DISCLAIMER

THIS DOCUMENT DOES NOT CLAIM ANY ORIGINALITY AND CANNOT BE USED AS A SUBSTITUTE FOR PRESCRIBED TEXTBOOKS. THE INFORMATION PRESENTED HERE IS MERELY A COLLECTION BY THE COMMITTEE MEMBERS FOR THEIR RESPECTIVE TEACHING ASSIGNMENTS. VARIOUS TEXT BOOKS AS WELL AS FREELY AVAILABLE MATERIAL FROM INTERNET WERE CONSULTED FOR PREPARING THIS DOCUMENT. THE OWNERSHIP OF THE INFORMATION LIES WITH THE RESPECTIVE AUTHORS OR INSTITUTIONS.

- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

1. CLASSICAL WATERFALL MODEL

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases as shown in fig.2.1:

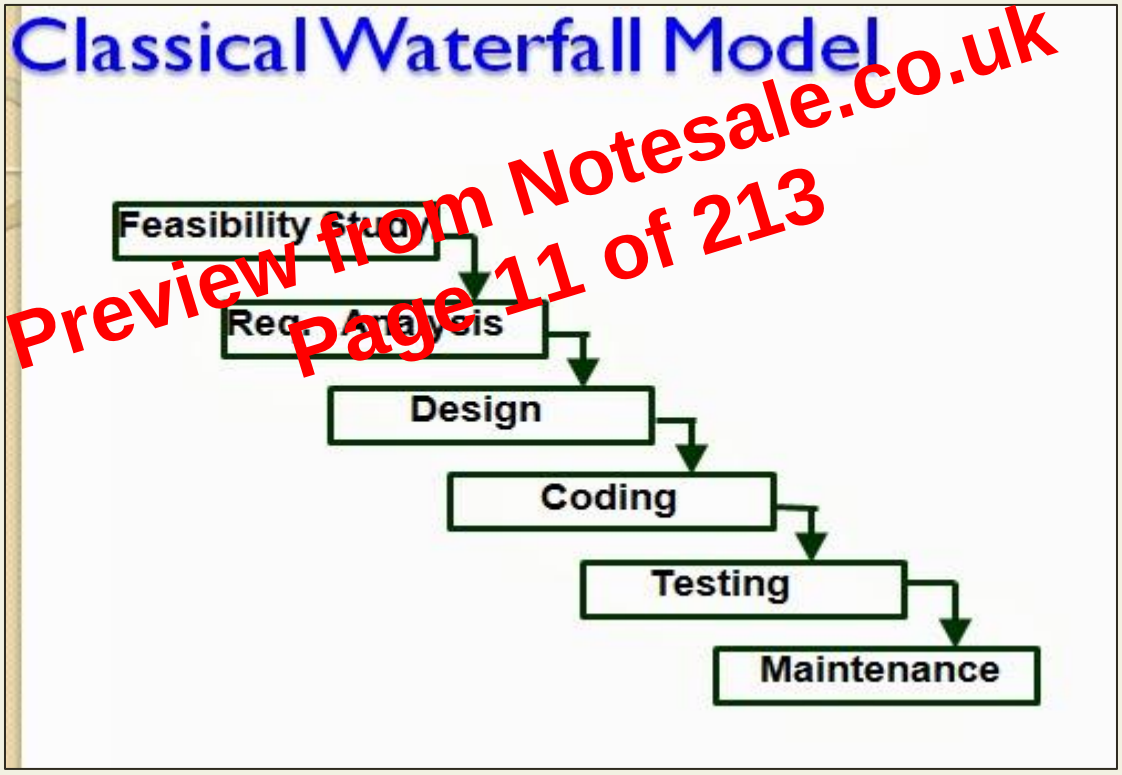

Fig 2.1: Classical Waterfall Model

Feasibility study - The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

- At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

- After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.

- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

Requirements analysis and specification: - The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f_i\}$. The functional view of the system is shown in fig. 5.1. Each function f_i of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.

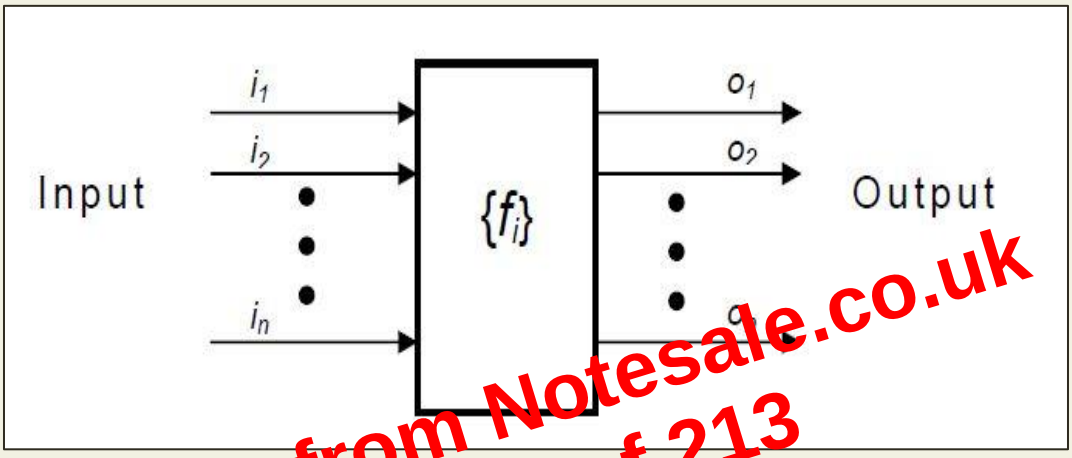

Fig. 5.1: View of a system performing a set of functions

Nonfunctional requirements:-

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Goals of implementation:-

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Model-oriented vs. property-oriented approaches

