Department of

Computer Science & Engineering

LAB MANUAL
SOFTWARE ENGINEERING LAB

B.Tech– VI Semester

KCT College OF ENGG AND TECH.
VILLAGE FATEHGARH
DISTT.SANGRUR

INDEX

S.NO.	NAME OF THE PROGRAM	PAGE NO.
1.	Introduction to DFD.	4-6
2.	Create Level 0 DFD using Smart Draw.	7-8
3.	Create Level 1 DFD using Smart Draw. Create Level 2 DFD using Smart Draw	9-10
4.	Drawing Entity Relationship diagram in Smart Draw.	11-12
5.	Representing mapping cardinalities in Entity Relationship diagram.	13-15
6.	Representing relationships in Entity Relationship diagram using Smart Draw.	16-17
7.	Representing sequence in a structured chart. Creating modules in a structured chart.	18-23
8.	Representing interrelationship in modules in structured chart. Converting DFD into structured chart.	24-26
9.	Introduction to Java Server Pages.	27-28
10.	Beyond Syllabus	29-30

Experiment No.1

Title: - INTRODUCTION TO DFD

Objective: - To get familiar with the data flow diagrams

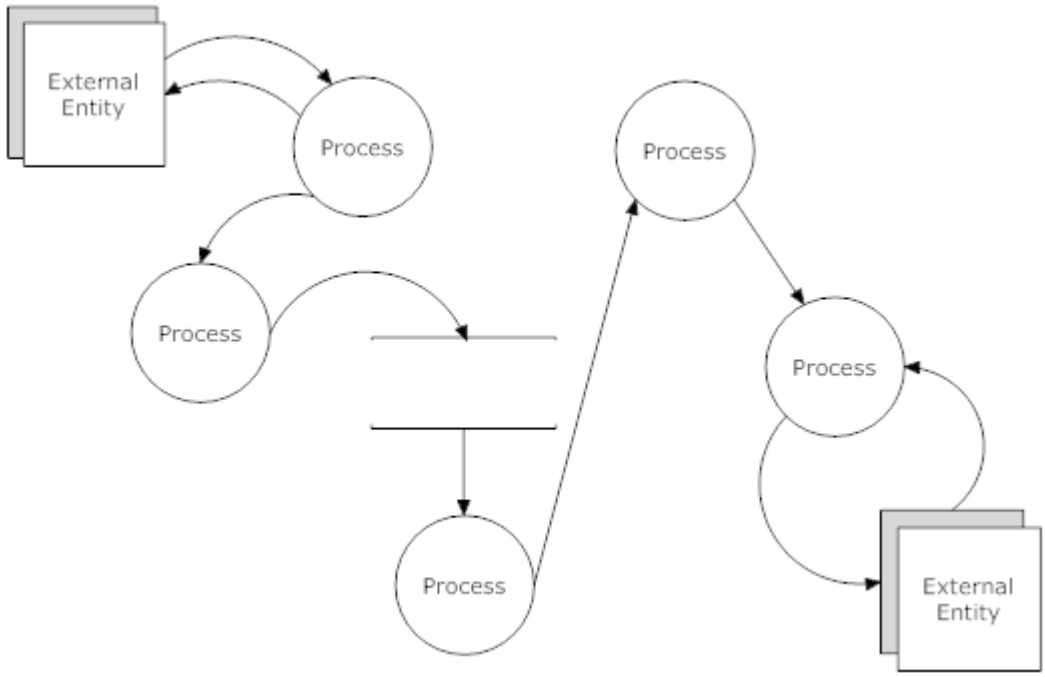
S/W Requirement: - Smart Draw

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Theory:-

Data flow diagrams illustrate how data is processed by a system in terms of inputs and outputs.



A data flow diagram

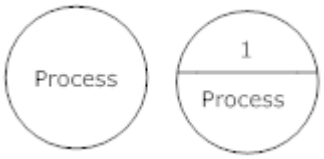
Data Flow Diagram Notations

You can use two different types of notations on your data flow diagrams: *Yourdon & Coad* or *Gane & Sarson*.

Process

A process transforms incoming data flow into outgoing data flow.

Process Notations



DataStore

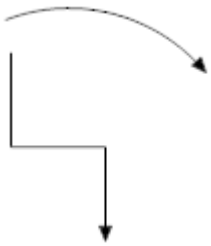
Datastores are repositories of data in the system. They are sometimes also referred to as files.

Datastore Notations

Dataflow

Dataflows are pipelines through which packets of information flow. Label the arrows with the name of the data that moves through it.

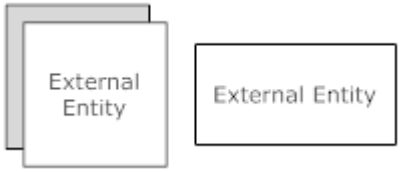
Dataflow Notations



External Entity

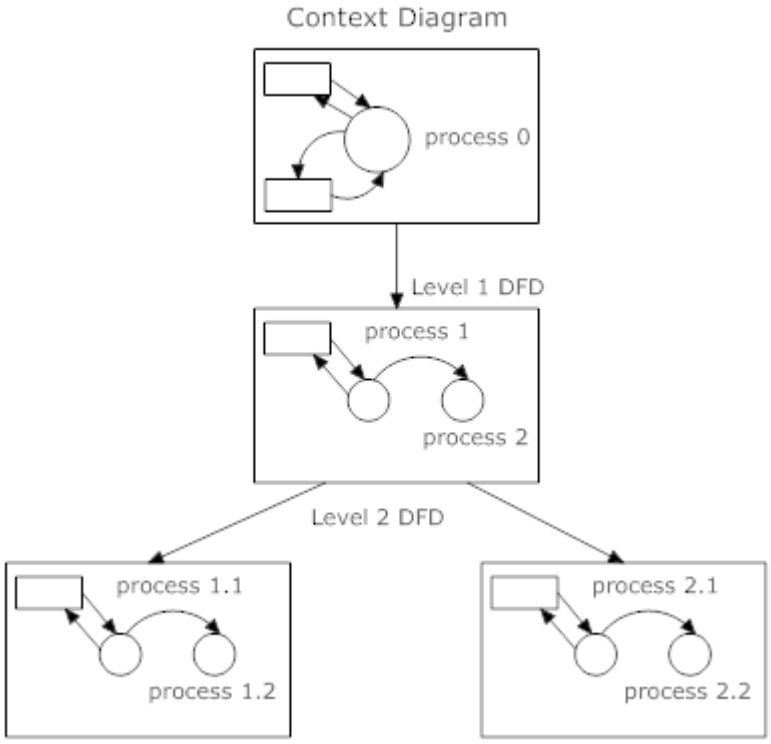
External entities are objects outside the system, with which the system communicates. External entities are sources and destinations of the system's inputs and outputs.

External Entity Notations



Data Flow Diagram Layers

Draw data flow diagrams in several nested layers. A single process node on a high level diagram can be expanded to show a more detailed data flow diagram. Draw the context diagram first, followed by various layers of data flow diagrams.



The nesting of data flow layers

RESULT: This experiment introduced the data flow diagrams and various notations used in the data flow diagrams.

Experiment No.2

Title:- CREATE A LEVEL 0 DFD USING SMART DRAW

Objective :- To get familiar with the LEVEL 0 DFD

S/W Requirement :- Smart Draw

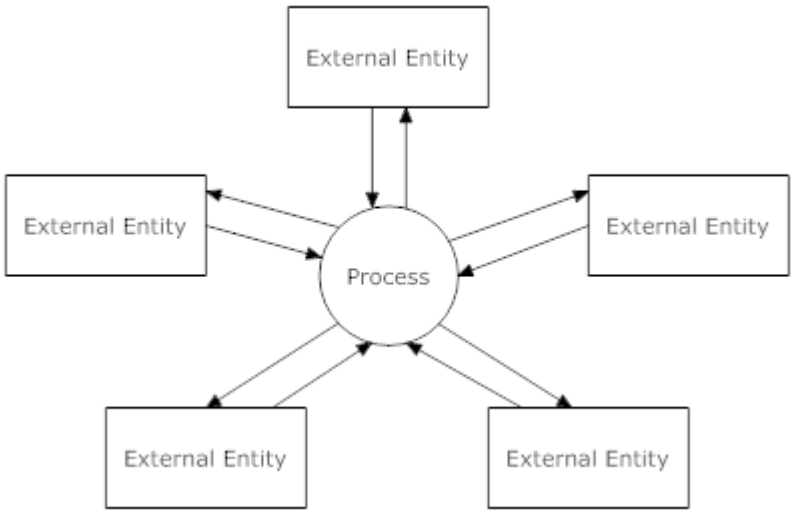
H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

Context Diagrams

A context diagram is a top level (also known as Level 0) data flow diagram. It only contains one process node (process 0) that generalizes the function of the entire system in relationship to external entities.



Data Flow Diagrams – Context Diagram Guidelines

Firstly, draw and name a single process box that represents the entire system.
Next, identify and add the external entities that communicate directly with the process box. Do this by considering origin and destination of the resource flows and data flows.
Finally, add the resource flows and data flows to the diagram.

RESULT: This experiment introduced the level 0 data flow diagram.

