SOFTWARE ENGINEERING

LECTURE-11

03/02/21

SOFTWARE ENGINEERING
LECTURE-23

PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

EMPIRICAL ESTIMATION TECHNIQUES

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: Expert judgment technique and Delphi cost estimation.

EXPERT JUDGMENT TECHNIQUE

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part. A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of 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

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:

Estimated Parameter = c1 * ed 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. c1 and d1 are constants. The values of the constants c1 and d1 are usually determined using data collected from past projects (historical data). The basic COCOMO model is an example of single variable cost 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 simple assumptions. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

Halstead's Software Science- An Analytical Technique

Halstead's software science is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for over all program length, potential minimum value, actual volume, effort, and development time. For a given program, let:

η1 be the number of unique operators used in the program,

η2 be the number of unique operands used in the program,

N1 be the total number of operators used in the program,

N2 be the total number of operands used in the program

LENGTH AND VOCABULARY

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length N = N1 +N2. Halstead's definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notation of the program length as the total number of tokens used in the program. The program vocabulary is the number of unique operators and operands used in the program. Thus, program vocabulary $\eta = \eta 1 + \eta 2$

PROGRAM VOLUME

The length of a program (i.e. the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used. Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N \log_2 \eta$$

POTENTIAL MINIMUM VOLUME

THE POTENTIAL MINIMUM VOLUME V* IS DEFINED AS THE VOLUME OF MOST SUCCINCT PROGRAM IN WHICH A PROBLEM CAN BE CODED. THE MINIMUM VOLUME IS OBTAINED WHEN THE PROGRAM CAN BE EXPRESSED USING A SINGLE SOURCE CODE INSTRUCTION., SAY A FUNCTION CALL LIKE FOO( ) ;. IN OTHER WORDS, THE VOLUME IS BOUND FROM BELOW DUE TO THE FACT THAT A PROGRAM WOULD HAVE AT LEAST TWO OPERATORS AND NO LESS THAN THE REQUISITE NUMBER OF OPERANDS. THUS, IF AN ALGORITHM OPERATES ON INPUT AND OUTPUT DATA D1, D2, … DN, THE MOST SUCCINCT PROGRAM WOULD BE F(D1, D2, … DN); FOR WHICH $H1 = 2$, $H2 = N$.

THEREFORE, $V^* = (2 + H2)LOG2(2 + H2)$.

THE PROGRAM LEVEL L IS GIVEN BY $L = V^*/V$. THE CONCEPT OF PROGRAM LEVEL L IS INTRODUCED IN AN ATTEMPT TO MEASURE THE LEVEL OF ABSTRACTION PROVIDED BY THE PROGRAMMING LANGUAGE. USING THIS DEFINITION, LANGUAGES CAN BE RANKED INTO LEVELS THAT ALSO APPEAR INTUITIVELY CORRECT. THE ABOVE RESULT IMPLIES THAT THE HIGHER THE LEVEL OF A LANGUAGE, THE LESS EFFORT IT TAKES TO DEVELOP A PROGRAM USING THAT LANGUAGE. THIS RESULT AGREES WITH THE INTUITIVE NOTION THAT IT TAKES MORE EFFORT TO DEVELOP A PROGRAM IN ASSEMBLY LANGUAGE THAN TO DEVELOP A PROGRAM IN A HIGH-LEVEL LANGUAGE TO SOLVE A PROBLEM.

EFFORT AND TIME

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to develop the code.

Thus, effort E = V/L, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the program.

Thus, the programming effort $E = V^2/V^*$ (since $L = V^*/V$) varies as the square of the volume. Experience shows that E is well correlated to the effort needed for maintenance of an existing program.

The programmer's time T = E/S, where S the speed of mental discriminations. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18

LENGTH ESTIMATION

Even though the length of a program can be found by calculating the total number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc. can be determined even before the start of any programming activity. His method is summarized below

$$N = \eta 1 \log 2 \eta 1 + \eta 2 \log 2 \eta 2$$

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

SPECIFIC INSTRUCTIONAL OBJECTIVES

At the end of this lesson the student would be able to:

- Differentiate among organic, semidetached and embedded software projects.
- Explain basic COCOMO.
- Differentiate between basic COCOMO model and intermediate COCOMO model.
- Explain the complete COCOMO model.