Formal methods are usually classified into two broad categories – model – oriented and property – oriented approaches. In a model-oriented style, one defines a system's behavior directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc.

In the property-oriented style, the system's behavior is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy.

> **Example:-**
>
> Let us consider a simple producer/consumer example. In a property-oriented style, it is probably started by listing the properties of the system like: the consumer can start consuming only after the producer has produced an item; the producer starts to produce an item only after the consumer has consumed the last item, etc. A good example of a producer-consumer problem is CPU-Printer coordination. After processing of data, CPU outputs characters to the buffer for printing. Printer, on the other hand, reads characters from the buffer and prints them. The CPU is constrained by the capacity of the buffer, whereas the printer is constrained by an empty buffer. Examples of property-oriented specification styles are axiomatic specification and algebraic specification.

In a model-oriented approach, we start by defining the basic operations, p (produce) and c (consume). Then we can state that $S1 + p \rightarrow S$, $S + c \rightarrow S1$. Thus the model-oriented approaches essentially specify a program by writing another, presumably simpler program. Examples of popular model-oriented specification techniques are Z, CSP, CCS, etc.

Model-oriented approaches are more suited to use in later phases of life cycle because here even minor changes to a specification may lead to drastic changes to the entire specification. They do not support logical conjunctions (AND) and disjunctions (OR).

Property-oriented approaches are suitable for requirements specification because they can be easily changed. They specify a system as a conjunction of axioms and you can easily replace one axiom with another one.

Operational Semantics

Informally, the operational semantics of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a single run of the system and how the runs are grouped together to describe the behavior of the system. Some commonly used operational semantics are as follows:

Linear Semantics:-

In this approach, a run of a system is described by a sequence (possibly infinite) of events or states. The concurrent activities of the system are represented by non-deterministic interleavings of the automatic actions. For example, a concurrent activity a‖b is represented by the set of

- The mathematical basis of the formal methods facilitates automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.
- Formal specifications can be executed to obtain immediate feedback on the features of the specified system. This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a "toy" working model of a system that can provide immediate feedback on the behavior of the specified system, and is especially useful in checking the completeness of specifications.

Limitations of formal requirements specification

It is clear that formal methods provide mathematically sound frameworks within large, complex systems can be specified, developed and verified in a systematic rather than in an ad hoc manner. However, formal methods suffer from several shortcomings, some of which are the following:

- Formal methods are difficult to learn and use.
- The basic incompleteness results of first-order logic suggest that it is impossible to check absolute correctness of systems using theorem proving techniques.

- Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulas is difficult to comprehend.

Axiomatic Specification

In axiomatic specification of a system, first-order logic is used to write the pre and post-conditions to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function completes execution for the function to be considered to have executed successfully. Thus, the post-conditions are essentially constraints on the results produced for the function execution to be considered successful.

There are seven types of cohesion, namely –

- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.
- **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely -

- **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

SOFTWARE ANALYSIS & DESIGN TOOLS

Software analysis and design includes all activities, which help the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable and understandable documents, to which a computer has nothing to do.

Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code.

Let us see few analysis and design tools used by software designers:

Data Flow Diagram

Data flow diagram is a graphical representation of data flow in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

Types of DFD

Data Flow Diagrams are either Logical or Physical.

- **Logical DFD** - This type of DFD concentrates on the system process and flow of data in the system. For example in a Banking software system, how data is moved between different entities.
- **Physical DFD** - This type of DFD shows how the data flow is actually implemented in the system. It is more specific and close to the implementation.

Data Dictionary

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data **grossPay** consists of the components regularPay and overtimePay.

grossPay = regularPay + overtimePay

For the smallest units of data items, the data dictionary lists their name and their type. Composite data items can be defined in terms of primitive data items using the following data definition operators:

+: denotes composition of two data items, e.g. **a+b** represents data a and **b**.

[,,]: represents selection, i.e. any one of the data items listed in the brackets can occur. For example, **[a,b]** represents either **a** occurs or **b** occurs.

(): the contents inside the bracket represent optional data which may or may not appear. e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.

{}: represents iterative data definition, e.g. **{name}5** represents five **name** data. **{name}*** represents zero or more instances of **name** data.

=: represents equivalence, e.g. **a=b+c** means that **a** represents **b** and **c**.

/* */: Anything appearing within **/*** and ***/** is considered as a comment.

Example 1: Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many bubbles, i.e. more than 7 bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

Numbering of Bubbles:-

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD by its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Example:-

A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 carat gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated.

