is constructed and integration tests are conducted. The strategy for integration testing depends on the architecture that has been chosen for the WebApp. Integration testing in this case is similar to the approach used for 00 systems.
- The assembled WebApp is tested for overall functionally and content delivery. Similar to conventional validation, the validation test of Web-based systems and

applications focuses on user-visible actions and user-recognizable output from the system.

- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration. A cross-reference matrix that defines all problems of operating systems, browsers, hardware platforms, and communications protocols is created. Tests are then conducted to uncover errors associated with each possible configuration.
- The WebApp is tested by a controlled and monitored population of end-users. A population of users that encompasses every possible user role is chosen. The WebApp is exercised by these users and the results of their interaction with the system are evaluated for content and navigation errors usability concerns, compatibility concerns, and WebApp reliability and performance.

14.10 LET US SUM UP

In this chapter we learnt level of testing which involves unit testing, integration testing, system testing , validation and acceptance testing. Then we saw black box and white box testing. We then understood the concept of functional and performance testing. Regression testing , volume and stress testing were then discussed. Further alpha and beta testing was explained. Robustness and mutation testing were also illustrated. Finally overview of object oriented software testing and website testing was studied.

14.11 REFERENCES AND SUGGESTED READING

(1) Software Engineering, Practitioner Approach, 7th Edition, by R.S. Pressman, Tata McGraw Hill, India, 2009.

(2) Integrated Approach to Software Engineering by, Pakaj Jalote, Narosa Publications, 2003.

(3) Introduction to Software engineering by Rajiv Mall – PHI, 2000.

(4) Software Engineering by I. Somerville, 7th Edition, Pearson Education, India, 2006.

(5) Software Testing Techniques by B. Bezier PHI, India, 2001.

14.12 EXERCISE

1. Explain unit, integration and system testing.

2. Explain validation and system testing

3. Explain white-box, performance and regression typing.

4. Explain volume and stress testing.

5 Describe alpha and beta testing.

6. Explain static testing in detail.

7. Describe the object oriented testing techniques

8. Explain web testing.

15

SOFTWARE TESTING TYPES II

Unit Structure

15.0 OBJECTIVES

The objective of this chapter is

- To understand black box testing
- To understand white box testing

15.1 INTRODUCTION

Black box testing: In this testing method we look at what are the available inputs for a software application and what the expected outputs for each input.

- It is not concerned with the inner structure of the software, the process that the software undertakes to achieve a particular output or any other internal aspect of the software that may be involved in the transformation of an input into an output.

- Most black-box testing tools employ either coordinate based interaction with the software graphical user interface (GUI) or image recognition.

White box testing: This testing technique looks under the covers and into the subsystem of the software.

- Unlike black-box testing which concerns itself exclusively with the inputs and outputs of the software, whitebox testing enables you to see what is happening inside the software.
- Whitebox testing provides a degree of sophistication that is not available with black-box testing as the tester is able to refer to and interact with the objects that comprise an application rather than only having access to the user interface.

Let us now discuss both these tests in detail.

15.2 BLACK BOX TESTING

In black-box testing, no knowledge of design or code is required and test cases are constructed by examining the input and output values only. The following are the two main approaches to designing black-box test cases.

- **Equivalence Class Partitioning:** In this approach, a set of equivalence classes if formed by partitioning the domain of input values of a program. The partition is done such that the behavior of the program is similar to every input data belonging to the same equivalence class. The main plan behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is good testing the software with any other value belonging to that equivalence class. Equivalence classes for the software can be designed by framing both the input data. The following some general guidelines for designing the equivalence classes.
  - If the input data values to a system can be specified by a range of value, then one valid and two invalid equivalence classes should be defined.
  - If the input data takes values from a set of discrete members of some domain, then one equivalence class for valid input values and another for invalid input values should be defined.

Example: For a software that computes the square root of an input integer which can assume values between 0 and 5000, there are three equivalence classes: the set of negative integers, the set of integers in the range between 0 and 5000, and the integers larger than 5000. therefore the test cases must include representative values from each of the three equivalence classes and a possible test set can therefore be :{-5, 500, 1000}

- **Boundary Value Analysis:** A type of programming error often occurs at boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the inputs values that lie at the boundary of different equivalence classes. For example, programmers may improperly uses instead of <=, or conversely <= instead of <. Boundary value analysis leads to selection of test cases at the boundaries of different equivalence classes.
- **Cause effect graph:** The Cause-Effect graph method is a technique for mapping input to output/response. It is graphical representation of inputs and the associated outputs effects which can be used to design test cases. It consists of the following steps
    - Decompose the unit to be tested, if it contains many facilities.
    - Identify the causes
    - Identify the effects
    - Establish a graph of relations between causes and effects
    - Complete the graph by adding constraints between causes and effects
    - Convert the graph to a decision table and test cases can be derived from it.

