SOFTWARE ENGINEERING
UNIT-2

1. Requirements Analysis And Specification:
2. Requirements Gathering and Analysis,
3. Software Requirement Specification (SRS)
4. Formal System Specification.

5. Software Design: Overview of the Design Process.
6. How to Characterize a Design?
7. Cohesion and Coupling.
8. Layered Arrangement of Modules.
9. Approaches to Software Design

1. REQUIREMENTS ANALYSIS AND SPECIFICATION

- The requirements analysis and specification to be a very important phase of software development life cycle and undertake it with utmost care.
- Before starting to develop a software, the exact requirements of the customer must be understood and documented.
- In the past, many projects have suffered because the developers started to implement something without determining whether they were building what the customers exactly wanted.
- Starting development work without properly understanding and documenting the requirements increases the number of iterative changes in the life cycle phases.
- Experienced developers take considerable time to understand the exact requirements of the customer.
- Without a clear understanding of the problem and proper documentation of the same, it is impossible to develop a satisfactory solution.
- For any type of software development project, availability of a good quality requirements document has been acknowledged to be a key factor in the successful completion of the project.
- A good requirements document not only helps to form a clear understanding of various features required from the software, but also serves as the basis for various activities carried out during later life cycle phases.
- When software is developed in a contract mode, the crucial role played by documentation of the requirements.

An overview of requirements analysis and specification phase

- The requirements analysis and specification phase starts after the feasibility study stage is complete and the project has been found to be financially viable and technically feasible.
- The requirements analysis and specification phase ends when the requirements specification document has been developed and reviewed.
- The requirements specification document is usually called as the *software requirements specification* (SRS) document.
- The goal of the requirements analysis and specification phase is
    - to clearly understand the customer requirements and
    - to systematically organize the requirements into a document called the Software Requirements Specification (SRS) document

.

Who carries out requirements analysis and specification?

- Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site.
- The engineers who gather and analyse customer requirements and then write the requirements specification document are known as *system analysts* in the software industry parlance.
- System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done.
- After understanding the precise user requirements, the analysts analyse the requirements to remoe inconsistencies, anomalies and incompleteness.

- They then proceed to write the *software requirements specification* (SRS) document. The SRS document is the final outcome of the requirements analysis and specification phase.

How is the SRS document validated?

- Once the SRS document is ready, it is first reviewed internally by the project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete.
- The SRS document is then given to the customer for review.
- After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organization.

What are the main activities carried out during requirements analysis and specification phase?

- Requirements analysis and specification phase mainly involves carrying out the following two important activities:
  - Requirements gathering and analysis
  - Requirements specification

2. REQUIREMENTS GATHERING AND ANALYSIS

- The complete set of requirements are almost never available in the form of a single document from the customer.
- The complete requirements are rarely obtainable from any single customer representative. Therefore, the requirements have to be gathered by the analyst from several sources in bits and pieces.
- These gathered requirements need to be analyzed to remove several types of problems.
- Requirements gathering and analysis activity divided into two separate tasks:
  - Requirements gathering
  - Requirements analysis

Requirements Gathering

- Requirements gathering is also popularly known as *requirements elicitation*.
- The primary objective of the requirements gathering task is to collect the requirements from the *stakeholders*.
- A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.
- Requirements gathering may sound like a simple task.
- Gathering requirements very difficult when no working model of the software is available .
- Availability of a working model is usually of great help in requirements gathering.
- Typically even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed.
- During visit to the customer site, the analysts normally interview the end-users and customer representatives, carry out requirements gathering activities such as
  - questionnaire surveys,
  - task analysis,
  - scenario analysis, and
  - Form analysis.

- Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively, make the functionalities more general and more complete.
- In the following, the important ways in which an experienced analyst gathers requirements:

1. Studying existing documentation:

- The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site.
- Customers usually provide statement of purpose (SoP) document to the developers.
- Typically these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, features of any similar software developed elsewhere, etc.