The context diagram for this problem is shown in fig. 10.5, the level 1 DFD in fig. 10.6, and the level 2 DFD in fig. 10.7.

- The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its sub-functions and we have to use subjective judgment to carry out decomposition.

- It is usually difficult to identify the different modules of the software from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

Transform Analysis

Transform analysis identifies the primary functional components (modules) and the high level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:

- Input
- Logical processing
- Output

The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called the central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component for the ***central transform***, and the ***afferent*** and ***efferent*** branches.

These are drawn below a root module, which would invoke these modules. Identifying the highest level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them are not central transforms. Processes which sort input or filter data from it are. The first level structure chart is produced by representing each input and output unit as boxes and each central transform as a single box. In the third step of transform analysis, the structure chart is refined by adding sub-functions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process, identifying customer modules, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

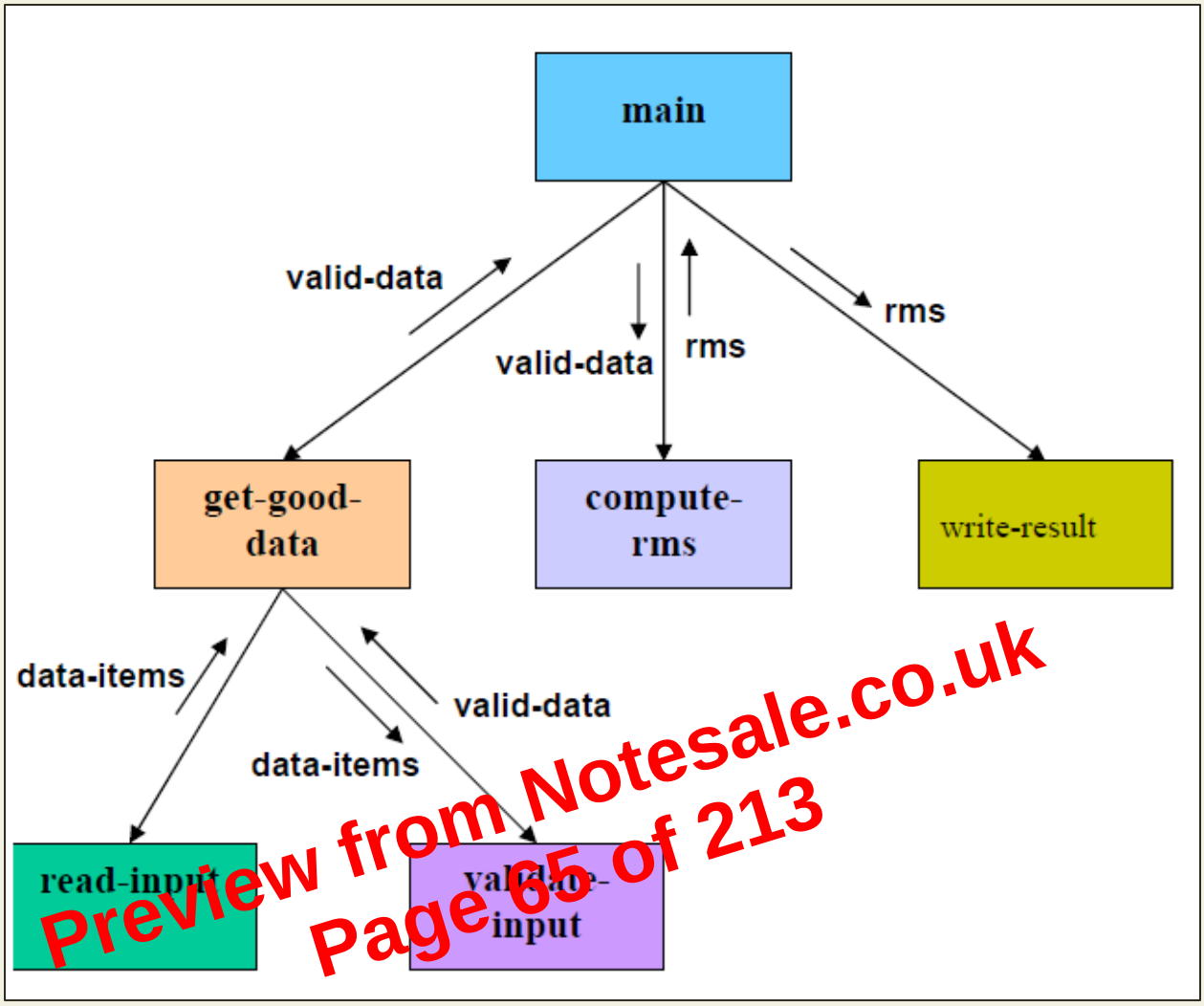

Fig. 11.2: Structure Chart

Transaction Analysis

A transaction allows the user to perform some meaningful piece of work. Transaction analysis is useful while designing transaction processing programs. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centered system which is characterized by similar processing steps for each data item. Each different way in which input data is handled is a transaction. A simple way to identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transaction may not require any input data. These transactions can be identified from the experience of solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction a module. Every transaction carries a tag, which identifies its type.