Let us define a simple example to explain cause effect graph. If we say a statement

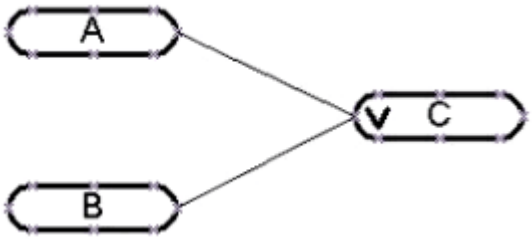
"If A OR B then C".

Here the causes are due to A and B and effect is C. The following rule holds for the above statement

- If A is true and B is true then C is true
- If A is false and B is false then C is true
- If A is true and B is false then C is true

- If A is false and B is false then C is false

This can be represented with the following cause effect graph



In this figure A, B and C are nodes. A and B are causes and C is the effect.

15.3 WHITE BOX TESTING

White Box Testing is the testing of a software solution's internal coding and infrastructure. It focuses primarily on strengthening security, the flow of inputs and outputs through the application, design and usability.

- **Statement coverage:** The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless we execute a statement, we have no way of determining if an error exists in that statement unless a statement is executed, we cannot observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is not guarantee that it will behave correctly for all input values. In the following, we illustrate how test cases can be designed using the statement coverage strategy.

Example: Consider the following Euclid's GCD computation algorithm

```
int compute- gcd (x,y)
    int x, y;
{
    1 while (x!=y)}
```

```
2 if (x>y) then
3 x=x-y;
4 else y=y-x;
5 {
6 return x;
}
```

By choosing the test set {(x=3, y=3), (x=4, y=3), (x=3, y=4)}. We can exercise the program such that all statements are executed at least once.

- **Branch Coverage:** In the branch coverage based testing strategy; test cases are designed to make each branch condition to assume true and false values. Branch testing is also known as edge testing as in this testing scheme, each edge of program's control flow graph is traversed at least once.

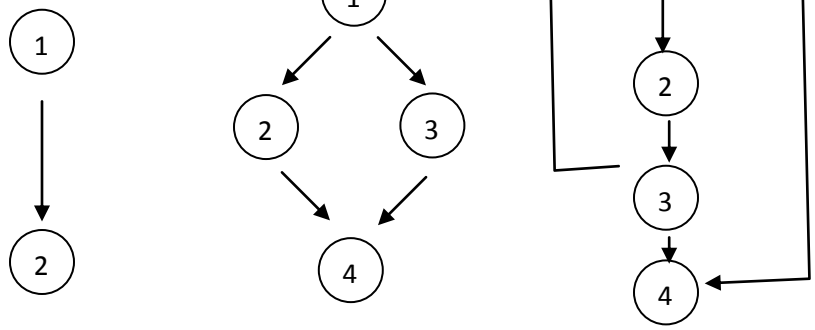
It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy than the statement coverage – based testing. For the program of example – the test cases for branch coverage can be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}

- **Condition coverage:** In this structural testing, test cases are designed to make each component as a composite condition expression that assumes both true and false values. For example, in the conditional expression ((G and c_2) OR (3), the components c_1, c_2 and c_3 are each made to assume both true and false values. Branch testing is probably the simplest condition testing strategy where only the compound condition appearing in the different branch statement are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is a stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of an components, for condition coverage, 2h test cases are required. Thus for condition coverage, the number of test cases increases exponentially with the number of component condition. Therefore, a condition coverage-based testing technique is pro critical only if n (number of conditions) is small.

- **Path Coverage:** The path coverage-based testing strategy requires up to design test cases such that all linearly independent paths in the program are executed at first once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Therefore, in order to understand the path coverage –based testing strategy, we need to first understand how the CFG of a program can be drawn.
- **Control Flow Graph (CFG):** A control flow graph describes how the control flows through the program. In other words, it expresses the sequence in which the different instructions of a program get executed. In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as the nodes of the control flow graph. An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

A program is made up from these types of statement the sequences, selection, and iteration. We can easily draw the CFG for any program, if we know how to represent these statements. It is important to note that for the iteration type of constructs such as the while construct, the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas the CFG of can be drawn as shown in figure below

Sequence	Selection	Iteration
1. a = s	1. If (a > b)	1. while (a > b){
2. b = q * 2 - 1	2. c = 3;	2. b = b - 1;
	3. else c = 5;	3. b = b * a;}
	4. c = c * c;	4. c = a + b;



CFG for (a) sequences, (b) selection, and (c) iteration

int compute-gcd (int x, int y){

1. while (x! = y){

2. if (x < y) then

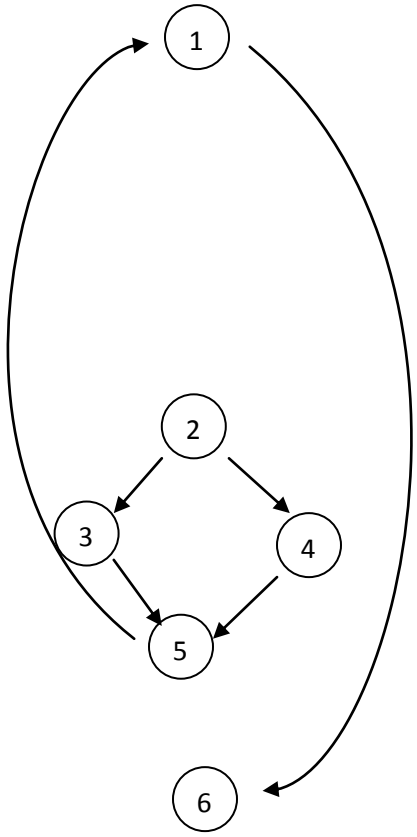
3. x = x – y;