2. Interview:

- Typically, there are many different categories of users of a software.
- Each category of users typically requires a different set of features from the software.
- Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each.
- For example, the different categories of users of a library automation software could be
    - the library members,
    - the librarians, and
    - the accountants.
- **The library members** would like to use the software to query availability of books and issue and return books.
- **The librarians** might like to use the software to determine books that are overdue, create member accounts, delete member accounts, etc.
- **The accounts** personnel might use the software to invoke functionalities concerning financial aspects such as the total fee collected from the members, book procurement expenditures, staff salary expenditures, etc.
- To systematise this method of requirements gathering, the Delphi technique can be followed.
- In this technique, the analyst consolidates the requirements as understood by him in a document and then circulates it for the comments of the various categories of users.
- Based on their feedback, he refines his document.
- This procedure is repeated till the different users agree on the set of requirements.

3. Task analysis:

- The users usually have a black-box view of software and consider the software as something that provides a set of services.
- A service supported by software is also called a *task*.
- The software performs various tasks of the users.
- The analyst tries to identify and understand the different tasks to be performed by the software.
- For each task, the analyst tries to formulate the different steps necessary to realize the required functionality in consultation with the users.
- For example, for the issue book service, the steps may be—
    - authenticate user,
    - check the number of books issued to the customer and
    - determine if the maximum number of books that this member can borrow has been reached,

    - check whether the book has been reserved,
    - post the book issue details in the member's record, and
    - Finally print out a book issue slip that can be presented by the member at the security counter to take the book out of the library premises.
- Task analysis helps the analyst to understand the various user tasks and to represent each task as a hierarchy of subtasks.
- **Scenario analysis:** A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations.
- **Form analysis:** Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.

Requirements Analysis

- After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to find out any problems in the gathered requirements.
- It is natural to expect that the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness.
- Therefore, it is necessary to identify all the problems in the requirements and resolve them through further discussions with the customer.
- The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements.
- The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:
    - What is the problem?
    - Why is it important to solve the problem?
    - What exactly are the data input to the system and what exactly are the data output by the system?
    - What are the possible procedures that need to be followed to solve the problem?
    - 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 should be the data interchange formats with the external systems?
- After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various problems that he detects in the gathered requirements.
- During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:
    - Anomaly
    - Inconsistency
    - Incompleteness

Let us examine these different types of requirements problems in detail.

- **Anomaly:** It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible.
    - Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development.

- **Incompleteness:** An incomplete set of requirements is one in which some requirements have been overlooked.
    - The lack of these features would be felt by the customer much later, possibly while using the software.
    - Often, incompleteness is caused by the inability of the customer to visualize the system that is to be developed and to anticipate all the features that would be required.
    - An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

3. SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

- After the analyst has gathered all the required information regarding the software to be developed, and has removed all incompleteness, inconsistencies, and anomalies from the specification, he starts to systematically organize the requirements in the form of an SRS document.
- The SRS document usually contains all the user requirements in a structured though an informal form.
- Among all the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write.
- In the following, the different categories of users of an SRS document and their needs from it.

Users of SRS Document

- Some of the important categories of users of the SRS document and their needs for use are as follows:
- **Users, customers, and marketing personnel:** These stakeholders need to refer to the SRS document to ensure that the system as described in the document will meet their needs. For generic products, the marketing personnel need to understand the requirements that they can explain to the customers.
- **Software developers:** The software developers refer to the SRS document to make sure that they are developing exactly what is required by the customer.
- **Test engineers:** The test engineers use the SRS document to understand the functionalities, and based on this write the test cases to validate its working. They need that the required functionality should be clearly described, and the input and output data should have been identified precisely.
- **User documentation writers:** The user documentation writers need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users' manuals.
- **Project managers:** The project managers refer to the SRS document to ensure that they can estimate the cost of the project easily by referring to the SRS document and that it contains all the information required to plan the project.
- **Maintenance engineers:** The SRS document helps the maintenance engineers to under- stand the functionalities supported by the system.
    - A clear knowledge of the functionalities can help them to understand the design and code.
    - Also, a proper understanding of the functionalities supported enables them to determine the specific modifications to the system's functionalities would be needed for a specific purpose.
    - Many software engineers in a project consider the SRS document to be a reference document.