USE CASE DIAGRAM

Use Case Model

The use case model for any system consists of a set of "use cases". Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: "What the users can do using the system?" Thus for the Library Information System (LIS), the use cases could be**:**

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user's point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.

Purpose of use cases

The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

Representation of Use Cases

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

The different users of the system are represented by using the stick person icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case. Both the human users and the external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

Example 1:

Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

The use case model for the Tic-tac-toe problem is shown in fig. 13.1. This software has only one use case "play move". Note that the use case "get-user-move" is not used here. The name "get-user-move" would be inappropriate because the use cases should be named from the user's perspective.

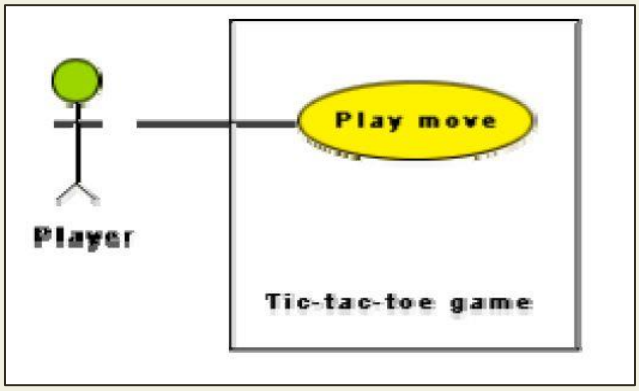

Fig. 13.1: Use case model for tic-tac-toe game

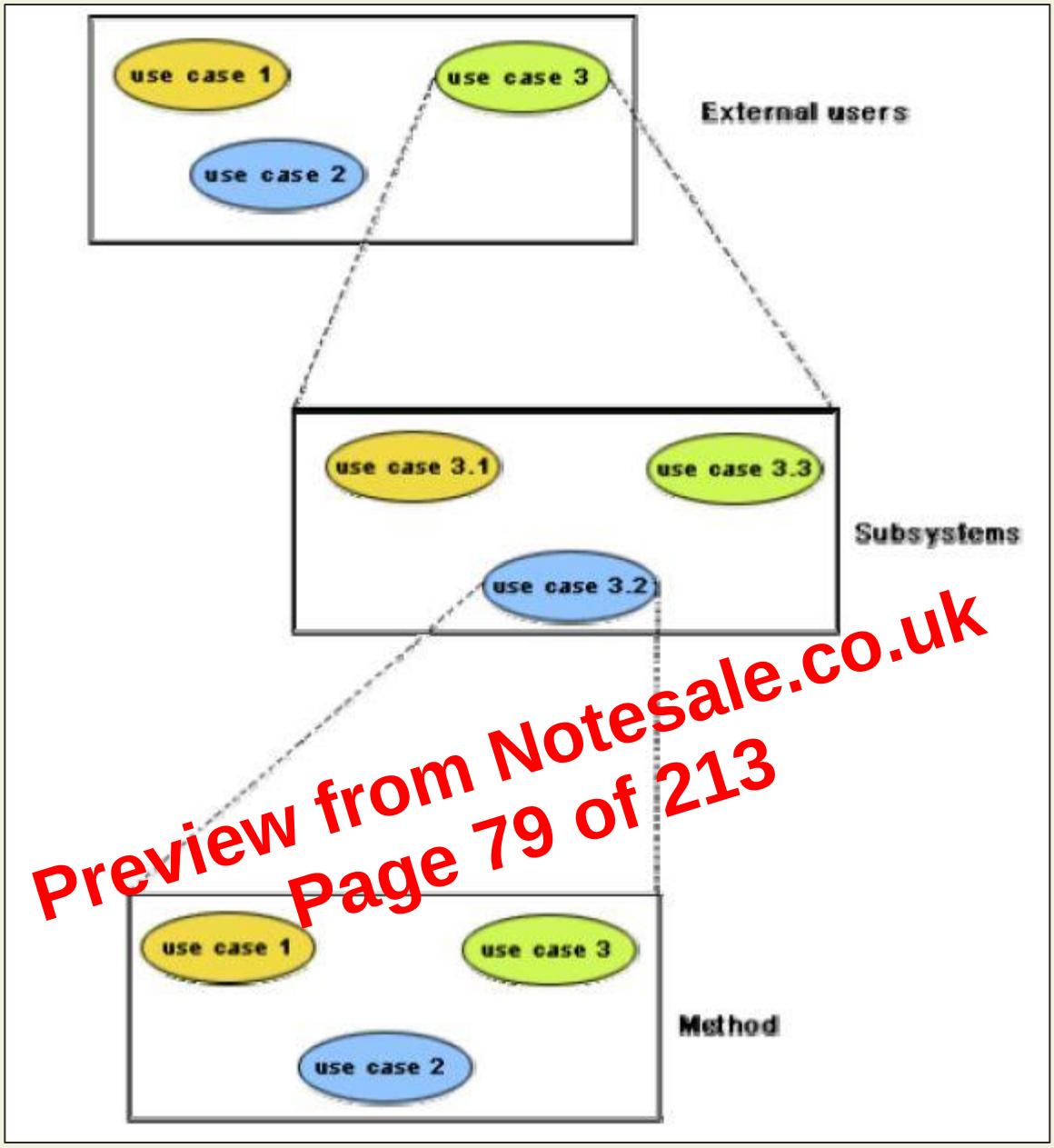

Fig. 13.7: Hierarchical organization of use cases

whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.