4. else y = y – x;

5. }

6. return x;

}



(a) Example program (b) Control flow graph

Path: A path in a control flow graph is a sequence of node and edge from the starting node to a terminal node. There can be more than one terminal node in a program, so many paths are possible. Test cases to cover all paths of a typical program is impractical to define, since there can be an infinite number of paths through a program in the presence of loops. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths: the number of test cases required may become indefinitely large.

- **Linearly independent path:** A linearly independent path is a path through the program that introduces at least one new edge that is not included in any other linearly independent path. Note that if a path has one new node compared to all other linearly independent path, then the path is also linearly independent. This is because any path

having a new node automatically implies that it has a new edge. Thus, a path that is a sub path of another path is not considered to be a linearly independent path.

It is straight forward to identify the linearly independent path of simple programs, but for more complicated program it is not easy to determine. McCabe's cyclomatic complexity defines an upper-bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute.

- **McCabe's Cyclomatic Complexity Metric:** This is also called as the structural complexity of the program. It defines an upper bound on the number of independent paths in a program. There are three different ways to compute the cyclomatic complexity and all of them results into the same value.

  Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as

  $$V(G) = E - N + 2$$

  where N is the number of nodes of the control flow graph and F is the number of edges in the control flow graph.

  Method 2: Another way of computing the cyclomatic complexity of a program from an inspection of its controls flow graph is as follows:

  V (G) = Total number of bounded cases +1

  In the program's control flow graphs G, any region enclosed by nodes and edges can be called a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But what if the graph G is not planar, i.e. however you may draw the graph, two or more edges intersect? Actually it can be shown that structured programs always yield planar graphs. But the presence of GOTOs can easily add intersecting edges. Therefore for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used.

  The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing the difficulty

and the ultimate reliability. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of program, then the McCabe's metric is equal to N+1 knowing the number of test cases required does not make it any easier to derive the test cases, only it gives an indication of the minimum number of test cases required for path coverage. However, for the CFG of a simple program segment of say 20 nodes and 30 edges, you may need hours to identify the entire linearly independent path in it and so design to test cases. Therefore for path testing, usually the tester proposes an initial set of test data using his experience and judgment (without identifying any independent path). A testing tool such as the dynamic program analyzer is used to indicate the percentage of linearly independent path covered by the test case.

- **Derivation of Test Cases:** The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases of a program:

    1. Draw the control flow graph
    2. Determine V (G)
    3. Determine the basic set of linearly independent paths
    4. Prepare the test case that will force execution of each path in the basic set.

Another interesting application of the cyclomatic complexity of programs is as follows. Experimental studies indicate that there exists a distinct relationship between the McCabe's metric and the number of errors existing in the code, as well as the time required to find and correct such errors. It is also generally accepted that the cyclomatic complexity of a program is an indication of the psychological complexity or the level of difficult in understanding the program.

Example: Calculate cyclomatic complexity of the compute_gcd() program defined earlier.

Method 1: For the CFG of figure shown for the compute_gcd() function , E = 7 and N = 6. Therefore the cyclomatic complexity,

$$V(G) = 7 - 6 + 2 = 3$$

Method 2: The number of bounded cases in the example =2

V (G) = Total number of bounded cases +1 =2 + 1 =3

Method 3: The number of decision statements in compute_gcd() function is 2, therefore cyclomatic complexity will be 2+1=3.

We can see here that all the three methods will generate same result.

15.4 LET US SUM UP

In this chapter we learnt about black box and white box testing. We studied black box testing approach which are equivalence partitioning and boundary value analysis. We then saw the use of cause effect graph. In white box testing we learnt about statement coverage, branch coverage and condition coverage. We then studied about control flow graph and finally we learnt about cyclomatic complexity.

15.5 REFERENCES AND SUGGESTED READING

(1) Software Engineering, Practitioner Approach, 7th Edition, by R.S. Pressman, Tata McGraw Hill, India, 2009.
(2) Integrated Approach to Software Engineering by, Pankaj Jalote, Narosa Publications, 2003.
(3) Introduction to Software engineering by Rajiv Mall – PHI, 2000.
(4) Software Engineering by I. Somerville, 7th Edition, Pearson Education, India, 2006.
(5) Software Testing Techniques by B. Bezier PHI, India, 2001

15.6 EXERCISE

1. Explain black box testing.

2. Describe equivalence class partition.

3. Explain cause and effect graph technique with an example.

4. Discuss white box testing.

5. Explain control flow graph with and example.

6. Explain cyclomatic complexity.