Experiment No.3

Title:- CREATE A LEVEL 0 DFD USING SMART DRAW

Objective :- To get familiar with the LEVEL 1 DFD

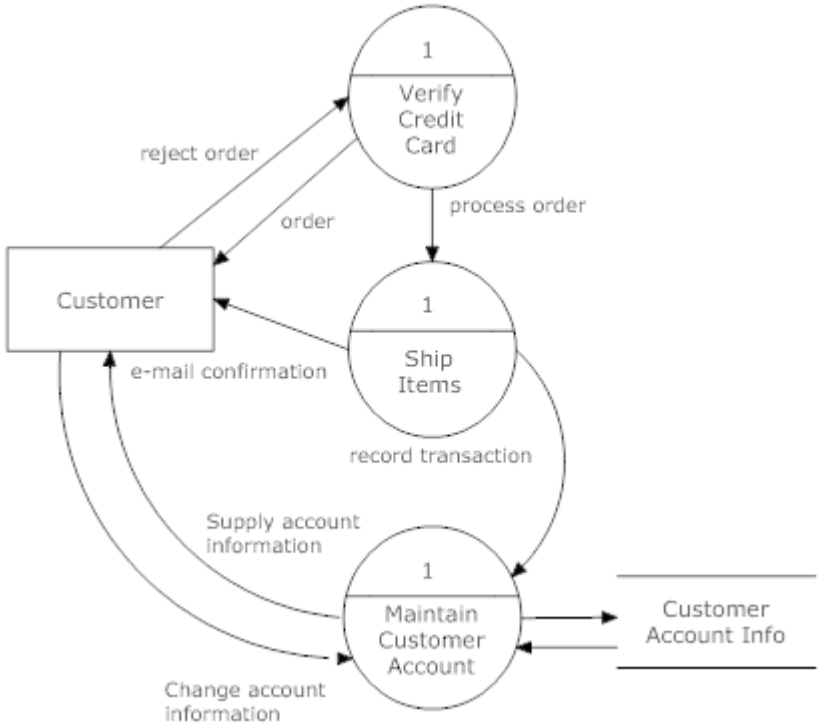
S/W Requirement :- Smart Draw

H/W Requirement :-

•**Processor** – Any suitable Processor eg. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

The first level DFD shows the main processes within the system. Each of these processes can be broken into further processes until you reach pseudocode.



An example first-level data flow diagram

Steps

- Start from the context diagram. Identify the parent process and the external entities with their net inputs and outputs.
- Place the external entities on the diagram. Draw the boundary.
- Identify the data flows needed to generate the net inputs and outputs to the external entities.
- Identify the business processe to perform the work needed to generate the input and output dataflows.
- Connect the data flows from the external entities to the processes.
- Identify the datastore.
- Connect the processes and data stores with data flows.
- Apply the Process Model Paradigm to verify that the diagram addresses the processing needs of all external entities.
- Apply the External Control Paradigm to further validate that the flows to the external entities are correct.
- Continue to decompose to the nthlevel DFD. Draw all DFDs at one level before moving to the next level of decomposing detail. You should decompose horizontally first to a sufficient nth level to ensure that the processes are partitioned correctly; then you can begin to decompose vertically.

RESULT: This experiment introduced the level 1 data flow diagram.

Experiment No.4

Title:- CREATE A LEVEL 2 DFD USING SMART DRAW

Objective :- To get familiar with the LEVEL 2 DFD

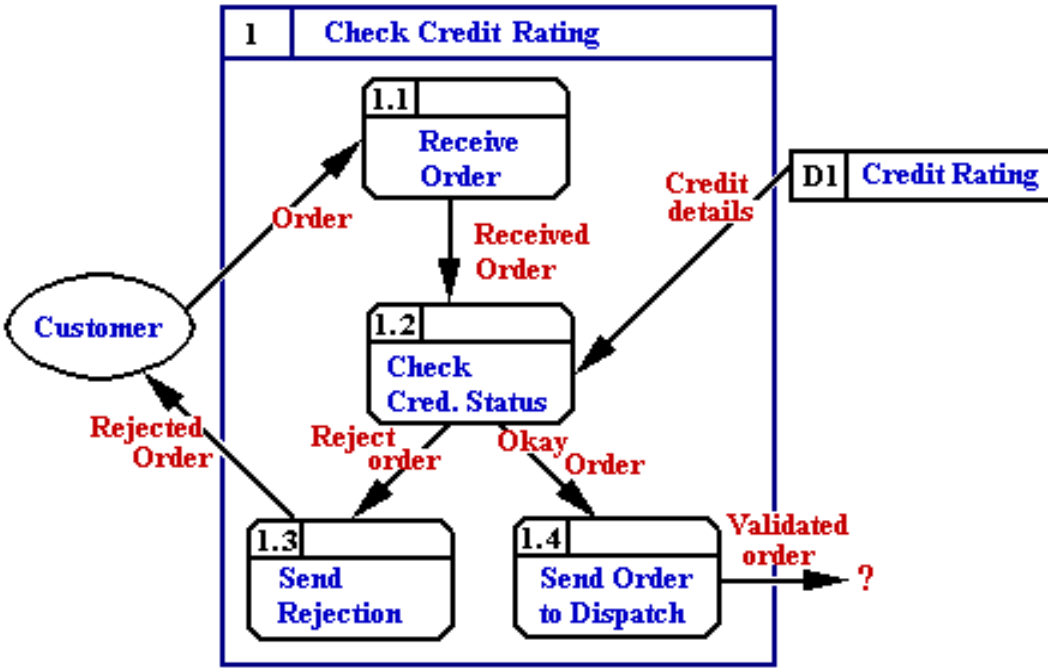
S/W Requirement :- Smart Draw

H/W Requirement :-

- **Processor** – Any suitable Processor e.g. Celeron
- **Main Memory** - 128 MB RAM
- **Hard Disk** – minimum 20 GB IDE Hard Disk
- **Removable Drives**–1.44 MB Floppy Disk Drive
  –52X IDE CD-ROM Drive
- PS/2 HCL **Keyboard and Mouse**

Method:-

- Each Process box in the Level 1 diagram will itself be made up of a number of processes, and will need to be decomposed as a Level 2 diagram.
- In complex systems, decomposition may need to go to Level 3 or 4+



RESULT: This experiment introduced the level 1 data flow diagram.

Experiment No.5

Title:- Drawing Entity Relationship diagram in Smart Draw.

Objective :- To get know Entity Relationship diagram.

S/W Requirement :- Smart Draw

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

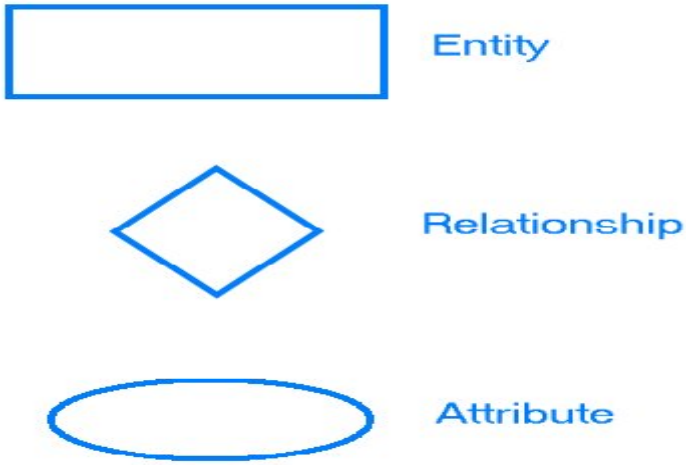
Method:-

An **entity-relationship model** (ERM) in software engineering is an abstract and conceptual representation of data. Entity-relationship modeling is a relational schema database modeling method, used to produce a type of conceptual schema or semantic data model of a system, often a relational database, and its requirements in a top-down fashion.

A sample ER diagram

Diagrams created using this process are called *entity-relationship diagrams*, or *ER diagrams* or *ERDs* for short.

Generally E-R Diagrams require the use of the following symbols:

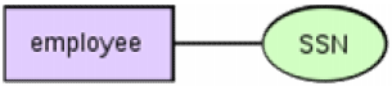


Connection

An entity may be defined as a thing which is recognized as being capable of an independent existence and which can be uniquely identified. An entity is an abstraction from the complexities of some domain. When we speak of an entity we normally speak of some aspect of the real world which can be distinguished from other aspects of the real world.

Two related entities

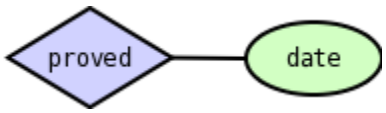
An entity may be a physical object such as a house or a car, an event such as a house sale or a car service, or a concept such as a customer transaction or order. Although the term entity is the one most commonly used, following Chen we should really distinguish between an entity and an entity-type. An entity-type is a category. An entity, strictly speaking, is an instance of a given entity-type. There are usually many instances of an entity-type. Because the term entity-type is somewhat cumbersome, most people tend to use the term entity as a synonym for this term.



An entity with an attribute

Entities can be thought of as nouns. Examples: a computer, an employee, a song, a mathematical theorem. Entities are represented as rectangles.