- In addition, in composition, the whole has the responsibility for the disposition of all its parts, i.e. for their creation and destruction.

  - in general, the lifetime of parts and composite coincides
  - parts with non-fixed multiplicity may be created after composite itself
  - parts might be explicitly removed before the death of the composite

  For example, when a **Frame** is created, it has to be attached to an enclosing **Window**. Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

Inheritance vs. Aggregation/Composition

- Inheritance describes *'is a' / 'is a kind of'* relationship between classes (base class - derived class) whereas aggregation describes *'has a'* relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation "cash payment *is a kind of* payment" is modeled using inheritance; "purchase order has a few items" is modeled using aggregation.

- Inheritance is used to model a "generic-specific" relationship between classes whereas aggregation/composition is used to model a "whole-part" relationship between classes.
- Inheritance means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse, i.e. wherever we could use instances of 'payment' in the system, we could substitute it with instances of 'cash payment', but the reverse cannot be done.
- Inheritance is defined statically. It cannot be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.

  For example, consider this situation in a business system. A **BusinessPartner** might be a **Customer** or a **Supplier** or both. Initially we might be tempted to model it as in Fig 14.4(a). But in fact, during its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we prefer aggregation instead (see Fig 14.4(b). Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a "C**ustomer_Supplier**" if it has both. Here, we have only two types. Hence, we are able to model it as inheritance. But what if there were several different types and combinations thereof? The inheritance tree would be absolutely incomprehensible.
  Also, the aggregation model allows the possibility for a business partner to be neither - i.e. has neither a customer nor a supplier object aggregated with it.

CODING

Coding- The objective of the coding phase is to transform the design of a system into code in a high level language and then to unit test this code. The programmers adhere to standard and well defined style of coding which they call their coding standard. The main advantages of adhering to a standard style of coding are as follows:

- A coding standard gives uniform appearances to the code written by different engineers
- It facilitates code of understanding.
- Promotes good programming practices.

For implementing our design into a code, we require a good high level language. A programming language should have the following features:

Characteristics of a Programming Language

- **Readability:** A good high-level language will allow programs to be written in some ways that resemble a quasi-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting.
- **Portability:** High-level languages, being essentially machine independent, should be able to develop portable software.
- **Generality:** Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.
- **Brevity:** Language should have the ability to implement the algorithm with less amount of code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.
- **Error checking:** Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.
- **Cost:** The ultimate cost of a programming language is a function of many of its characteristics.

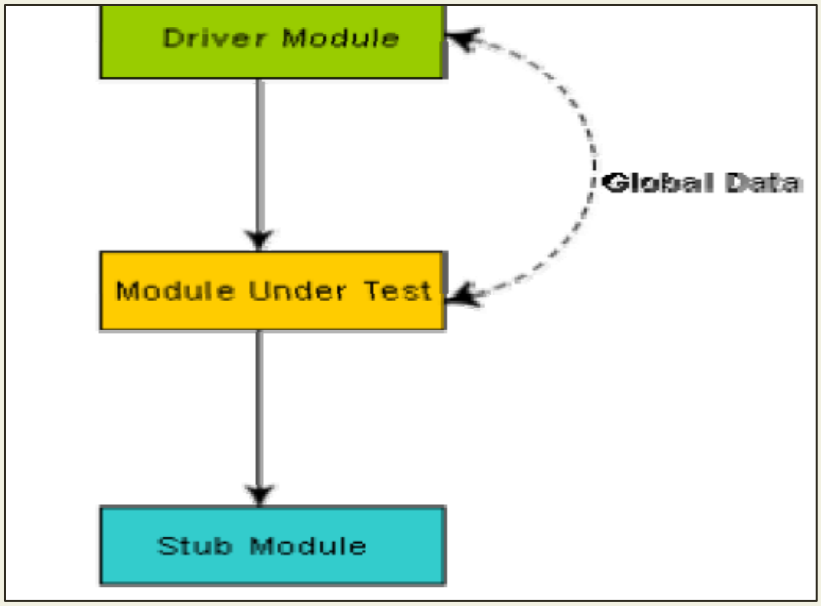

Fig. 19.1: Unit testing with the help of driver and stub modules

Black Box Testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches to designing black box test cases.

- Equivalence class partitioning
- Boundary value analysis

Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

LECTURE NOTE 22

DEBUGGING, INTEGRATION AND SYSTEM TESTING

Need for Debugging

Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix them up are known as debugging.

Debugging Approaches

The following are some of the approaches popularly adopted by programmers for debugging.