ORGANIC, SEMIDETACHED AND EMBEDDED SOFTWARE PROJECTS

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs.

System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing. Boehm’s [1981] definitio

Organic: A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached: A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded: A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist

COCOMO

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages:

Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

BASIC COCOMO MODEL

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

effort = a1 x (kloc)a 2 pm

tdev = b1 x (effort)b 2 months

Where

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
- a1, a2, b1, b2 are constants for each category of software products,
- Tdev is the estimated time to develop the software, expressed in months,
- Effort is the total effort required to develop the software product, expressed in person months (PMs).

Estimation of development effort For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : **Effort = 2.4(KLOC) 1.05 PM**
Semi-detached : **Effort = 3.0(KLOC) 1.12 PM**
Embedded : **Effort = 3.6(KLOC) 1.20 PM**

Estimation of development time For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : **Tdev = 2.5(Effort) 0.38 Months**
Semi-detached : **Tdev = 2.5(Effort) 0.35 Months**
Embedded : **Tdev = 2.5(Effort) 0.32 Months**

Example:

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time. From the basic COCOMO estimation formula for organic software:

Effort = 2.4 x (32)1.05 = 91 PM Nominal development time = 2.5 x (91)0.38 = 14 months

Cost required to develop the product = 14 x 15,000 = Rs. 210,000/-

INTERMEDIATE COCOMO MODEL

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items:

PRODUCT: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

COMPUTER: Characteristics of the computer that are considered include the execution speed required, storage space required etc.

PERSONNEL: The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.

DEVELOPMENT ENVIRONMENT: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

COMPLETE COCOMO MODEL

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These subsystems may have widely different characteristics. For example, some subsystems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems may be different, but also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate. The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

- Database part
- Graphical User Interface (GUI) part
- Communication part Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system

SCHEDULING

- Gantt chart Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity. Gantt charts are used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished. A Gantt chart representation for the MIS problem of fig. 11.8 is shown in the fig. 11.9.