A relationship captures how two or more entities are related to one another. Relationships can be thought of as verbs, linking two or more nouns. Examples: an *owns* relationship between a company and a computer, a *supervises* relationship between an employee and a department, a *performs* relationship between an artist and a song, a *proved* relationship between a mathematician and a theorem. Relationships are represented as diamonds, connected by lines to each of the entities in the relationship.



A relationship with an attribute

Entities and relationships can both have attributes. Examples: an *employee* entity might have a *Social Security Number* (SSN) attribute; the *proved* relationship may have a *date* attribute. Attributes are represented as ellipses connected to their owning entity sets by a line.

Every entity (unless it is a weak entity) must have a minimal set of uniquely identifying attributes, which is called the entity's primary key.

Primary key

Entity-relationship diagrams don't show single entities or single instances of relations. Rather, they show entity sets and relationship sets. Example: a particular *song* is an entity. The collection of all songs in a database is an entity set. The *eaten* relationship between a child and her lunch is a single relationship. The set of all such child-lunch relationships in a database is a relationship set.

Lines are drawn between entity sets and the relationship sets they are involved in. If all entities in an entity set must participate in the relationship set, a thick or double line is drawn. This is called a participation constraint. If each entity of the entity set can participate in at most one relationship in the relationship set, an arrow is drawn from the entity set to the relationship set. This is called a key constraint. To indicate that each entity in the entity set is involved in exactly one relationship, a thick arrow is drawn.

RESULT: This experiment introduces the concept of entity relationship diagram.

Experiment No.6

Title:- Representing mapping cardinalities in Entity Relationship diagram
Objective :-To get familiarize with mapping cardinalities

S/W Requirement :- Smart Draw

H/W Requirement :-

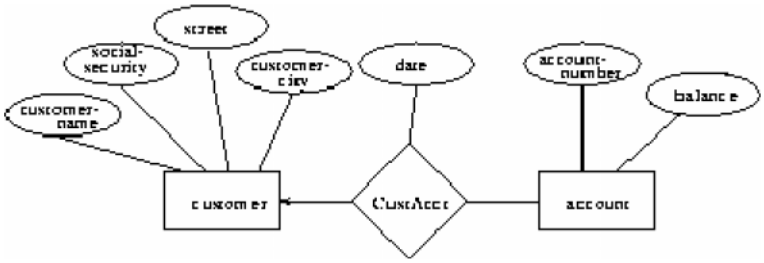
•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

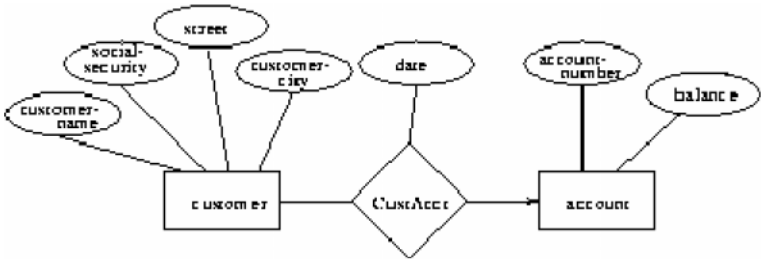
Mapping Cardinalities: express the number of entities to which another entity can be associated via a relationship. For binary relationship sets between entity sets A and B, the mapping cardinality must be one of:

1. **One-to-one**: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.
2. **One-to-many**: An entity in A is associated with any number in B. An entity in B is associated with at most one entity in A.
3. **Many-to-one**: An entity in A is associated with at most one entity in B. An entity in B is associated with any number in A.
4. **Many-to-many**: Entities in A and B are associated with any number from each other.

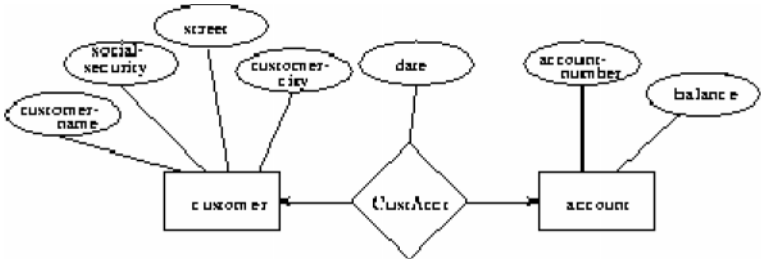
The appropriate mapping cardinality for a particular relationship set depends on the real world being modeled.



One-to-many from *customer* to *account*



Many-to-one from *customer* to *account*



One-to-one from *customer* to *account*

RESULT: This experiment introduces the concept of mapping cardinalities in entity relationship diagram.

Experiment No.7

Title:- Representing relationships in Entity Relationship diagram using Smart Draw.

Objective :-To get familiarize with relationships in ER diagrams

S/W Requirement :- Smart Draw

H/W Requirement :-

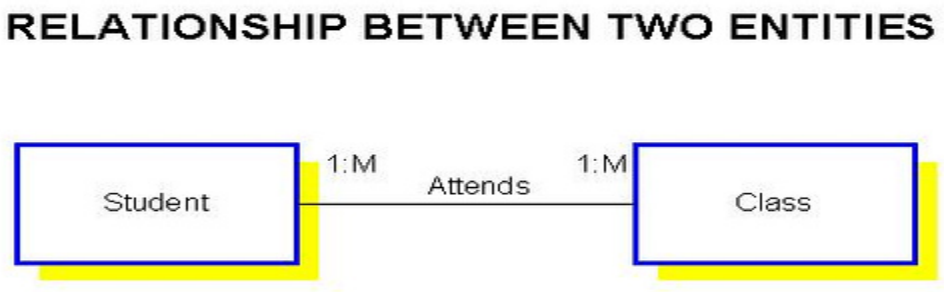
•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

A relationship is an association that exists between two entities. For example, Instructor teaches Class or Student attends Class. Most relationships can also be stated inversely. For example, Class is taught by Instructor.

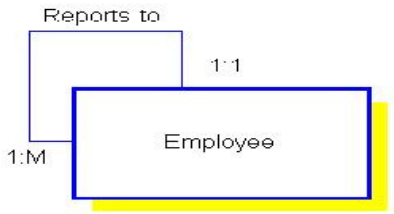
The relationships on an Entity-Relationship Diagram are represented by lines drawn between the entities involved in the association. The name of the relationship is placed either above, below, or beside the line.

Relationships Between Entities
There can be a simple relationship between two entities. For example, Student attends a Class:



Some relationships involve only one entity. For example, Employee reports to Employee:

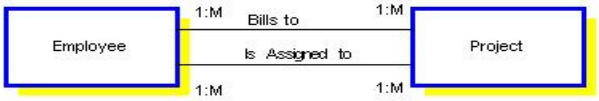
RELATIONSHIP BETWEEN ONE ENTITY



This type of relationship is called a recursive relationship.

There can be a number of different relationships between the same two entities. For example:

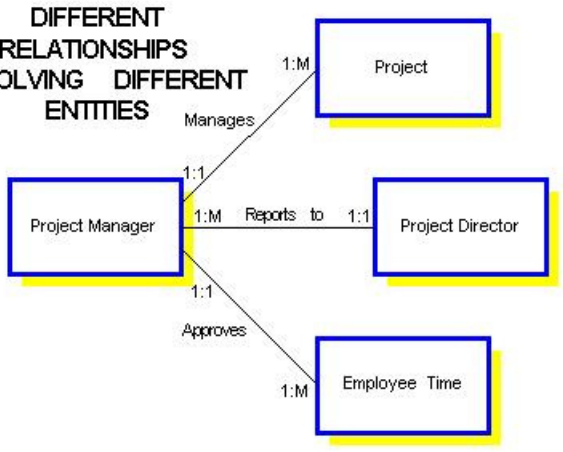
DIFFERENT RELATIONSHIPS BETWEEN TWO ENTITIES



- Employee is assigned to a Project,
- Employee bills to a Project.

One entity can participate in a number of different relationships involving different entities. For example:

DIFFERENT RELATIONSHIPS INVOLVING DIFFERENT ENTITIES



- Project Manager manages a Project,
- Project Manager reports to Project Director,

- Project Manager approves Employee Time.

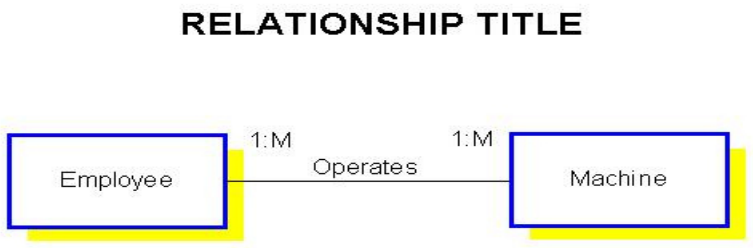
Characteristics of Relationships
A relationship may be depicted in a variety of ways to improve the accuracy of the representation of the real world. The major aspects of a relationship are:

RELATIONSHIP TITLE

Naming the Relationship
Place a name for the relationship on the line representing the relationship on the E-R diagram. Use a simple but meaningful action verb (e.g., buys, places, takes) to name the relationship. Assign relationship names that are significant to the business or that are commonly understood in everyday language.