Brute Force Method:

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

Backtracking:

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

Cause Elimination Method:

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program Slicing:

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

Debugging Guidelines

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

Program Analysis Tools

A program analysis tool means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc. We can classify these into two broad categories of program analysis tools:

- Static Analysis tools
- Dynamic Analysis tools
- Static program analysis tools

Static Analysis Tool is also a program analysis tool. It assesses and computes various characteristics of a software product without executing it. Typically, static analysis tools analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold. The structural properties that are usually analyzed are:

- Whether the coding standards have been adhered to?
- Certain programming errors such as uninitialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

Code walk throughs and code inspections might be considered as static analysis methods. But, the term static program analysis is used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

Dynamic program analysis tools - Dynamic program analysis techniques require the program to be executed and its actual behavior recorded. A dynamic analyzer usually instruments the code (i.e. adds additional statements in the source code to collect program execution traces). The instrumented code when executed allows us to record the behavior of the software for different test cases. After the software has been tested with its full test suite and its behavior recorded, the

Top-Down Integration Testing

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

Mixed Integration Testing

A mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

Phased Vs. Incremental Testing

The different integration testing strategies are either phased or incremental. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partial system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system in the incremental testing approach since it is known that the error is caused by addition of a single module. In fact, big bang testing is a degenerate case of the phased integration testing approach.

System testing

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

- **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the developing organization.
- **Beta testing.** Beta testing is the system testing performed by a select group of friendly customers.
- **Acceptance Testing.** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

Reasons for software reliability being difficult to measure

The reasons why software reliability is difficult to measure can be summarized as follows:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.
- Hardware reliability vs. software reliability differs.

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase). The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 26.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics “bath tub” shape. On the other hand, for software the failure rate is at it’s highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.

A quality system consists of the following:

Managerial Structure and Individual Responsibilities- A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

Quality System Activities- The quality system activities encompass the following:

- auditing of projects
- review of the quality system
- development of standards, procedures, and guidelines, etc.
- production of reports for the top management summarizing the effectiveness of the quality system in the organization.

Evolution of Quality Management System

Quality systems have rapidly evolved over the last five decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality systems of organizations have undergone through four stages of evolution as shown in the fig. 28.1. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products. The next breakthrough in quality systems was the development of quality assurance principles.

The basic premise of modern quality assurance is that if an organization's processes are good and are followed rigorously, then the products are bound to be of good quality. The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization. From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance (as shown in fig. 28.1).

Level 4: Managed - At this level, the focus is on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc. Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimizing - At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement. For example, if from an analysis of the process measurement results, it was found that the code reviews were not very effective and a large number of errors were detected only during the unit testing, then the process may be fine-tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated in to the process. Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies. Such an organization identifies the best software engineering practices and innovations which may be tools, methods, or processes. These best practices are transferred throughout the organization.

Key process areas (KPA) of a software organization

Except for SEI CMM level 1, each maturity level is characterized by several Key Process Areas (KPAs) that includes the areas an organization should focus to improve its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig. 29.1.

PSP2 introduces defect management via the use of checklists for code and design reviews. The checklists are developed from gathering and analyzing defect data earlier projects.

Six Sigma

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

Six Sigma at many organizations simply means striving for near perfection. Six Sigma is a disciplined, data-driven approach to eliminate defects in any process – from manufacturing to transactional and 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. 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 the Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of Six Sigma improvement projects. This is accomplished through the use of two Six Sigma sub-methodologies – DMAIC and DMADV. The Six Sigma DMAIC process (define, measure, analyze, improve, control) is an improvement system for existing processes failing below specification and looking for incremental improvement. The Six Sigma DMADV process (define, measure, analyze, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Both Six Sigma processes are executed by Six Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts.

Many frameworks exist for implementing the Six Sigma methodology. Six Sigma Consultants all over the world have also developed proprietary methodologies for implementing Six Sigma quality, based on the similar change management philosophies and applications of tools.

- **Duration**: How long is it going to take to complete development?
- **Effort**: How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

- Scheduling manpower and other resources.
- Staff organization and staffing plans.
- Risk identification, analysis, and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

Precedence ordering among project planning activities

Different project related estimates done by a project manager have already been discussed. Fig. 30.1 shows the order in which important project planning activities may be undertaken. From fig. 30.1 it can be easily observed that size estimation is the first activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.