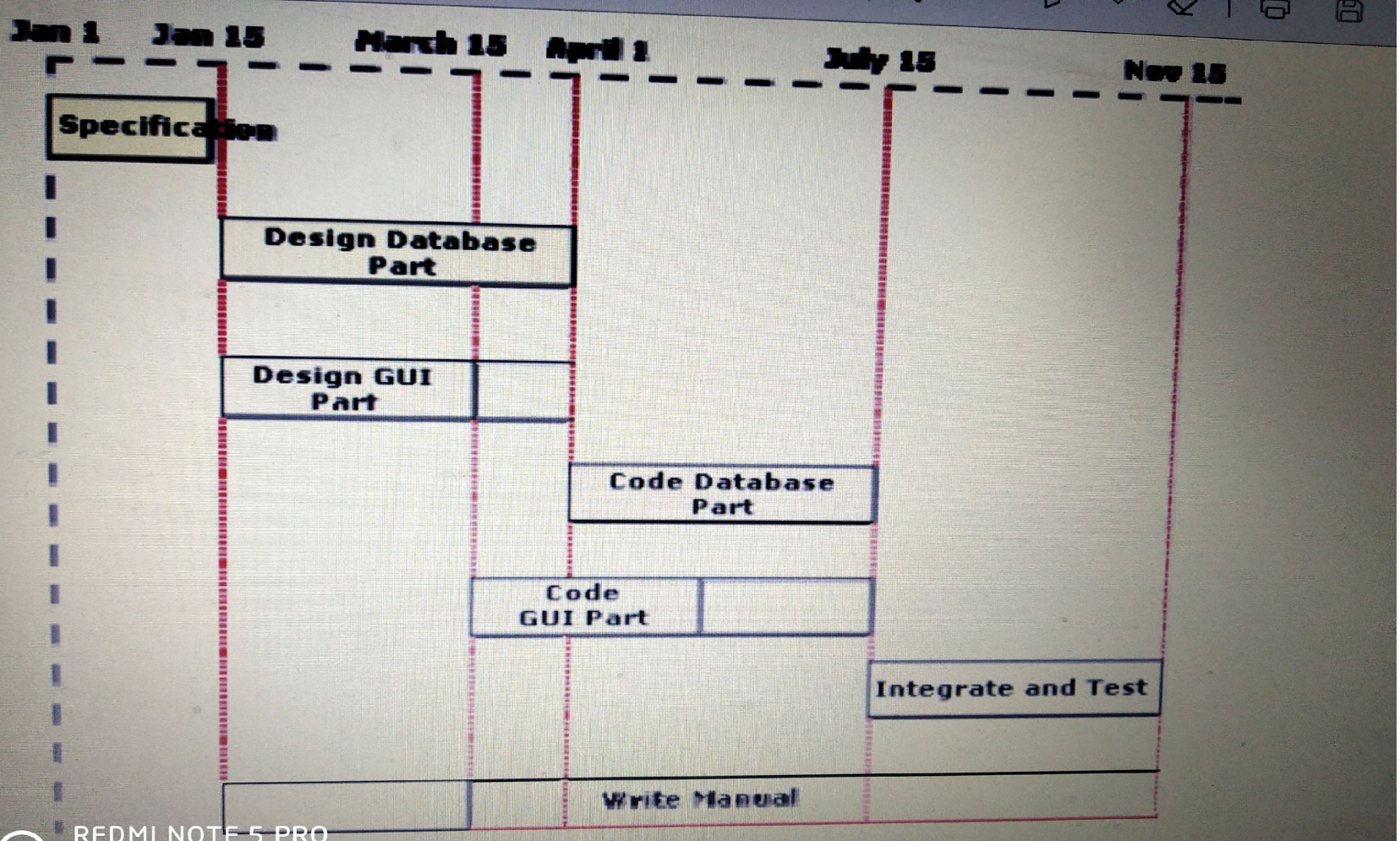

Fig. 11.9: Gantt chart representation of the MIS problem

PERT CHART

PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made. The boxes of PERT charts are usually annotated with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem of fig. 11.8 is shown in fig. 11.10. PERT charts are a more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since, the actual durations might vary from the estimated durations, the utility of the activity diagrams are limited. 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.