Uses of a well-formulated SRS document

- **Forms an agreement between the customers and the developers:** A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.
- **Reduces future reworks:** The process of preparation of the SRS document forces the stakeholders to rigorously think about all of the requirements before design and development get underway. This reduces later redesign, recoding, and retesting. Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.
- **Provides a basis for estimating costs and schedules:** Project managers usually estimate the size of the software from an analysis of the SRS document. Based on this estimate they make other estimations such as the effort required to develop the software and the total cost of development. The SRS document also serves as a basis for price negotiations with the customer. The project manager also uses the SRS document for work scheduling.
- **Provides a baseline for validation and verification:** The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the *test plan*.
- **Facilitates future extensions:** The SRS document usually serves as a basis for planning future enhancements. Before we discuss about how to write an SRS document, we first discuss the characteristics of a good SRS document and the pitfalls that one must consciously avoid while writing an SRS document.

Characteristics of a Good SRS Document

- The skill of writing a good SRS document usually comes from the experience gained from writing SRS documents for many projects.
- However, the analyst should be aware of the desirable qualities that every good SRS document should possess.
- IEEE Recommended Practice for Software Requirements Specifications describes the content and qualities of a good software requirements specification (SRS).

Some of the identified desirable qualities of an SRS document are the following:

- **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete.
- **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements.
    - It should only specify what the system should do and avoid that how to do these.
    - This means that the SRS document should specify the externally visible behavior of the system and not discuss the implementation issues.
    - This view with which a requirements specification is written, has been shown in Figure 4.1.
    - **Figure 4.1:** The black-box view of a system as performing a set of functions.

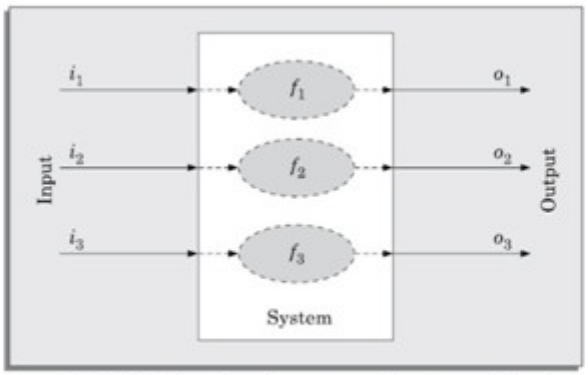


Figure 4.1: The black-box view of a system as performing a set of functions.

- **Traceable:** It should be possible to trace a specific requirement to the design elements that implement it and *vice versa*.
  - Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and *vice versa*.
  - Traceability is also important to verify the results of a phase.
- **Modifiable:** Customers frequently change the requirements during the software development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development.
- **Identification of response to undesired events:** The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.
- **Verifiable:** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation.

Bad SRS Documents

- SRS documents written by novices frequently suffer from a variety of problems.

Some of the important categories of problems that many SRS documents suffer from are as follows:

- **Over-specification:** It occurs when the analyst tries to address the "how to" aspects in the SRS document.
- **Forward references:** One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.
- **Wishful thinking:** This type of problems concern description of aspects which would be difficult to implement.
- **Noise:** The term noise refers to presence of material not directly relevant to the software development process.

Categories of Customer Requirements

- A good SRS document, should properly categorize and organize the requirements into different sections.
- As per the IEEE 830 guidelines, the important categories of user requirements are the following. An SRS document should clearly document the following aspects of a software:
  - Functional requirements
  - Non-functional requirements—
    - Design and implementation constraints
    - External interfaces required
    - Other non-functional requirements

- Constraints
- Goals of implementation.

In the following, the different categories of requirements.

- **Functional requirements**
  - The functional requirements capture the functionalities required by the users from the system.
  - It is useful to consider a software as offering a set of functions {f*i*} to the user.
  - These functions can be considered similar to a mathematical function f : I → O, meaning that a function transforms an element (i*i*) in the input domain (I) to a value (o*i*) in the output (O).
  - This functional view of a system is shown schematically in Figure 4.1.