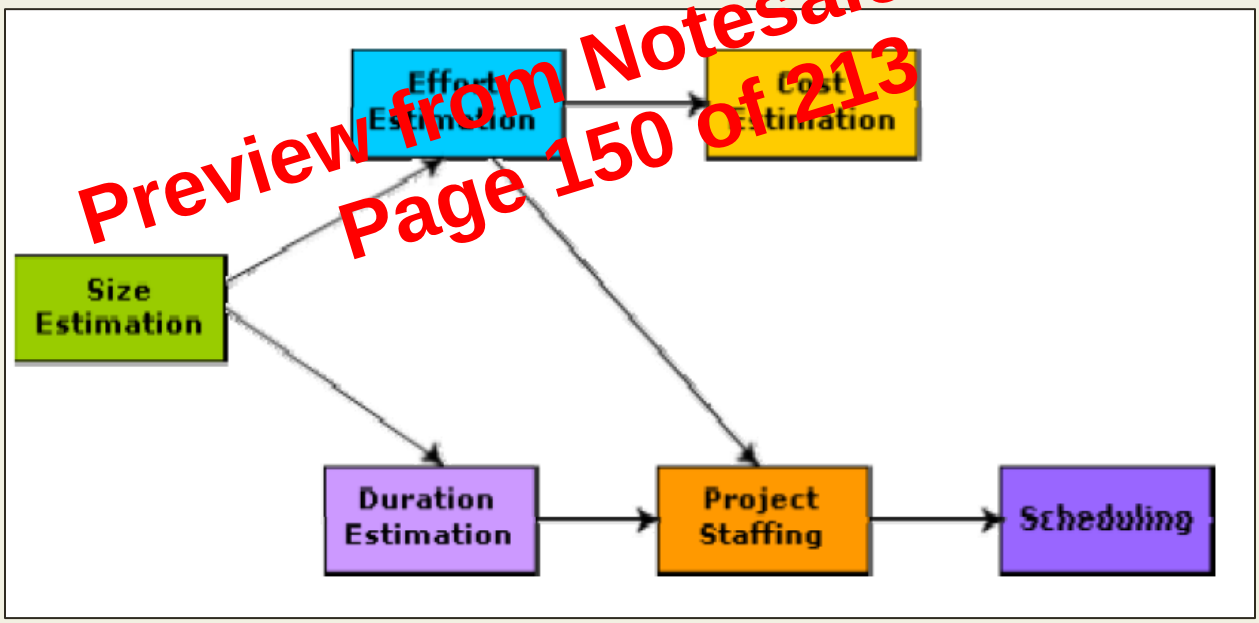

Fig. 30.1: Precedence ordering among planning activities

Sliding Window Planning

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. However,

experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

Delphi Cost Estimation

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

LECTURE NOTE 32

HEURISTIC TECHNIQUES

Heuristic techniques assume that the relationships among the different project parameters can be modeled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and the multi variable model.

Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

$$\text{Estimated Parameter} = c_1 * e^{d_1}$$

In the above expression, e is the characteristic of the software which has already been estimated (independent variable). *Estimated Parameter* is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc. c_1 and d_1 are constants. The values of the constants c_1 and d_1 are usually determined using data collected from past projects (historical data). The basic COCOMO model is an example of single variable cost estimation model.

A multivariable cost estimation model takes the following form:

$$\text{Estimated Resource} = c_1 * e_1^{d_1} + c_2 * e_2^{d_2} + \ldots$$

Where e_1, e_2, … are the basic (independent) characteristics of the software already estimated, and c_1, c_2, d_1, d_2, … are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modeled by the constants c_1, c_2, d_1, d_2, … . Values of these constants are usually determined from historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead's software science is an example of an analytical technique. Halstead's software science can be used to derive some interesting results starting with a few

the program. Halstead's software science provides gross estimation of properties of a large collection of software, but extends to individual cases rather inaccurately.

Example:

Let us consider the following C program:

```
main( )
{
int a, b, c, avg;
scanf("%d %d %d", &a, &b, &c);
avg = (a+b+c)/3;
printf("avg = %d", avg);
}
```

The unique operators are:

main,(),{},int,scanf,&,",",";",=,+,/, printf

The unique operands are:

a, b, c, &a, &b, &c, a+b+c, avg, 3,
"%d %d %d", "avg = %d"

Therefore,

$\eta_1 = 12$, $\eta_2 = 11$

Estimated Length = (12*log12 + 11*log11)
= (12*3.58 + 11*3.45)
= (43+38) = 81

Volume = **Length*log(23)**
= 81*4.52
= 366

$$\text{or, } K_1/K_2 = t_{d2}^4 / t_{d1}^4$$

$$\text{or, } K \propto 1/t_d^4$$

$$\text{or, cost} \propto 1/t_d$$

(as project development effort is equally proportional to project development cost)

From the above expression, it can be easily observed that when the schedule of a project is compressed, the required development effort as well as project development cost increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in substantial penalty of human effort as well as development cost. For example, if the estimated development time is 1 year, then in order to develop the product in 6 months, the total effort required to develop the product (and hence the project cost) increases 16 times.

Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different engineers.