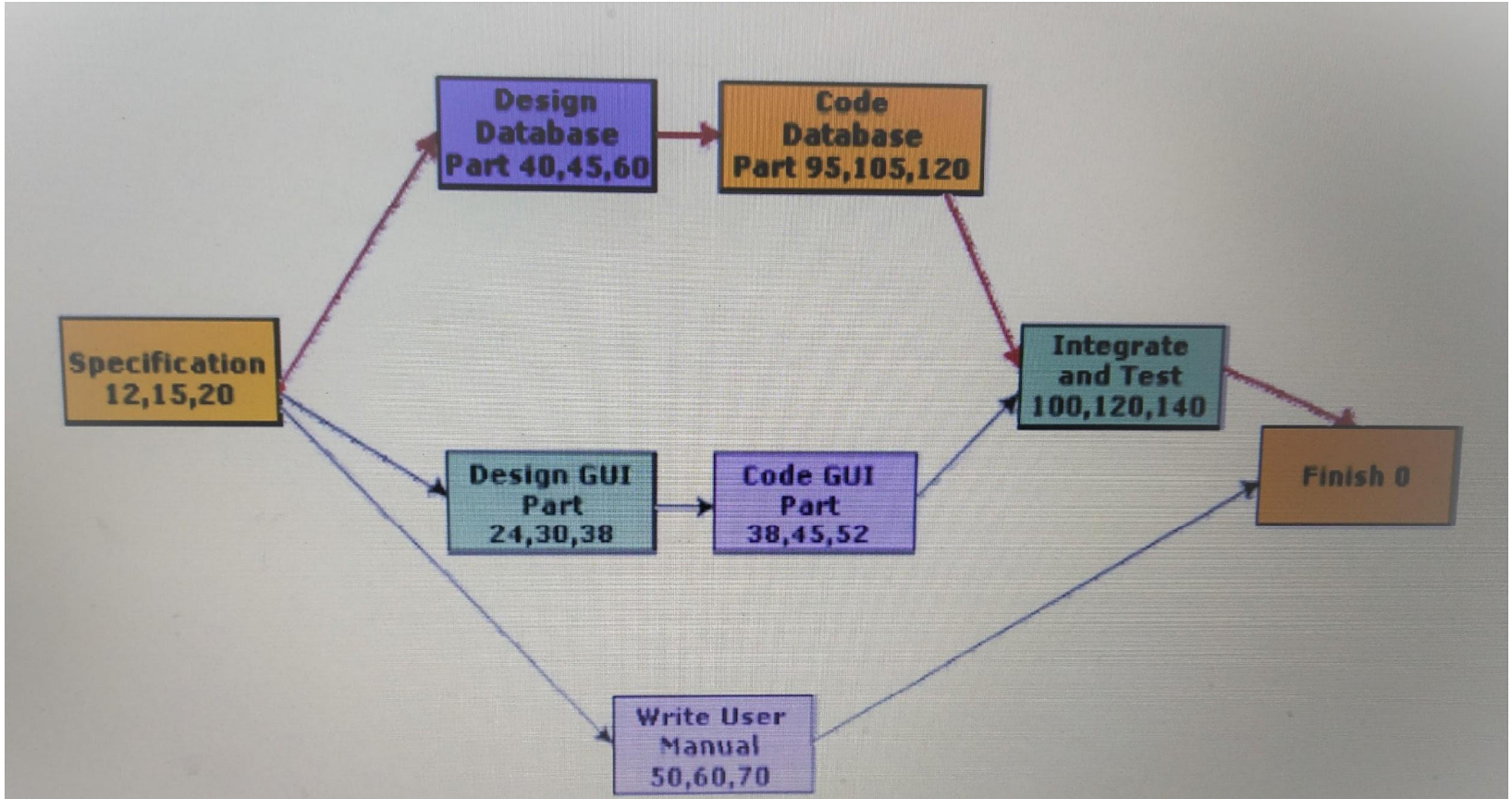

Fig. 11.10: PERT chart representation of the MIS problem

THANKS

SOFTWARE ENGINEERING
LECTURE-12
05/02/21

BASIC CONCEPTS IN REQUIREMENTS ANALYSIS AND SPECIFICATION

SPECIFIC INSTRUCTIONAL OBJECTIVES

- At the end of this lesson the student will be able to:
- Explain the role of a system analyst.
- Identify the important parts of SRS document.
- Identify the functional requirements from any given problem description.
- Document the functional requirements from any given problem description.
- Identify the important properties of a good SRS document.
- Identify the important problems that an organization would face if it does not develop an SRS document.
- Identify non-functional requirements from any given problem description.
- Identify the problems that an unstructured specification would create during software development.
- Represent complex conditions in the form of a decision tree.
- Represent complex conditions in the form of decision table.

Role of a system analyst

The analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be? After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness.

PARTS OF A SRS DOCUMENT

- **The important parts of SRS document are:**

Functional requirements of the system
Non-functional requirements of the system, and
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 highlevel functions {fi}. The functional view of the system is shown in fig. 3.1. Each function fi of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data (oi). The user can get some meaningful piece of work done using a high-level function.

4.2.2 Functional Requirements

In order to document the functional requirements of the system, it is necessary to learn how to first identify the high-level functional requirements of the system. Each high-level functional requirement corresponds to an instance of use of the system by the user in some way. Through the execution of a high-level requirement, the user can get some useful work done. Each high-level requirement typically involves accepting some data from the user, transforming it to the required response, and outputting the response to the user. For example, in a Library Automation Software, a high-level functional requirement might be `search-book`. This function involves accepting a book name or a set of key words from the user, running a matching algorithm on the book list, and finally outputting the matched entries. The generated system response can be in several forms, e.g. display on the terminal, a printout, some data transferred to the other systems, etc. However, in degenerate cases, a high-level requirement may not involve any input data or production of results.

Each high-level functional requirement may involve a series of interactions between the system and one or more users. An example of the interactions that may occur to complete a single high-level requirement is shown in Figure 4.2. Typically, there is some initial data input by the user. To this, the system may display some response (called system action). Based on this, the user may input further data, and so on. Even for the same high-level function, there can be different interaction sequences or scenarios (see Figure 4.2) due to users selecting different options or entering different data items. The different scenarios are essentially different paths (taken during an execution of a function) in a schematic interaction representation of a high-level functional requirement as shown in Figure 4.2. Typically, each user input and the corresponding system action may be considered as a sub-requirement of a high-level requirement. Thus, each high-level requirement might consist of several sub-requirements.