Bi-directional Relationships
Whenever possible, use the active form of the verb to name the relationship. Note that all relationships are bi-directional. In one direction, the active form of the verb applies. In the opposite direction, the passive form applies.

For example, the relationship Employee operates Machine is named using the active verb operates:



However, the relationship Machine is operated by Employee also applies. This is the passive form of the verb. By convention, the passive form of the relationship name is not included on the E-R diagram. This helps avoid clutter on the diagram.

RELATIONSHIP CARDINALITY

Relationship cardinality identifies the maximum number of instances in which an entity participates in a relationship. There are three types of relationship cardinality:

- one-to-one,
- one-to-many,
- many-to-many.

Foreign Keys

To relate one entity to another, make the primary key of one entity an attribute of the other entity (foreign key).In a one-to-one relationship the foreign key may be placed in either of the entities. In a one-to-many or many-to-one relationship the foreign key is placed in the entity that has the many relationship.

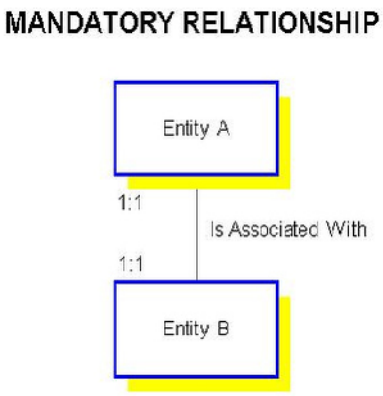
RELATIONSHIP DEPENDENCY

Three relationship dependencies are possible:

- mandatory,
- optional,
- contingent.

Relationship dependencies may be of different degrees. Each relationship dependency is illustrated differently.

Mandatory Relationship: A mandatory relationship indicates that for every occurrence of entity



A there must exist an entity B, and vice versa.
When specifying a relationship as being mandatory one-to-one, you are imposing requirements known as integrity constraints. For example, there is one Project Manager associated with each Project, and each Project Manager is associated with one Project at a time. A Project Manager may not be removed if the removal causes a Project to be without a Project Manager. If a Project Manager must be removed, its corresponding project must also be removed. A Project may not be removed if it leaves a Project Manager without a Project. A new project may be added if it can be managed by an existing Project Manager. If there is no Project Manager to manage the Project, a Project Manager must be added with the addition of a new Project.

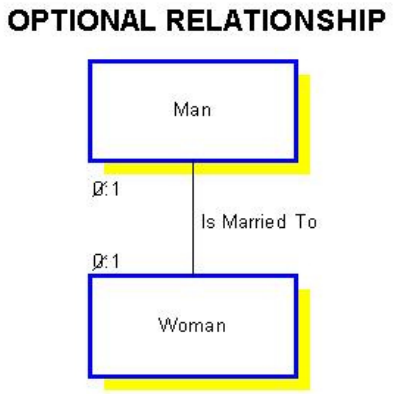
When specifying a relationship as being mandatory one-to-many or many-to-one, you are imposing integrity constraints. For example, an Employee is assigned one to many tasks and a task is assigned to one and only one Employee. There would not be a Task without an Employee, and there would not be an Employee without a Task. Similarly, if an Employee is added, Task must be added.

Some relationships are naturally or inherently mandatory. For example, consider the relationship Mother has Child. There would not be a Child without a Mother, nor would there be a Mother without a Child.

Other relationships are mandatory due to legislative or business rules, such as "a project is not considered to exist until it has been assigned a budget." This type of mandatory relationship should be analyzed to assess whether or not the rule is a temporary or unnecessary restriction.

Optional Relationship: An optional relationship between two entities indicates that it is not necessary for every entity occurrence to participate in the relationship. In other words, for both entities the minimum number of instances in which each participates, in each instance of the relationship is zero (0).

As an example, consider the relationship Man is married to Woman. Both entities may be depicted in an Entity-Relationship Model because they are of interest to the organization. However, not every man, or woman, is necessarily married. In this relationship, if an employee is not married to another employee in the organization, the relationship could not be shown.

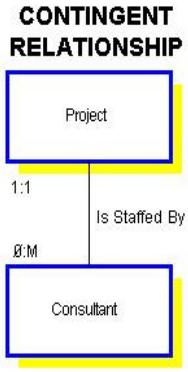


The optional relationship is useful for depicting changes over time where relationships may exist one day but not the next. For example, consider the relationship "Employee attends Training Seminar." There is a period of time when an Employee is not attending a Training Seminar or a Training Seminar may not be held.

Optional relationships may be unnecessary if an entity can be subdivided into subtypes or entirely different entities. For example, the entity Person could represent both employees and dependents in a superannuation system. In the wider aspects of a personnel system, an optional relationship would be necessary to link Person to Job. However, breaking down the Person entity into the separate entities Employee (which would have a mandatory relationship to Job), and Dependent (which would not be involved in such a relationship), provides a clearer representation.

Contingent Relationship: A contingent relationship represents an association which is mandatory for one of the involved entities, but optional for the other. In other words, for one of the entities the minimum number of instances that it participates in each instance of the relationship is one (1), the mandatory association, and for the other entity the minimum number of instances that it participates in each instance of the relationship is zero (0), the optional association.

For example, consider the relationship Man fathers Child. Not all occurrences of the entity Man will have produced a child. However, if an occurrence of the Child entity exists, it must be related to a Man entity. This is an inherent or natural contingent relationship. Contingent relationships may exist due to business rules, such as Project is staffed by Consultant.



In this case, a Project may or may not be staffed by a Consultant. However, if a Consultant is registered in the system, a business rule may state that a Consultant must be associated with a Project.

RESULT: This experiment introduces the concept of relationships in ER Diagrams.

Experiment No.8

Title :- Representing sequence in a structured chart.

Objective :- To familiarize with the concept of structured charts

S/W Requirement :- Smart Draw

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

Structured software design is arranged hierarchically. Structured software follows rules:

- Modules are arranged hierarchically.
- There is only one root (i.e., top level) module.
- Execution begins with the root module.
- Program control must enter a module at its entry point and leave at its exit point.
- Control returns to the calling module when the lower level module completes execution.

Description of a Module
Logically, a module is one problem-related task that the program performs, such as Create Invoice or Validate Customer Request. Physically, a module is implemented as a sequence of programming instructions bounded by an entry point and an exit point.

Common Modules
There are two categories of common modules:

- system (e.g., I/O handlers, locking),
- application (e.g., edits, calculations).

Some common modules (e.g., security, navigation, audit trails, and help) do not fall clearly in either category. These modules may be common to more than one application but are not system level modules.

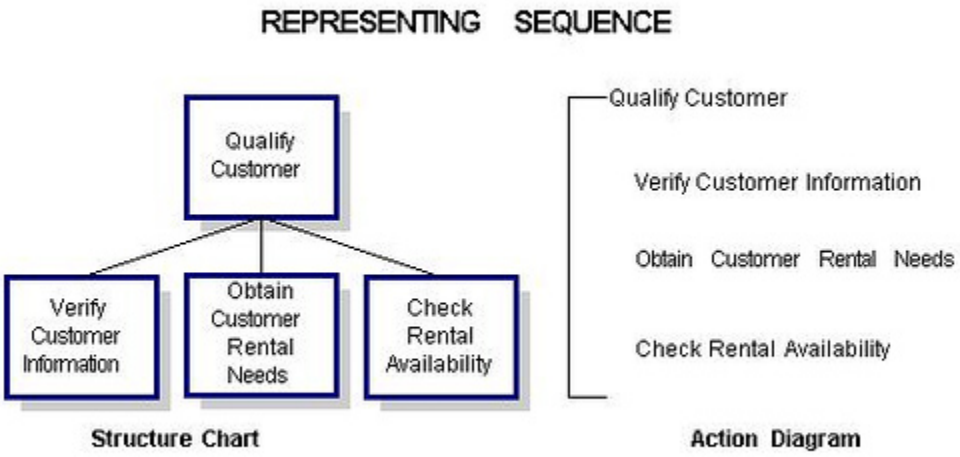
System common modules should be defined as part of the preliminary design. Application common modules should be defined as early as possible but many may not be identified until detailed design.

Constructs of Structured Software Design

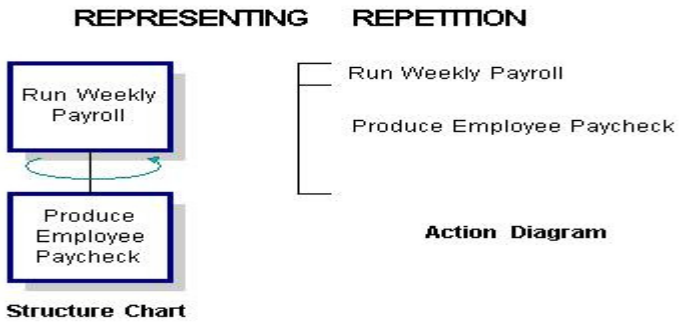
Tree-structure diagrams are used to illustrate modules that follow the rules of structured software design. A tree-structure can be drawn as a set of blocks, for example, a Structure Chart, or a set of brackets, such as an Action Diagram.

