Course Name: M. Tech
Semester: 2nd
Paper Name: Object Oriented Software Engineering (MTCS-203B)
Topic: Software Design.

Basic Concept of Software Design

Introduction

Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language. A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps. Design activities can be broadly classified into two important parts:

- Preliminary (or high-level) design.
- Detailed design.

Characteristics of a good software design

The definition of “a good software design” can vary depending on the application being designed. For example, the memory size used by a program may be an important issue to characterize a good solution for embedded software development – since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption considerations. For embedded applications, one may sacrifice design comprehensibility to achieve code compactness. For embedded applications, factors like design comprehensibility may take a back seat while judging the goodness of design. Therefore, the criteria used to judge how good a given design solution is can vary widely depending upon the application. Not only is the goodness of design dependent on the targeted application, but also the notion of goodness of a design itself varies widely across software engineers and academicians. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general application must possess. The characteristics are listed below:

- Correctness: A good design should correctly implement all the functionalities identified in the SRS document.
- Understandability: A good design is easily understandable.

- Efficiency: It should be efficient.
- Maintainability: It should be easily amenable to change.

Architectural Design-is used to carry out the top-down decomposition of a set of high-level functions depicted in the problem description and to represent them graphically. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analyzed and hierarchically decomposed into more detailed functions. Structured analysis technique is based on the following essential underlying principles:

- Top-down decomposition approach.
- Divide and conquer principle. Each function is decomposed independently.
- Graphical representation of the analysis results using Data Flow Diagrams (DFDs).

Low Level Design

This phase starts with the requirement document delivered by the requirement phase and maps the requirements into architecture. The architecture defines the components, their interfaces and behaviors. The deliverable design document is the architecture. The design document describes a plan to implement the requirements. This phase represents the ``how" phase. Details on computer programming languages and environments, machines, packages, application architecture, distributed architecture layering, memory size, platform, algorithms, data structures, global type definitions, interfaces, and many other engineering details are established. The design may include the usage of existing components.

Modularization

Modular programming is a software design technique that increases the extent to which software is composed of separate parts, called modules. Conceptually, modules represent a separation of_concerns, and improve maintainability by enforcing logical boundaries between components. Modules are typically incorporated into the program through interfaces. A module interface expresses the elements that are provided and required by the module. The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface.

Design Structure Charts

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks which are used to design structure charts are the following:

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from one module to another module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
- **Library modules:** Represented by a rectangle with double edges.
- **Selection:** Represented by a diamond symbol.
- **Repetition:** Represented by a loop around the control flow arrow.

Structure Chart vs. Flow Chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- 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.

Pseudo Codes

Pseudo code is a technique of writing application code in comment form in great detail. The code is easier for developer to understand as it sets the ground work for actual code logic as well as the location of actual programming code .the use of pseudo code can help with documentation as it can be explored from the application source during the documentation phase for use in actual solution documentation.

Flow Charts

Flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem. Flowcharts are generally drawn in the early stages of formulating computer solutions. Flowcharts facilitate communication between programmers and business people. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Often we see how flowcharts are helpful in explaining the program to others. Hence, it is correct to say that a flowchart is a must for the better documentation of a complex program.

Cohesion

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling. Cohesion is measure of functional strength of a module. A module having high cohesion and low couplings said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity.

Function-oriented design

The following are the salient features of a typical function-oriented design approach:

1. A system is viewed as something that performs a set of functions. Starting at this high level view of the system, each function is successively refined into more detailed functions. For example, consider a function create-new-library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This function may consist of the following sub-functions:
   - Assign-membership-number
   - Create-member-record
   - Print-bill

Each of these sub-functions may be split into more detailed sub functions and so on.

2. The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updation to several functions such as:

- create-new-member
- delete-member
- update-member-record

Object-oriented design

In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

Function-oriented vs. object-oriented design approach

The following are some of the important differences between function-oriented and object-oriented design.

• Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as employees, departments, etc. Grady Booch sums up this difference as “identify verbs if you are after procedural design and nouns if you are after object-oriented design”

• In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or other the real-world functions must be implemented. In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.

Top-Down and Bottom-Up Strategies

Top-down and bottom-up are strategies of information processing and knowledge ordering, mostly involving software. In practice, they can be seen as a style of thinking and teaching. In many cases top-down is used as a synonym of analysis or decomposition, and bottom-up of synthesis.

A top-down approach (is also known as step-wise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is first formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate.

However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

A bottom-up approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness.

Software Measurement and Metrics: Various Size Oriented Measures

Software process and product metrics are quantitative measures that enable software people to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved. The following topic categories are presented:

Halestead's Software Science

It is an analytical technique to measure

1- size
2- development effort
3- development time

Halestead use primitive program parameters

For some program let

- η^1 be the number of unique operators used in the program,
- η^2 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.

- The terms operators and operands have intuitive meanings,
    - A precise definition of these terms is needed to avoid ambiguities.

    - Unfortunately there is no general agreement among researchers on definition of operators and operands.

Operators

- Some general guidelines can be provided:
    - All assignment, arithmetic, and logical operators are operators.
    - A pair of parentheses,
        - As well as a block begin --- block end pair, are considered as single operators.
    - A label is considered to be an operator,
        - If it is used as the target of a GOTO statement.
- An if ... then ... else ... endif and a while ... do construct are single operators.
- A sequence (statement termination) operator ';' is a single operator.
- function call
    - Function name is an operator,
    - I/O parameters are considered as operands.
- Operands are those variables and constants
    - Which are being used with operators in expressions?
- Note that variable names appearing in declarations
    - Are not considered as operands.