In requirements specification, it is important to define the precise data input to the system and the precise data output by the system. The data in a high-level requirement should be described using high-level terms and it may be very difficult to identify the exact components of this data accurately. A reason for this can be that in a high-level function, the data might be input to the system in stages. For example, consider the `withdraw-cash` function of an Automated Teller Machine (ATM) of Figure 4.2. Since during the course of execution of the `withdraw-cash` function, the user would have to input the type of account, the amount to be withdrawn, it is very difficult to form a single high-level name that would accurately describe both the input data. However, the input data for the subfunctions can be more accurately described.

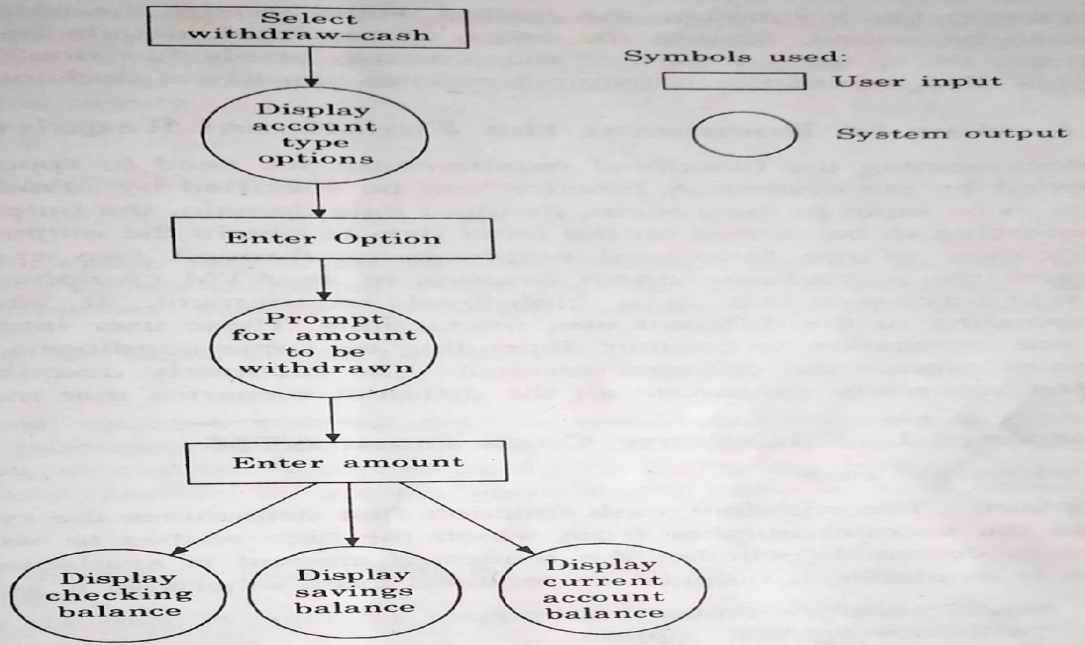

FIGURE 4.2 Interactions between the user and the system in the withdraw–cash high-level functional requirement.

4.2.3 How to Identify the Functional Requirements?

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. Remember that there can be many types of users of a system and their requirements (or expectations) from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

The decision regarding which functionality of the system can be considered to be a high-level functional requirement and the one that can be considered as part of another function (that is, a subfunction) leaves scope for some subjectivity. For example, consider the `issue-book` function in a Library Automation System. Suppose, when a user invokes

the `issue-book` function, the system would require the user to enter the details of each book to be issued. Should the entry of the book details be considered as a high-level function, or as only a part of the `issue-book` function? Many times, the choice is obvious. But, sometimes it requires making non-trivial decisions.

4.2.4 How to Document the Functional Requirements?

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. We now illustrate the specification of the functional requirements through two examples. Let us first try to document the `withdraw-cash` function of an ATM (Automated Teller Machine) system. The `withdraw-cash` is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These user interaction sequences may vary from one invocation to another depending on some conditions. These different interaction sequences capture the different *scenarios*. To accurately describe a functional requirement, we must accurately enumerate all the different scenarios that may occur.