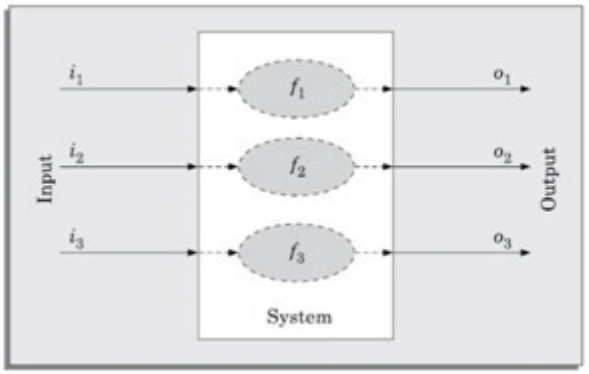


Figure 4.1: The black-box view of a system as performing a set of functions.

  - Each function f*i* of the system can be considered as reading certain data i*i*, and then transforming a set of input data (i*i*) to the corresponding set of output data (o*i*).
  - The functional requirements of the system, should clearly describe each functionality that the system would support along with the corresponding input and output data set.
  - Considering that the functional requirements are a crucial part of the SRS document,

- **Non-functional requirements**
  - The non-functional requirements capture those requirements of the customer that cannot be expressed as functions (i.e., accepting input data and producing output data).
  - Non-functional requirements usually address aspects concerning
    - external interfaces,
    - user interfaces,
    - maintainability,
    - portability,
    - usability,
    - maximum number of concurrent users,
    - timing, and
    - Throughput (transactions per second, etc.).
  - The non-functional requirements can be critical in the sense that any failure by the developed software to achieve some minimum level in these requirements.
  - The IEEE 830 standard recommends that out of the various non-functional requirements, the external interfaces, and the design and implementation constraints should be documented in two different sections.

- **Constraints:** Design and implementation constraints are an important category of non-functional requirements describe any items or issues that will limit the options available to the developers.
    - Some of the example constraints can be—
        - corporate or regulatory policies that needs to be honored;
        - hardware limitations;
        - interfaces with other applications;
        - specific technologies,
        - tools, and databases to be used;
        - specific communications protocols to be used;
        - security considerations;
        - design conventions or
        - Programming standards to be followed, etc.
        - Consider an example of a constraint that can be included in this section —Oracle DBMS needs to be used as this would facilitate easy interfacing with other applications that are already operational in the organization.
- **External interfaces required:** Examples of external interfaces are—
    - hardware, software and communication interfaces,
    - user interfaces,
    - Report formats, etc.
    - To specify the user interfaces, each interface between the software and the users must be described.
    - The description may include sample screen images,
    - any GUI standards or style guides that are to be followed,
    - screen layout constraints,
    - standard buttons and functions (e.g., help) that will appear on every screen,
    - keyboard shortcuts,
    - Error message display standards, and so on.
    - One example of a user interface requirement of a software can be that it should be usable by factory shop floor workers who may not even have a high school degree.
    - The details of the user interface design such as screen designs, menu structure, navigation diagram, etc. should be documented in a separate user interface specification document.
- **Other non-functional requirements:** This section contains a description of non-functional requirements that are neither design constraints and nor are external interface requirements.
    - An important example is a performance requirement such as the number of transactions completed per unit time.
    - The other non-functional requirements may include
        - reliability issues,
        - accuracy of results, and
        - Security issues.
- **Goals of implementation:** The ‘goals of implementation’ part of the SRS document offers some general suggestions regarding the software to be developed.
    - These are not binding on the developers, and they may take these suggestions into account if possible.
    - For example, the developers may use these suggestions while choosing among different design solutions.

- A goal, in contrast to the functional and non-functional requirements, is not checked by the customer for conformance at the time of acceptance testing.
- The goals of implementation section might document issues such as
  - Easier revisions to the system functionalities.
  - easier support for new devices to be supported,
  - 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.
- It is useful to remember that anything that would be tested by the user and the acceptance of the system would depend on the outcome of this task, is usually considered as a requirement to be fulfilled by the system and not a goal and *vice versa*.