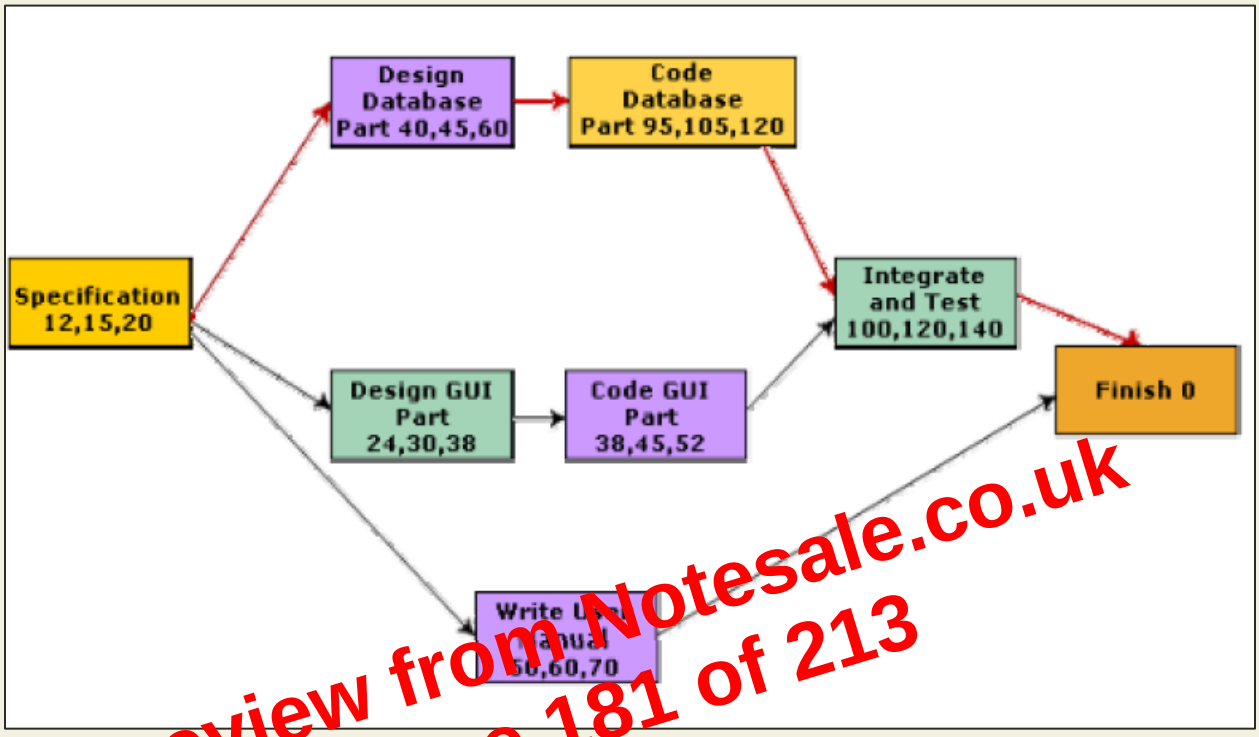

Fig. 36.4: PERT chart representation of the MIS problem

COMPUTER AIDED SOFTWARE ENGINEERING

CASE tool and its scope

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool means any tool used to automate some activity associated with software development. Many CASE tools are available. Some of these CASE tools assist in phase related tasks such as specification, structured analysis, design, coding, testing, etc.; and others to non-phase activities such as project management and configuration management.

Reasons for using CASE tools

The primary reasons for using a CASE tool are:

- To increase productivity
- To help produce better quality software at lower cost

CASE environment

Although individual CASE tools are useful, the true power of a tool set can be realized only when these set of tools are integrated into a common framework or environment. CASE tools are characterized by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software development artifacts. This central repository is usually a data dictionary containing the definition of all composite and elementary

data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilities the automation of the step-by-step methodologies for software development. A schematic representation of a CASE environment is shown in fig. 39.1.

- CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced and therefore chances of inconsistent documentation is reduced to a great extent.
- CASE tools take out most of the drudgery in a software engineer's work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.
- CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.
- Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

Requirements of a prototyping CASE tool

Prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on. The important features of a prototyping CASE tool are as follows:

- Define user interaction
- Define the system control flow
- Store and retrieve data required by the system
- Incorporate some processing logic

Features of a good prototyping CASE tool

There are several stand-alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype. A good prototyping tool should support the following features:

- Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.
- If possible, it should be able to integrate with external user defined modules written in C or some popular high level programming languages.
- The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.

LECTURE NOTE 41

REUSE APPROACH

Components Classification

Components need to be properly classified in order to develop an effective indexing and storage scheme. Hardware reuse has been very successful. Hardware components are classified using a multilevel hierarchy. At the lowest level, the components are described in several forms: natural language description, logic schema, timing information, etc. The higher the level at which a component is described, the more is the ambiguity. This has motivated the Prieto-Diaz's classification scheme.

Prieto-Diaz's classification scheme: Each component is best described using a number of different characteristics or facets. For example, objects can be classified using the following:

Searching- The domain repository may contain thousands of reuse items. A popular search technique that has proved to be very effective is one that provides a web interface to the repository. Using such a web interface, one would search an item using an approximate automated search using key words, and then from these results do a browsing using the links provided to look up related items. The approximate automated search locates products that appear to fulfill some of the specified requirements. The items located through the approximate search serve as a starting point for browsing the repository. These serve as the starting point for browsing the repository. The developer may follow links to other products until a sufficiently good match is found. Browsing is done using the keyword-to-keyword, keyword-to-product, and product-to-product links. These links help to locate additional products and compare their detailed attributes. Finding a satisfactorily item from the repository may require several locations of approximate search followed by browsing. With each iteration, the developer would get a better understanding of the available products and their differences. However, we must remember that the items to be searched may be components, designs, models, requirements, and even knowledge.

Repository maintenance - Repository maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search. The software industry is always trying to implement something that has not been quite done before. As patterns requirements emerge, new reusable components are identified, which may ultimately become more or less the standards. However, as technology advances, some components which are still reusable, do not fully address the current requirements. On the other hand, restricting reuse to highly mature components, sacrifices one of that creates potential reuse opportunity. Making a product available before it has been thoroughly