Example 4.1 Withdraw Cash from ATM

R1: withdraw cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R1.1: select withdraw amount option
Input: "withdraw amount" option
Output: user prompted to enter the account type

R1.2: select account type
Input: user option
Output: prompt to enter amount

R1.3: get required amount
Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.
Output: The requested cash and printed transaction statement.
processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message is displayed.

Example 4.2 Search Book Availability in Library

R1: search book

Description: Once the user selects the search option, he would be asked to enter the

keywords. The system would search the book in the book list based on the key words entered. After making the search, the system should output the details of all books whose title or author name match any of the key words entered. The book details to be displayed include: title, author name, publisher name, year of publication, ISBN number, catalog number, and the location in the library.

R1.1: select search option
Input: "search" option
Output: user prompted to enter the key words

R1.2: search and display
Input: Key words
Output: Details of all books whose title or author name matches any of the key words entered by the user. The book details displayed would include: title of the book, author name, ISBN number, catalog number, year of publication, number of copies available, and the location in the library.
Processing: Search the book list based on the key words

R2: renew book

Description: When the 'renew' option is selected, the user is asked to enter his membership number and password. After password validation, the list of the books borrowed by him is displayed. The user can renew any of his borrowed books by indicating them. A requested book cannot be renewed if it is reserved by another user. In this case, an error message is displayed.

R2.1: select renew option
State: The user has logged in and the main menu has been displayed.
Input: 'renew' option selection
Output: Prompt message to the user to enter his membership number and password

R2.2: login
State: The renew option has been selected.
Input: Membership number and password
Output: List of the books borrowed by the user is displayed, and the user is prompted to select the books to be renewed, if the password is valid. If the password is invalid, the user is asked to reenter the password.
Processing: Password validation, search the books issued to the user from the borrower's list and display.
Next function: R2.3 if password is valid and R2.2 if password is invalid.

R2.3: renew selected books
Input: User choice for books to be renewed out of the books borrowed by him.
Output: Confirmation of the books successfully renewed and apology message for the books that could not be renewed.
Processing: Check if anyone has reserved any of the requested books. Renew the books selected by the user in the borrower's list, if no one has reserved those books.

IDENTIFYING NON-FUNCTIONAL REQUIREMENTS

Nonfunctional requirements are the characteristics of the system which can not be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc. Nonfunctional requirements may include:

reliability issues, # performance issues,
human - computer interface issues,
interface with other external systems,
security and maintainability of the system, etc.

Nonfunctional requirements:-

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

Nonfunctional requirements may include:

reliability issues,
accuracy of results,
human - computer interface issues,
constraints on the system implementation, 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

PROPERTIES OF A GOOD SRS DOCUMENT

• The important properties of a good SRS document are the following:

Concise. The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

Structured. It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

Black-box view. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

Conceptual integrity. It should show conceptual integrity so that the reader can easily understand it.

Response to undesired events. It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

Verifiable. All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation. Problems without a SRS document • The important problems that an organization would face if it does not develop an SRS document are as follows:

Without developing the SRS document, the system would not be implemented according to customer needs.

Software developers would not know whether what they are developing is what exactly required by the customer.

Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system. It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Level 1 DFD

To develop the level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the level 1 DFD. We can then examine the input data to these functions and the data output by these functions and represent them appropriately in the diagram. If a system has more than 7 high-level functional requirements, then some of the related requirements have to be combined and represented in the form of a bubble in the level 1 DFD. Such a bubble can be split in the lower DFD levels. If a system has less than three high-level functional requirements, then some of them need to be split into their sub-functions so that we have roughly about 5 to 7 bubbles on the diagram.

DECOMPOSITION

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything between 3 to 7 bubbles. Too few bubbles at any level make that level 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.

COMMONLY MADE ERRORS WHILE CONSTRUCTING A DFD MODEL

Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is powerful thing, it is an expensive pedagogical technique in the business world. It is therefore helpful to understand the different types of mistakes that users usually make while constructing the DFD model of systems.