Example -3.1 (Withdraw cash from ATM):

- An initial informal description of a required functionality is usually given by the customer as a *statement of purpose* (SoP).
- An SoP serves as a starting point for the analyst and he proceeds with the requirements gathering activity after a basic understanding of the SoP.
- However, the functionalities of withdraw cash from ATM is intuitively obvious to anyone who has used a bank ATM.
- So, we are not including an informal description of withdraw cash functionality here and in the following, we documents this functional requirement.

R.1: Withdraw cash (R.1 Means Requirement. one)

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.

***: Select withdraw amount optio*n**

Input: "Withdraw amount" option selected *Output:* User prompted to enter the account type

: Select account type

I n p u t : User selects option from any one of the followings— savings/checking/deposit.

Output: Prompt to enter amount

: 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 displayed.

Example 3.2 (Search book availability in library):

- An initial informal description of a required functionality is usually given by the customer as a *statement of purpose* (SoP) based on which an later requirements gathering, the analyst understand the functionality.
- However, the functionalities of *search book availability* is intuitively obvious to any one who has used a library. So, we are not including an informal description of *search book availability* functionality here and in the following, we documents this functional requirement.

R.1: 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.

: Select search option

Input: "Search" option

Output: User prompted to enter the key words

: 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:

R.2: 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 are 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 would be displayed.

: 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.

: Login

State: The renew option has been selected.

Input: Membership number and password.

Output: Li st of the books borrowed by the user is displayed, and 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 re-enter the password.

Processing: Password validation, search the books issued to the user from the borrower's list and display.

Next function: R.2.3 if password is valid and R.2.2 if password is invalid.

: 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. In order to properly identify the high-level requirements, a lot of common sense and the ability to visualize various scenarios that might arise in the operation of a function are required. Please note that when any of the aspects of a requirement, such as the state, processing description, next function to be executed, etc. are obvious, we have omitted it. We have to make a trade-off between cluttering the document with trivial details versus missing out some important descriptions.

Organization of the SRS Document

- **Introduction :** Introduction about the project
- **Purpose:** This section should describe where the software would be deployed and and how the software would be used.
- **Project scope:** This section should briefly describe the overall context within which the software is being developed.
- **Environmental characteristics:** This section should briefly outline the environment (hardware and other software) with which the software will interact.

Overall description of organization of SRS document

- **Product perspective:** This section needs to briefly state as to whether the software is intended to be a replacement for a certain existing systems, or it is a new software.
- **Product features:** This section should summarize the major ways in which the software would be used.
- **User classes:** Various user classes that are expected to use this software are identified and described here. The different classes of users are identified by the types of functionalities
- **Operating environment:** This section should discuss in some detail the hardware platform on which the software would run, the operating system, and other application software with which the developed software would interact.
- **Design and implementation constraints:** In this section, the different constraints on the design and implementation are discussed. These might include—corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; specific programming language to be used; specific communication protocols to be used; security considerations; design conventions or programming standards.
- **User documentation:** This section should list out the types of user documentation, such as user manuals, on-line help, and trouble-shooting manuals that will be delivered to the customer along with the software.

Functional requirements for organization of SRS document

- This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes, etc., depending what may be appropriate.
  1. User class 1
  (a) Functional requirement 1.1
  (b) Functional requirement 1.2
  2. User class 2
  (a) Functional requirement 2.1
  (b) Functional requirement 2.2

External interface requirements

- **User interfaces:** This section should describe a high-level description of various interfaces and various principles to be followed. The user interface description may include sample screen images, any GUI standards or style guides etc.
- **Hardware interfaces:** This section should describe the interface between the software and the hardware components of the system.
- **Software interfaces:** This section should describe the connections between this software and other specific software components, including databases, operating systems, tools, libraries, and integrated commercial components, etc.

- **Communications interfaces:** This section should describe the requirements associated with any type of communications required by the software, such as e-mail, web access, network server communications protocols, etc.