When designing structured software, three basic constructs are represented:

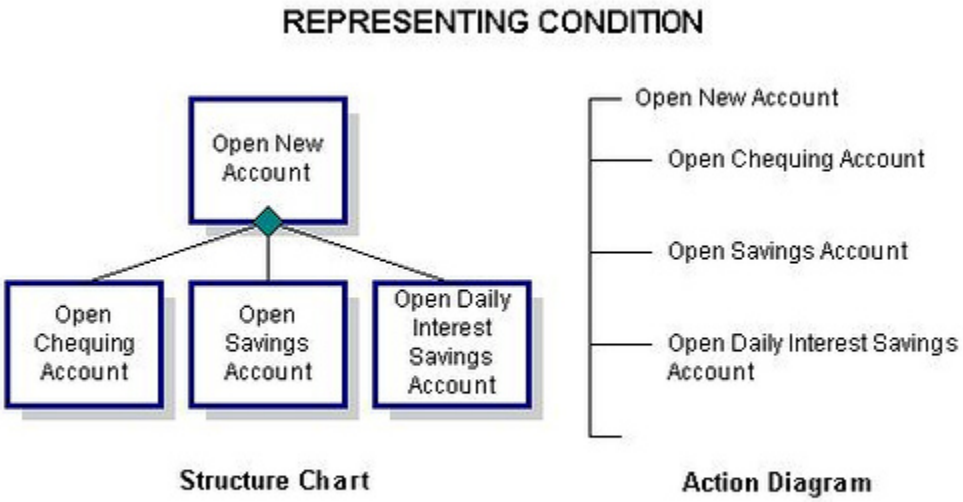
- Sequence - items are executed from top to bottom.



- Repetition - a set of operations is repeated. The repetition is terminated based on the repetition test.



- Condition - a set of operations are executed only if a certain condition or CASE statement applies.



RESULT: This experiment introduces the concept structured charts.

Experiment No.9

Title:- Creating modules in a structured chart.

Objective :- To get familiar with modules in structured charts

S/W Requirement :- Smart Draw

H/W Requirement :-

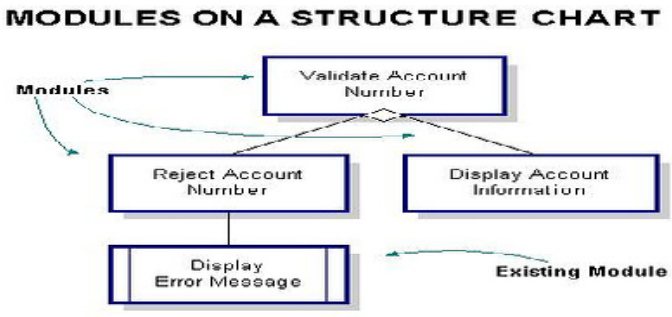
•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

A rectangle is used to represent a module on a Structure Chart. The module name is written inside the rectangle. Other than the module name, the Structure Chart gives no information about the internals of the module.

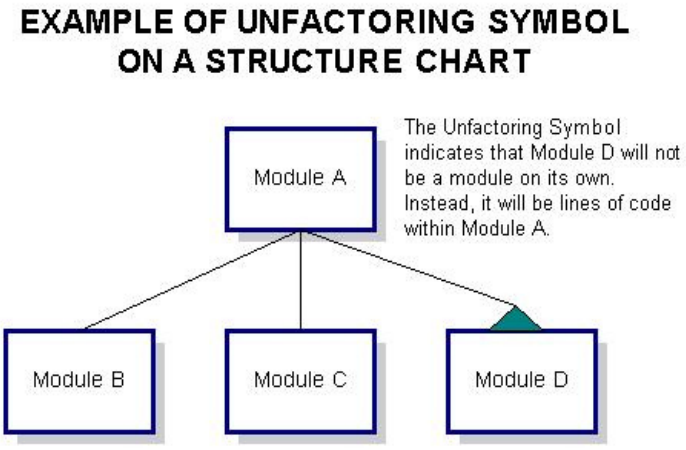
Module Names and Numerical Identifiers
Each module must have a module name. Module names should consist of a transitive (or action) verb and an object noun. Module names and numerical identifiers may be taken directly from corresponding process names on Data Flow Diagrams or other process charts. The name of the module and the numerical identifier is written inside the module rectangle. Other than the module name and number, no other information is provided about the internals of the module.

Existing Module
Existing modules may be shown on a Structure Chart. An existing module is represented by double vertical lines.



Unfactoring Symbol

An unfactoring symbol is a construct on a Structure Chart that indicates the module will not be a module on its own but will be lines of code in the parent module. An unfactoring symbol is represented with a flat rectangle on top of the module that will not be a module when the program is developed.



An unfactoring symbol reduces factoring without having to redraw the Structure Chart. Use an unfactoring symbol when a module that is too small to exist on its own has been included on the Structure Chart. The module may exist because factoring was taken too far or it may be shown to make the chart easier to understand. (Factoring is the separation of a process contained as code in one module into a new module of its own).

RESULT: This experiment introduces the concept of creating modules in structured charts

Experiment No.10

Title:- Representing interrelationship in modules in structured chart.

Objective :- To get familiar with relationship of modules

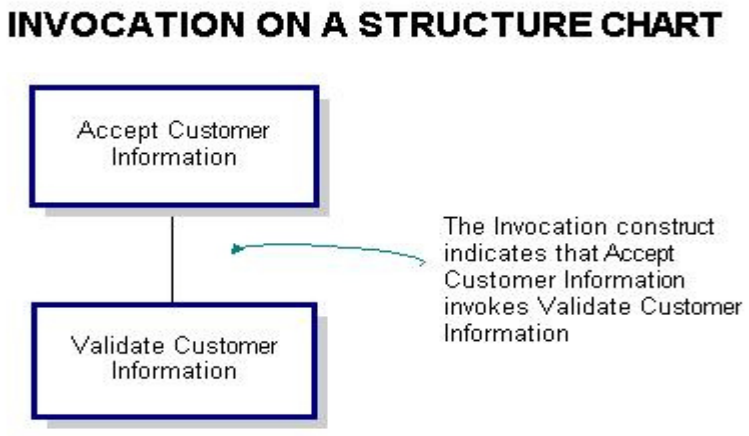
S/W Requirement :- Smart Draw

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

Invocation is a connecting line that shows the interrelationship among two modules. Modules are organized in levels and the invocation lines connect modules between levels. Early forms of Structure Charts drew arrowheads at the end of a line between two modules to indicate that program control is passed from one module to the second in the direction of the arrow. Since the Structure Chart is hierarchical, arrowheads are not necessary. Control is always passed from the higher level module to the lower level module. Also, eliminating arrowheads reduces clutter on the chart.



Control Rules Between Modules

The root level (i.e., top level) of the Structure Chart contains only one module. Control passes level by level from the root to lower level modules. Control is always returned to the invoking module. Control eventually returns to the root. There is a maximum of one control relationship between any two modules. In other words, if Module A invokes Module B, Module B cannot also invoke Module A.

If a module is called by more than one module and it is not an existing module, the module which calls it the most is referred to as its parent. A module can only have one parent even though it may be called by several modules.

Modules may communicate with other modules only by calling a module or through data or control couples. Couples are information passed between two modules. A module may not invoke a module that is higher in the hierarchy than the invoking module.

RESULT: This experiment introduces the concept of relationships among modules in structured charts

Experiment No.11

Title: - Converting DFD into structured chart

Objective: - To get familiar with the DFD and structured charts

S/W Requirement: - Smart Draw

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

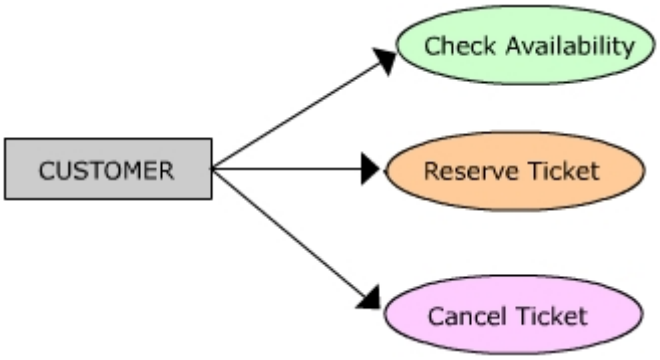
Theory:-

Strategies for converting the DFD into Structure Chart

Steps

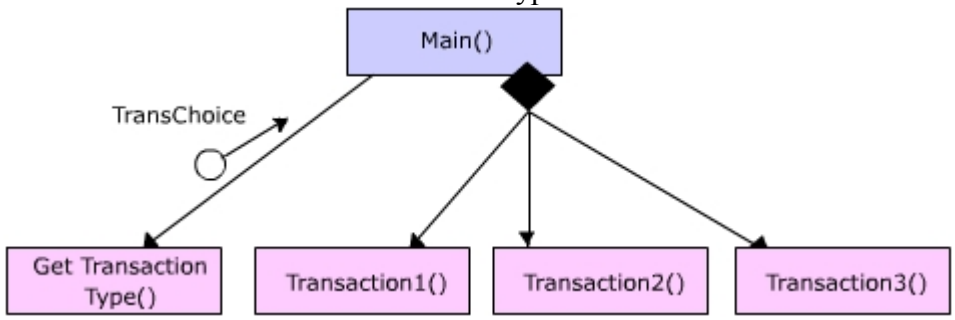
- Break the system into suitably tractable units by means of transaction analysis
- Convert each unit into into a good structure chart by means of transform analysis
- Link back the separate units into overall system implementation