- **Many beginners commit the mistake of drawing more than one bubble in the context diagram. A context diagram should depict the system as a single bubble.**
- **Many beginners have external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear at other levels of the DFD.**
- **It is a common oversight to have either too less or too many bubbles in a DFD. Only 3 to 7 bubbles per diagram should be allowed, i.e. each bubble should be decomposed to between 3 and 7 bubbles.**
- **Many beginners leave different levels of DFD unbalanced.**

A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD. It is important to realize that a DFD is the data flow representation of a system, and it does not represent control information. For an example mistake of this kind: o Consider the following example. A book can be searched in the library catalog by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalog, then an error message is generated. While generating the DFD model for this simple problem, many beginners commit the mistake of drawing an arrow (as shown in fig.) to indicate the error function is invoked after the search book. But, this is a control information and should not be shown on the DFD

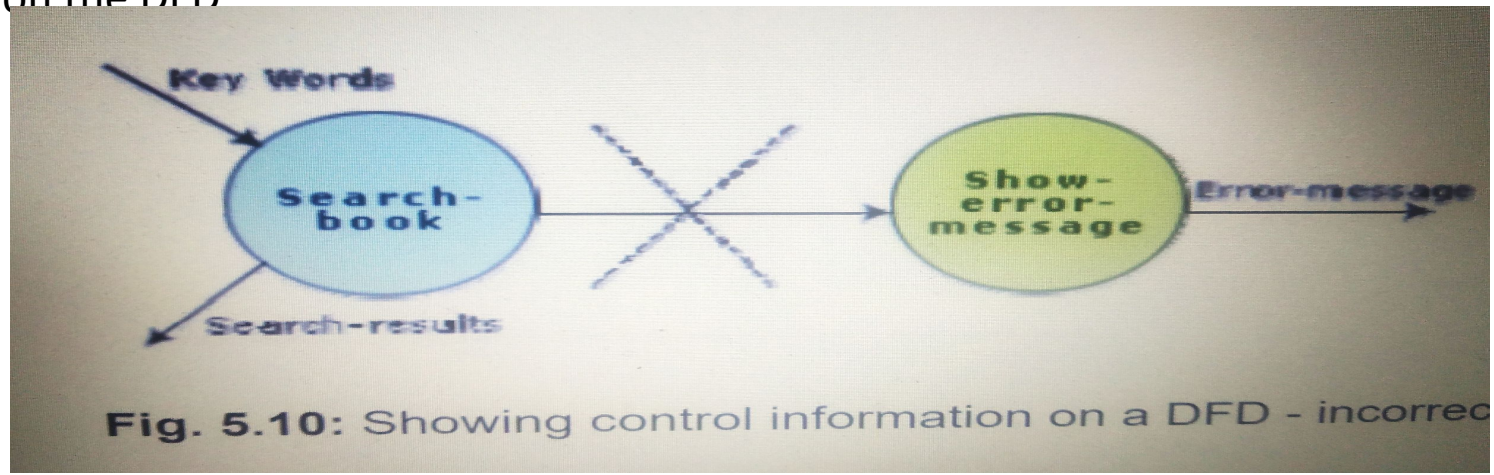

Fig. 5.10: Showing control information on a DFD - incorrec

o Another error is trying to represent when or in what order different functions (processes) are invoked and not representing the conditions under which different functions are invoked.

o If a bubble A invokes either the bubble B or the bubble C depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.

- A data store should be connected only to bubbles through data arrows. A data store cannot be connected to another data store or to an external entity.
- All the functionalities of the system must be captured by the DFD model. No function of the system specified in its SRS document should be overlooked.
- Only those functions of the system specified in the SRS document should be represented, i.e. the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD. • Improper or unsatisfactory data dictionary.
- The data and function names must be intuitive. Some students and even practicing engineers use symbolic data names such a, b, c, etc. Such names hinder understanding the DFD model.

SHORTCOMINGS OF A DFD MODEL

- DFDs leave ample scope to be imprecise. In the DFD model, the function performed by a bubble is judged from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information are missing or are incorrect. Further, the find-bookposition bubble may not convey anything regarding what happens when the required book is missing.
- Control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modeling real-time systems.
- The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one

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