Other non-functional requirements for organization of SRS document

- **Performance requirements:** Some performance requirements may be specific to individual functional requirements or features.
- **Safety requirements:** Those requirements that are concerned with possible loss or damage that could result from the use of the software are specified here.
- **Security requirements:** This section should specify any requirements regarding security or privacy requirements on data used or created by the software.

Functional requirements

1. Operation mode 1
(a) Functional requirement 1.1
(b) Functional requirement 1.2
2. Operation mode 2
(a) Functional requirement 2.1
(b) Functional requirement 2.2

Structure of SRS Document:

1. Introduction
    1.1 Purpose
    1.2 Scope
    1.3 Definition, Acronyms, and abbreviations
    1.4 References
    1.5 Documentation Overview
2. General Descriptions
    2.1 Product perspective
    2.2 Product functions
    2.3 User functions
    2.4 General Constraints
    2.5 Assumptions and Dependencies
3. Specific Requirements
    3.1 Functional Requirements
        3.1.1 Functional requirement 1
            3.1.1.1 Introduction
            3.1.1.2 Input
            3.1.1.3 Processing
            3.1.1.4 Output
        3.1.2 Functional requirement 2
        .
        .
        3.1.N Functional requirement N
    3.2 External Interface Requirements
    3.3 Performance Requirements
    3.4 Design constraints
    3.5 Security Requirements
    3.6 Maintainability Requirements
    3.7 Reliability Requirements
    3.8 Data base Requirements
    3.9 Documentation requirements
    3.10 Operational Requirements

Representing Complex Logic

- A good SRS document should properly characterize the conditions.
- Sometimes the conditions can be complex and numerous and several alternative interaction and processing sequences may exist depending on the outcome of the corresponding condition checking.
- A simple text description in such cases can be difficult to analyze.
- In such situations, a decision tree or a decision table can be used to represent the logic and the processing involved.
- Also, when the decision making in a functional requirement has been represented as a decision table, it becomes easy to automatically or at least manually design test cases for it.
- There are two main techniques available to analyze and represent complex processing logic—decision trees and decision tables.
- Once the decision making logic is captured in the form of trees or tables, the test cases to validate these logic can be automatically obtained.

Decision tree

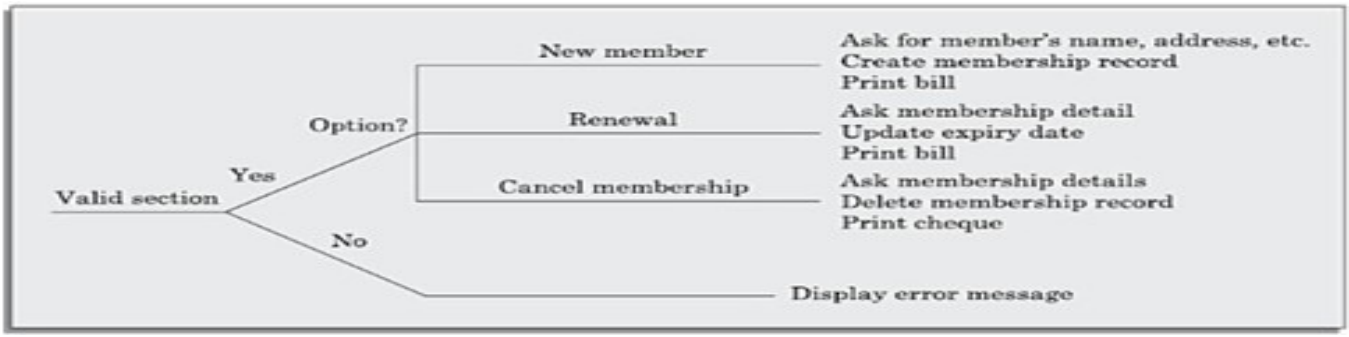
- A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken.
- The **edges** of a decision tree **represent conditions** and the **leaf nodes represent the actions** to be performed depending on the outcome of testing the conditions.