Transaction Analysis
The transaction is identified by studying the discrete event types that drive the system. For example, with respect to railway reservation, a customer may give the following transaction stimulus:



The three transaction types here are: Check Availability (an enquiry), Reserve Ticket (booking) and Cancel Ticket (cancellation). On any given time we will get customers interested in giving any of the above transaction stimuli. In a typical situation, any one stimulus may be entered

through a particular terminal. The human user would inform the system her preference by selecting a transaction type from a menu. The first step in our strategy is to identify such transaction types and draw the first level breakup of modules in the structure chart, by creating separate module to co-ordinate various transaction types. This is shown as follows:



The Main ( ) which is a over-all coordinating module, gets the information about what transaction the user prefers to do through TransChoice. The TransChoice is returned as a parameter to Main ( ). Remember, we are following our design principles faithfully in decomposing our modules. The actual details of how GetTransactionType ( ) is not relevant for Main ( ). It may for example, refresh and print a text menu and prompt the user to select a choice and return this choice to Main ( ). It will not affect any other components in our breakup, even when this module is changed later to return the same input through graphical interface instead of textual menu. The modules Transaction1 ( ), Transaction2 ( ) and Transaction3 ( ) are the coordinators of transactions one, two and three respectively. The details of these transactions are to be exploded in the next levels of abstraction.

We will continue to identify more transaction centers by drawing a navigation chart of all input screens that are needed to get various transaction stimuli from the user. These are to be factored out in the next levels of the structure chart (in exactly the same way as seen before), for all identified transaction centers.

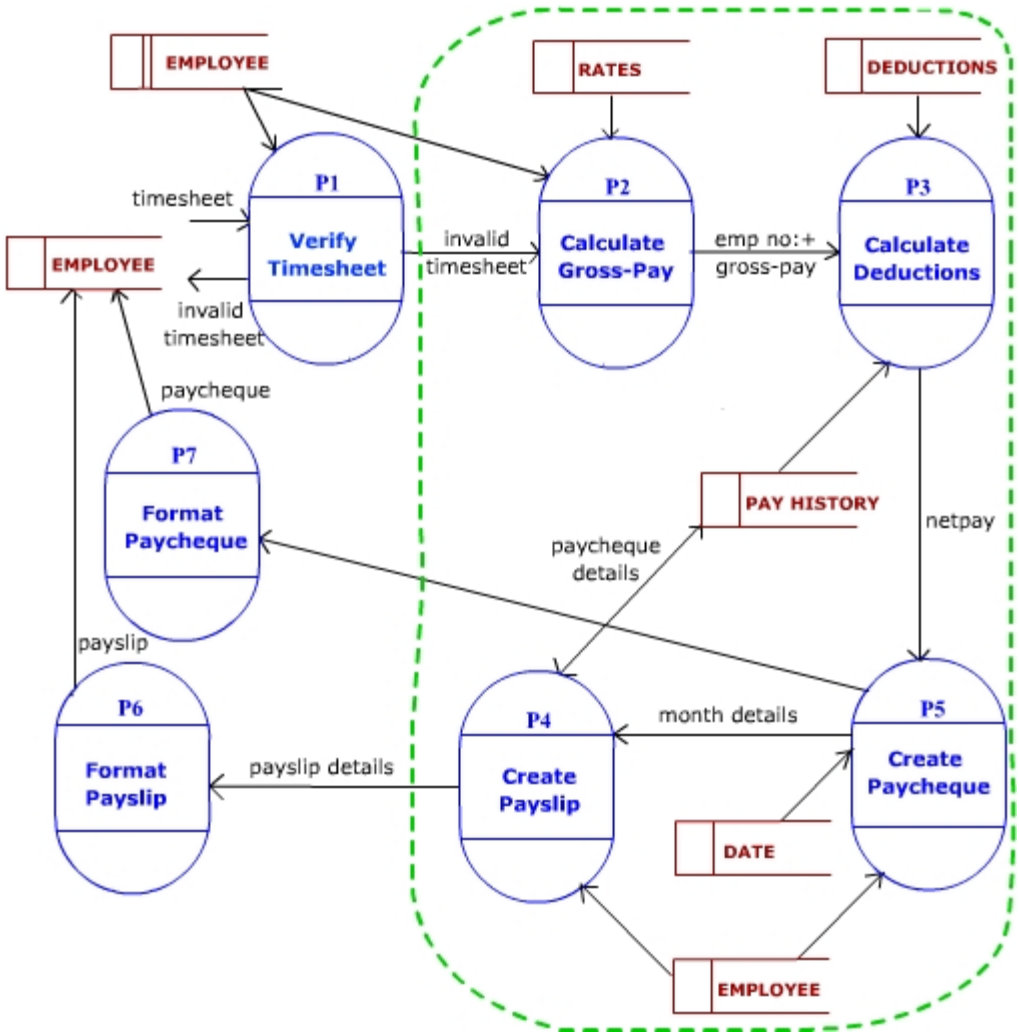
Transform Analysis

Transform analysis is strategy of converting each piece of DFD (may be from level 2 or level 3, etc.) for all the identified transaction centers. In case, the given system has only one transaction (like a payroll system), then we can start transformation from level 1 DFD itself. Transform analysis is composed of the following five steps [Page-Jones, 1988]:

1. Draw a DFD of a transaction type (usually done during analysis phase)
2. Find the central functions of the DFD
3. Convert the DFD into a first-cut structure chart
4. Refine the structure chart
5. Verify that the final structure chart meets the requirements of the original DFD

Let us understand these steps through a payroll system example:

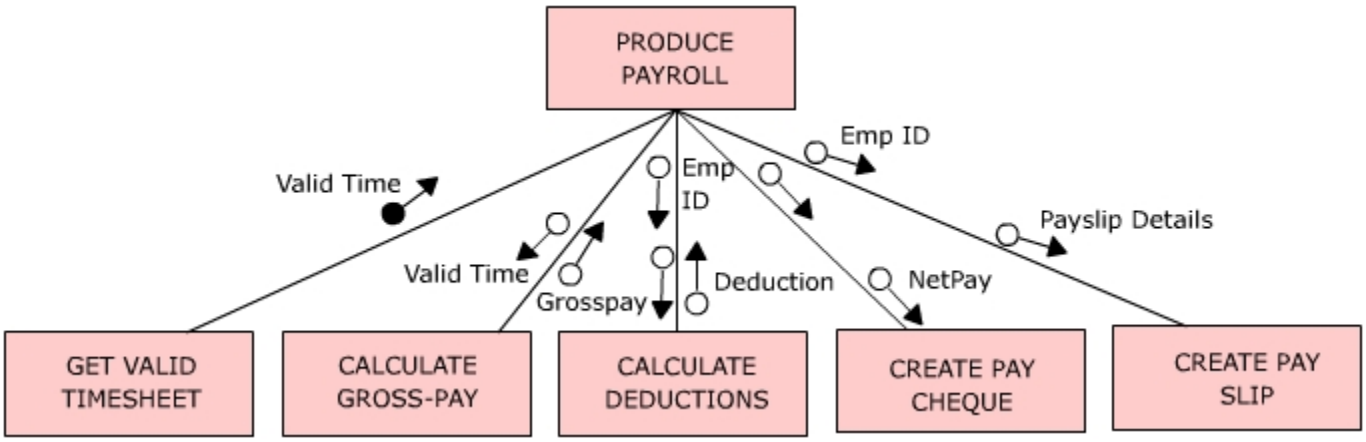
- **Identifying the central transform**



The central transform is the portion of DFD that contains the essential functions of the system and is independent of the particular implementation of the input and output. One way of identifying central transform (Page-Jones, 1988) is to identify the centre of the DFD by pruning off its afferent and efferent branches. Afferent stream is traced from outside of the DFD to a flow point inside, just before the input is being transformed into some form of output (For example, a format or validation process only refines the input – does not transform it). Similarly an efferent stream is a flow point from where output is formatted for better presentation. The processes between afferent and efferent stream represent the central transform (marked within dotted lines above). In the above example, P1 is an input process, and P6 & P7 are output processes. Central transform processes are P2, P3, P4 & P5 - which transform the given input into some form of output.

- **First-cut Structure Chart**

To produce first-cut (first draft) structure chart, first we have to establish a boss module. A boss module can be one of the central transform processes. Ideally, such process has to be more of a coordinating process (encompassing the essence of transformation). In case we fail to find a boss module within, a dummy coordinating module is created



In the above illustration, we have a dummy boss module "Produce Payroll" – which is named in a way that it indicate what the program is about. Having established the boss module, the afferent stream processes are moved to left most side of the next level of structure chart; the efferent stream process on the right most side and the central transform processes in the middle. Here, we moved a module to get valid timesheet (afferent process) to the left side (indicated in yellow). The two central transform processes are move in the middle (indicated in orange). By grouping the other two central transform processes with the respective efferent processes, we have created two modules (in blue) – essentially to print results, on the right side.
The main advantage of hierarchical (functional) arrangement of module is that it leads to flexibility in the software. For instance, if "Calculate Deduction" module is to select deduction rates from multiple rates, the module can be split into two in the next level – one to get the selection and another to calculate. Even after this change, the "Calculate Deduction" module would return the same value.