Length and Vocabulary

- Length of a program quantifies
    - total usage of all operators and operands in the program:
    - Thus, length N=N1+N2.
- Program vocabulary:
    - Number of unique operators and operands used in the program.
    - Program vocabulary $\eta = \eta^1 + \eta^2$.

Program Volume:

- The length of a program:
    - total number of operators and operands used in the code
    - depends on the choice of the operators and operands,
        - i.e. for the same program, the length depends on the style of programming.

We can have highly different measures of length

    - for essentially the same problem.

- To avoid this kind of problem,
    - the notion of program volume V is introduced:
    - V= N log2 η

Potential Minimum Volume:

- Intuitively, program volume V denotes
    - Minimum number of bits needed to encode the program.
- To represent η different identifiers,
    - we need at least log2 η bits (η is the program vocabulary)
- The potential minimum volume V*:
    - Volume of the most succinct program in which the program can be coded.
- Minimum volume is obtained :
    - when the program can be expressed using a single source code instruction:
        - Say a function call like foo().
- The program level L is given by L=V*/V.
- L is a measure of the level of abstraction:
    - Languages can be ranked into levels that appear intuitively correct.

Effort and Time:

- Effort E=V/L, where
    - E is the number of mental discriminations required to write the program
    - Also the effort required to read and understand the program.
- Thus, programming effort E = V2/V*
    - Since L= V*/V varies as the square of the volume.
- Experience shows
    - E is well correlated to the effort needed for maintenance.

Length Estimation:

- Halstead assumed that it is quite unlikely that a program has several identical parts ---
    - or substrings of length greater than (η being the program vocabulary).
- In fact, once a piece of code occurs identically in several places,
    - it is usually made into a procedure or a function.

- Thus, we can safely assume:
- Any program of length N consists of N/η) unique strings of length. It is a standard combinatorial result that for any given alphabet of size K,
    - There are exactly Kr different strings of length r.
- Thus, N/ $\eta < \eta$
- Or, $N < \eta^{\eta+1}$
- N must include not only the ordered set of N elements,
    - but it must also include all possible subsets of that ordered set,
    - i.e. the power set of N strings.
    - Therefore, $2^N = (\eta^1)^{\eta 1} (\eta^2)^{\eta 2}$.
- $N=\log2((\eta^1)^{\eta 1} (\eta^2)^{\eta 2})$ (approximately)
- $N=\log2(\eta^1)^{\eta 1}+\log2(\eta^2)^{\eta 2}$

  $= \eta^1 \log2 \eta^1 + \eta^2 \log2 \eta^2$

Example:

- 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 1=12, 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

Function Point (FP) Based Measures

Software is identified and each one is categorized into one of five types: outputs, inquiries, inputs, internal files, and external interfaces. Once the function is identified and categorized into a type, it is then assessed for complexity and assigned a number of function points. Each of these functional user requirements maps to an end-user business function, such as a data entry for an Input or a user query for an Inquiry. This distinction is important because it tends to make the functions measured in function points map easily into user-oriented requirements, but it also tends to hide internal functions (e.g. algorithms), which also require resources to implement. Over the years there have been different approaches proposed to deal with this perceived weakness; however there is no ISO recognized FSM Method that includes algorithmic complexity in the sizing result. The variations of the Albrecht based IFPUG method designed to make up for this (and other weaknesses) include:

- Early and easy function points. Adjusts for problem and data complexity with two questions that yield a somewhat subjective complexity measurement; simplifies measurement by eliminating the need to count data elements.
- Engineering function points. Elements (variable names) and operators (e.g., arithmetic, equality/inequality, Boolean) are counted. This variation highlights computational function. The intent is similar to that of the operator/operand-based Halstead_Complexity Measures.
- Bang measure - Defines a function metric based on twelve primitive (simple) counts that affect or show Bang, defined as "the measure of true function to be delivered as perceived by the user." Bang measure may be helpful in evaluating a software unit's value in terms of how much useful function it provides, although there is little evidence in the literature of such application. The use of Bang measure could apply when re-engineering (either complete or piecewise) is being considered, as discussed in Maintenance of Operational Systems—An Overview.

Cyclomatic Complexity Measures: Control Flow Graphs:

Control flow graphs describe the logic structure of software modules. A module corresponds to a single function or subroutine in typical languages, has a single entry and exit point, and is able to be used as a design component via a call/return mechanism. This document uses C as the language for examples, and in C a module is a function. Each flow graph consists of nodes and edges. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes.

Definition of Cyclomatic complexity, v (G):

Cyclomatic complexity is defined for each module to be e - n + 2, where e and n are the number of edges and nodes in the control flow graph, respectively.
The word "Cyclomatic" comes from the number of fundamental (or basic) cycles in connected, undirected graphs .More importantly, it also gives the number of independent paths through strongly connected directed graphs. A strongly connected graph is one in which each node can be reached from any other node by following directed edges in the graph. The Cyclomatic number in graph theory is defined as e - n + 1. Program control flow graphs are not strongly connected, but they become strongly connected when a "virtual edge" is added connecting the exit node to the entry node. The Cyclomatic complexity definition for program control flow graphs is derived from the Cyclomatic number formula by simply adding one to represent the contribution of the virtual edge. This definition makes the Cyclomatic complexity equal the number of independent paths through the standard control flow graph model, and avoids explicit mention of the virtual edge.