Example: 3.3 A library membership management software (LMS) should support the following three options—

new member,
renewal, and
cancel membership.

- When the *new member* option is selected, the software should ask the member's name, address, and phone number.

- If proper information is entered, the software should create a membership record for the new member and print a bill for the annual membership charge and the security deposit payable.
- If the *renewal* option is chosen, the LMS should ask the member's name and his membership number and check whether he is a valid member.
- If the member details entered are valid, then the membership expiry date in the membership record should be updated and the annual membership charge payable by the member should be printed.
- If the membership details entered are invalid, an error message should be displayed.
- If the *cancel membership* option is selected and the name of a valid member is entered, then the membership is cancelled, a choke for the balance amount due to the member is printed and his membership record is deleted.
- The decision tree representation for this problem is shown in the following Figure.



- Observe from above Figure that the internal nodes represent conditions, the edges of the tree correspond to the outcome of the corresponding conditions.
- The leaf nodes represent the actions to be performed by the system. In the decision tree of Figure, first the user selection is checked. Based on whether the selection is valid, either further condition checking is undertaken or an error message is displayed. Observe that the order of condition checking is explicitly represented.

Decision table

- A decision table shows the decision making logic and the corresponding actions taken in a tabular or a matrix form.
- The upper rows of the table specify the variables or conditions to be evaluated and the lower rows specify the actions to be taken when an evaluation test is satisfied.
- A column in the table is called a *rule*. A rule implies that if a certain condition combination is true, then the corresponding action is executed.

- The decision table for the LMS problem of Example- 3.3 is as shown in Table.

Conditions				
Valid selection	NO	YES	YES	YES
New member	--	YES	NO	NO
Renewal	--	NO	YES	NO
Cancellation	--	NO	NO	YES
Actions				
Display error message				
Ask member's name etc.				
Build customer record	×	×		×
Generate bill	×			×
Ask membership details		×		
Update expiry date	×	×	×	
Print cheque		×	×	×
Delete record	×	×	×	

Decision table *versus* decision tree

Even though both decision tables and decision trees can be used to represent complex program logic, they can be distinguishable on the following three considerations:

Readability: Decision trees are easier to read and understand when the number of conditions are small. On the other hand, a decision table causes the analyst to look at every possible combination of conditions which he might otherwise omit.

Explicit representation of the order of decision making: In contrast to the decision trees, the order of decision making is abstracted out in decision tables. A situation where decision tree is more useful is when multilevel decision making is required. Decision trees can more intuitively represent multilevel decision making hierarchically, whereas decision tables can only represent a single decision to select the appropriate action for execution.

Representing complex decision logic: Decision trees become very complex to understand when the number of conditions and actions increase. It may even be to draw the tree on a single page. When very large number of decisions is involved, the decision table representation may be preferred.

4. FORMAL SYSTEM SPECIFICATION

What is a Formal Technique?

- A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc.
- The mathematical basis of a formal method is provided by its specification language.
- In general, formal techniques can be used at every stage of the system development activity to verify that the output of one stage conforms to the output of the previous stage.
- **Syntactic domains:** The syntactic domain of a formal specification language consists of an alphabet of symbols and a set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.
- **Semantic domains:** Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronization trees, partial orders, state machines, etc.
- **Satisfaction relation:** Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as *semantic abstraction function*. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behaviour and the others describe the system's structure. Consequently, t wo broad classes of semantic abstraction functions are defined— those that *preserve* a system's behaviour and those that *preserve* a system's structure.
- **Operational Semantics:** Informally, the *operational semantics* of a formal method is the way computations are represented. There are different types of operational semantics.
    - **Linear semantics:** In this approach, a *run* of a system is described by a sequence (possibly infinite) of events or states.
    - **Branching semantics:** In this approach, the behaviour of a system is represented by a directed graph. The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state.
    - **Maximally parallel semantics:** In this approach, all the concurrent actions enabled at any state are assumed to be taken together.
    - **Partial order semantics:** Under this view, the semantics ascribed to a system is a *structure of states* satisfying a partial order relation among the states (events).
- **Algebraic specification:** In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages. Essentially, algebraic specifications define a system as a *heterogeneous algebra*. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations defined in this set; e.g. { I, +, -, *, / }.