- **Refine the Structure Chart**

Expand the structure chart further by using the different levels of DFD. Factor down till you reach to modules that correspond to processes that access source / sink or data stores. Once this is ready, other features of the software like error handling, security, etc. has to be added. A module name should not be used for two different modules. If the same module is to be used in more than one place, it will be demoted down such that "fan in" can be done from the higher levels. Ideally, the name should sum up the activities done by the module and its sub-ordinates.

- **Verify Structure Chart vis-à-vis with DFD**

Because of the orientation towards the end-product, the software, the finer details of how data gets originated and stored (as appeared in DFD) is not explicit in Structure Chart. Hence DFD may still be needed along with Structure Chart to understand the data flow while creating low-level design.

- **Constructing Structure Chart (An illustration)**

Let us consider an illustration of structured chart. The following are the major processes in bank:

P1: Saving Procedure

P2: Current Procedure

P3: Loan Procedure

P4: DD Procedure

P5: MT Procedure

Since all are major subsystems with its own major processing, we first do transaction analysis on them

Some characteristics of the structure chart as a whole would give some clues about the quality of the system. Page-Jones (1988) suggest following guidelines for a good decomposition of structure chart:

- Avoid decision splits - Keep span-of-effect within scope-of-control: i.e. A module can affect only those modules which comes under it's control (All sub-ordinates, immediate ones and modules reporting to them, etc.)
- Error should be reported from the module that both detects an error and knows what the error is.
- Restrict fan-out (number of subordinates to a module) of a module to seven. Increase fan-in (number of immediate bosses for a module). High fan-ins (in a functional way) improves reusability.

RESULT: This experiment introduces the concept of DFD and structured charts

Experiment No.12

Title: - Introduction to Java Server Pages

Objective: - To get familiar with Java Server Pages

S/W Requirement: - Smart Draw

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Theory:-

Based on servlet technology, and currently shaping up at breakneck speed, JavaServer Pages (JSP) is set to be one of the most important elements of Java server programming. It's by no means complete yet, but that will change as the Java 2 Enterprise Edition comes together.

So what are JavaServer Pages? Well, they combine markup (whether HTML or XML) with nuggets of Java code to produce a dynamic web page. Each page is automatically compiled to a servlet by the JSP engine, the first time it is requested, and then executed. JSP provides a variety of ways to talk to Java classes, servlets, applets and the web server. With it, you can split the functionality of your web applications into components with well-defined public interfaces glued together by a simple page.

This model allows tasks to be sub-divided - a developer builds custom components and the page designer assembles the application with a few judicious method calls. In this 'application assembly' model, the business logic is separated from the presentation of data.

To give you an idea of the future, this separation of logic and presentation may become yet more extreme with the use of custom tags slated for JSP 1.1.

JavaServer Pages is a specification that is already implemented by several web servers on which your code will run without change, making it more portable and the server market more competitive than its rivals. Finally, it's nice and simple!

Architectural Overview

A JavaServer Page is a simple text file consisting of HTML or XML content along with JSP

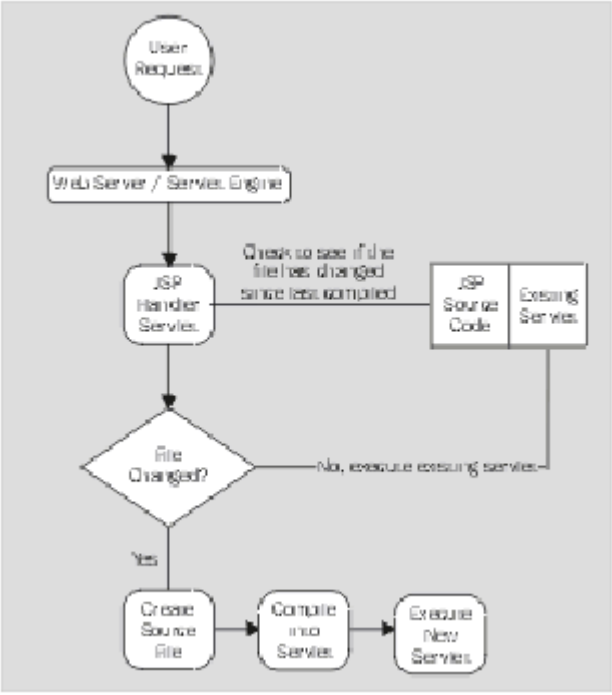
elements (a sort of shorthand for Java code). When a client requests a JSP page of the web server and it has not been run before, the page is first passed to a JSP engine which compiles the page to a servlet, runs it and returns the resulting content to the client. Thereafter, the web server's servlet engine will run the compiled page.
It should be no surprise, given the flexibility of the servlet model, that the current reference implementation of the JSP engine is itself a servlet.
It is possible to view the finished servlet code that is generated by locating it within the directory structure of the servlet engine. For example, with JRun, you can find the source code for your JSP files (in servlet form) in the jrun/jsm-default/services/jse/servlets/jsp directory. This is very helpful when trying to debug your JSP files.
If you take a look in the source files for the javax.servlet.jsp package, you'll find the following classes :

- JSPPage
- HttpJspPage

They define the interface for the compiled JSP page - namely that it must have three methods. Not surprisingly they are :

- jspInit()
- jspDestroy()
- _jspService(HttpServletRequest request, HttpServletResponse response)

The first two methods can be defined by the JSP author (we'll see how in a moment), but the third is the compiled version of the JSP page, and its creation is the responsibility of the JSP engine.



JSP Methods

It's time we saw a JSP file :

```
<html>
<head>
<title>Demo of a JSP page</title>
</head>
<body>

<!-- Set global information for the page -->
<%@ page language="java" %>

<!-- Declare the character variable -->
<%! char c = 0; %>

<!-- Scriptlet - Java code -->
<%
        for (int i = 0; i < 26; i++)
{
        for (int j = 0; j < 26; j++)
{
// Output capital
letters of the alphabet, and change starting letter
c = (char)(0x41 +
(26 - i + j)%26);
%>

<!-- Output the value of c.toString() to the HTML page -->
<%= c %>

<%
        }
%>
<br>
<%
        }
%>

</body>
</html>
```

This page just outputs the alphabet 26 times and changes the starting letter. The HTML is self-explanatory and written the way it should be, rather than cluttering up methods of a servlet.

Elements of a JavaServer Page

The Java code in the page includes :

- Directives – these provide global information to the page, for example, import statements, the page for error handling or whether the page is part of a session. In the above example we set the script language to Java.
- Declaratives - these are for page-wide variable and method declarations.
- Scriptlets - the Java code embedded in the page.
- Expressions - formats the expression as a string for inclusion in the output of the page.

We will meet the last JSP element type, actions, soon. These elements follow an XML-like syntax, and perform a function behind the scenes. They provide the means to totally separate presentation from logic. A good example is <jsp:useBean .../> which finds or creates an instance of a bean with the given scope and name. With the tag extension mechanism to be introduced in JSP 1.1, you'll be able to define similar action tags and put their functionality in a tag library.
A JSP directive is a statement that gives the JSP engine information for the page that follows. The general syntax of a JSP directive is <%@ directive { attribute=”value” } %>, where the directive may have a number of (optional) attributes. Each directive has an optional XML equivalent, but these are intended for future JSP tools, so we won't consider them here.
Possible directives in JSP 1.0 are :

- Page – information for that page
- Include – files to be included verbatim
- Taglib – the URI for a library of tags that you'll use in the page (unimplemented at the time of writing)

As is to be expected, the page directive has many possible attributes. Specifying these is optional, as the mandatory ones have default values.

RESULT: This experiment introduces the concept of Java Server Pages

Experiments Beyond Syllabus

Title:- Techniques used for white box testing.

Objective :- To get familiar with white box testing

S/W Requirement :- Rational Rose

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

White box testing focuses on the internal functioning of the product. For this different procedures are tested. White box testing tests the following

- Loops of the procedure
- Decision points
- Execution paths

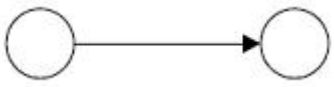
For performing white box testing, basic path testing technique is used. We will illustrate how to use this technique, in the following section.

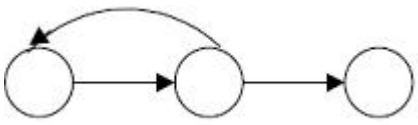
Basis Path Testing

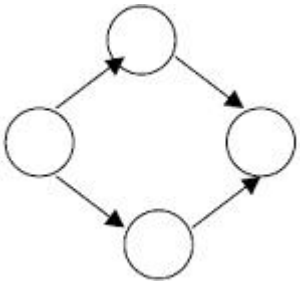
Basic path testing a white box testing technique .It was proposed by Tom McCabe. These tests guarantee to execute every statement in the program at least one time during testing. Basic set is the set of all the execution path of a procedure.

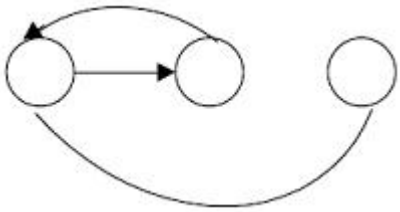
Flow graph Notation

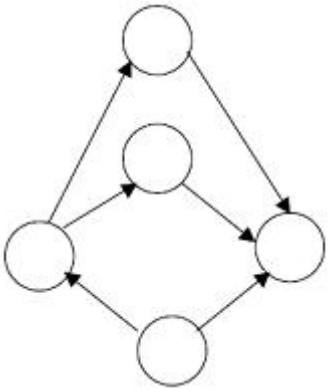
Before basic path procedure is discussed, it is important to know the simple notation used for the repres4enttation of control flow. This notation is known as flow graph. Flow graph depicts control flow and uses the following constructs.
These individual constructs combine together to produce the flow graph for a particular procedure

Sequence - 

Until - 

If - 

While - 

Case - 

Basic terminology associated with the flow graph

Node: Each flow graph node represents one or more procedural statements. Each node that contains a condition is called a predicate node.

Edge: Edge is the connection between two nodes. The edges between nodes represent flow of control. An edge must terminate at a node, even if the node does not represent any useful procedural statements.

Region: A region in a flow graph is an area bounded by edges and nodes. Cyclomatic complexity: Independent path is an execution flow from the start point to the end point. Since a procedure contains control statements, there are various execution paths depending upon decision taken on the control statement. So Cyclomatic complexity provides the number of such execution independent paths. Thus it provides a upper bound for number of tests that must be produced because for each independent path, a test should be conducted to see if it is actually reaching the end point of the procedure or not.

Cyclomatic Complexity

Cyclomatic Complexity for a flow graph is computed in one of three ways:

1. The numbers of regions of the flow graph correspond to the Cyclomatic complexity.

1. Cyclomatic complexity, V(G), for a flow graph G is defined as

   V(G) = E – N + 2

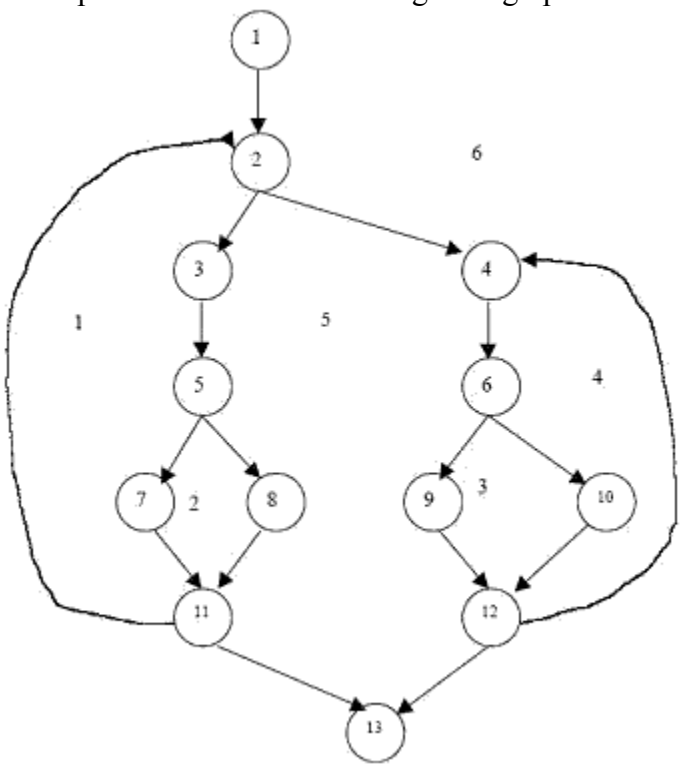
   where E is the number of flow graph edges and N is the number of flow graph nodes.

2. Cyclomatic complexity, V(G), for a graph flow G is also defined as

   V(G) = P + 1

   Where P is the number of predicate nodes contained in the flow graph G.

Example: Consider the following flow graph



Region, R= 6
Number of Nodes = 13
Number of edges = 17
Number of Predicate Nodes = 5
Cyclomatic Complexity, V(C) :
V(C) = R = 6;
Or
V(C) = Predicate Nodes + 1
=5+1 =6
Or
V(C)= E-N+2
= 17-13+2

Deriving Test Cases

The main objective of basic path testing is to derive the test cases for the procedure under test. The process of deriving test cases is following

1. From the design or source code, derive a flow graph.

1. Determine the Cyclomatic complexity, V(G) of this flow graph using any of the formula discussed above.

   · Even without a flow graph, V(G) can be determined by counting the number of conditional statements in the code and adding one to it.

2. Prepare test cases that will force execution of each path in the basis set.

   · Each test case is executed and compared to the expected results.

Graph Matrices

Graph matrix is a two dimensional matrix that helps in determining the basic set. It has rows and columns each equal to number of nodes in flow graph. Entry corresponding to each node-node pair represents an edge in flow graph. Each edge is represented by some letter (as given in the flow chart) to distinguish it from other edges. Then each edge is provided with some link weight, 0 if there is no connection and 1 if there is connection.

For providing weights each letter is replaced by 1 indicating a connection. Now the graph matrix is called connection matrix. Each row with two entries represents a predicate node. Then for each row sum of the entries is obtained and 1 is subtracted from it. Now the value so obtained for each row is added and 1 is again added to get the cyclomatic complexity.

Once the internal working of the different procedure are tested, then the testing for the overall functionality of program structure is tested.

RESULT: This experiment introduces the various techniques used for white box testing

Experiments Beyond Syllabus

Title:- Techniques used for Black box testing.

Objective :- To get familiar with Black box testing

S/W Requirement :- Rational Rose

H/W Requirement :-

•**Processor** – Any suitable Processor e.g. Celeron
•**Main Memory** - 128 MB RAM
•**Hard Disk** – minimum 20 GB IDE Hard Disk
•**Removable Drives**–1.44 MB Floppy Disk Drive
–52X IDE CD-ROM Drive
•PS/2 HCL **Keyboard and Mouse**

Method:-

Black box testing test the overall functional requirements of product. Input are supplied to product and outputs are verified. If the outputs obtained are same as the expected ones then the product meets the functional requirements. In this approach internal procedures are not considered. It is conducted at later stages of testing. Now we will look at black box testing technique.
Black box testing uncovers following types of errors.

1. Incorrect or missing functions
2. Interface errors
3. External database access
4. Performance errors
5. Initialization and termination errors.

The following techniques are employed during black box testing

Equivalence Partitioning
In equivalence partitioning, a test case is designed so as to uncover a group or class of error. This limits the number of test cases that might need to be developed otherwise. Here input domain is divided into classes or group of data. These classes are known as equivalence classes and the process of making equivalence classes is called equivalence partitioning. Equivalence classes represent a set of valid or invalid states for input condition.

An input condition can be a range, a specific value, a set of values, or a boolean value. Then depending upon type of input equivalence classes is defined. For defining equivalence classes the following guidelines should be used.

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific value, then one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, then one valid and one invalid equivalence class are defined.

4. If an input condition is Boolean, then one valid and one invalid equivalence class are defined.

For example, the range is say, 0 < count < Max1000. Then form a valid equivalence class with that range of values and two invalid equivalence classes, one with values less than the lower bound of range (i.e., count < 0) and other with values higher than the higher bound(count > 1000).

Boundary Value Analysis

It has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values. These values often lie on the boundary of the equivalence class. Test cases that have values on the boundaries of equivalence classes are therefore likely to be error producing so selecting such test cases for those boundaries is the aim of boundary value analysis. In boundary value analysis, we choose input for a test case from an equivalence class, such that the input lies at the edge of the equivalence classes. Boundary values for each equivalence class, including the equivalence classes of the output, should be covered. Boundary value test cases are also called "extreme cases".

Hence, a boundary value test case is a set of input data that lies on the edge or boundary of a class of input data or that generates output that lies at the boundary of a class of output data.

In case of ranges, for boundary value analysis it is useful to select boundary elements of the range and an invalid value just beyond the two ends (for the two invalid equivalence classes. For example, if the range is 0.0 <= x <= 1.0, then the test cases are 0.0,1.0for valid inputs and –0.1 and 1.1 for invalid inputs.

For boundary value analysis, the following guidelines should be used:

For input ranges bounded by a and b, test cases should include values a and b and just above and just below a and b respectively.

If an input condition specifies a number of values, test cases should be developed to exercise the minimum and maximum numbers and values just above and below these limits.

If internal data structures have prescribed boundaries, a test case should be designed to exercise the data structure at its boundary.

Now we know how the testing for software product is done. But testing software is not an easy task since the size of software developed for the various systems is often too big. Testing needs a specific systematic procedure, which should guide the tester in performing different tests at correct time. This systematic procedure is testing strategies, which should be followed in order to

test the system developed thoroughly. Performing testing without some testing strategy would be very cumbersome and difficult.

RESULT: This experiment introduces the various techniques used for black box testing