5. OVERVIEW OF THE DESIGN PROCESS

- During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document.
- Main objectives of the design phase is transform the SRS document into the design document.
- This view of a design process has been shown schematically in Figure 5.1.

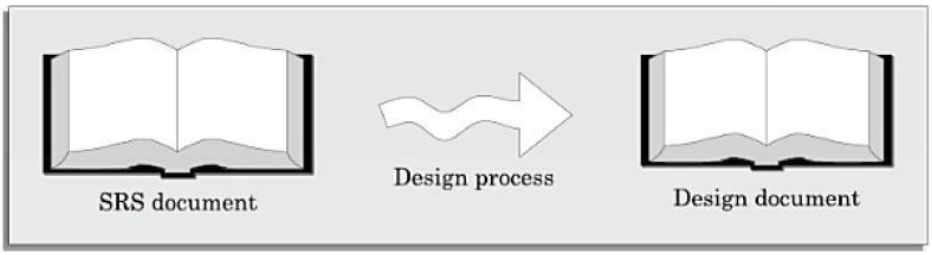


Figure 5.1: The design process.

As shown in Figure 5.1, the design process starts using the SRS document and completes with the production of the design document.

- The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.
- The design process essentially transforms the SRS document into a design document.

In the following sections, few important issues associated with the design process.

Outcome of the Design Process

The following items are designed and documented during the design phase.

- **Different modules required:**
  - The different modules in the solution should be clearly identified.
  - Each module is a collection of functions and the data shared by the functions of the module.
  - Each module should accomplish some well-defined task out of the overall responsibility of the software.
  - Each module should be named according to the task it performs.
- **Control relationships among modules:**
  - A control relationship between two modules essentially arises due to function calls across the two modules.
  - The control relationships existing among various modules should be identified in the design document.
- **Interfaces among different modules:**
  - The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.
- **Data structures of the individual modules:**
  - Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module.
  - Suitable data structures for storing and managing the data of a module need to be properly designed and documented.
- **Algorithms required to implement the individual modules:**
  - Each function in a module usually performs some processing activity.
  - The algorithms required to accomplish the processing activities of various modules.

- Starting with the SRS document (as shown in Figure 5.1), the design documents are produced through iterations over a series of steps.
- The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

Classification of Design Activities

- A good software design is seldom realized by using a single step procedure, rather it requires iterating over a series of steps called the design activities.
- Depending on the order in which various design activities are performed, we can broadly classify them into two important stages.
  - Preliminary (or high-level) design, and
  - Detailed design.
- The meaning and scope of these two stages can vary considerably from one design methodology to another.
- However, for the traditional function-oriented design approach, it is possible to define the objectives of the high-level design as follows:
- Through high-level design, a problem is decomposed into a set of modules.
- The control relationships among the modules are identified, and also the interfaces among various modules are identified.
- The outcome of high-level design is called the program structure or the software architecture.
- High-level design is a crucial step in the overall design of a software.
- When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy.
- A notation that is widely being used for procedural development is a tree-like diagram called the structure chart.
- Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.
- Though other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram are available to document a software design.

Classification of Design Methodologies

- The design activities vary considerably based on the specific design methodology being used.
- A large number of software design methodologies are available.
- These methodologies classified into procedural and object-oriented approaches.
- These two approaches are two fundamentally different design paradigms.
- **Do design techniques result in unique solutions?**
  - Even while using the same design methodology, different designers usually arrive at very different design solutions.
  - The reason is that a design technique often requires the designer to make many subjective decisions and work out compromises to contradictory objectives.
  - As a result, it is possible that even the same designer can work out many different solutions to the same problem.
  - Therefore, obtaining a good design would involve trying out several alternatives (or candidate solutions) and picking out the best one.
  - However, a fundamental question that arises at this point is—how to distinguish superior design solution from an inferior one?
  - Unless we know what a good software design is and how to distinguish a superior design solution from an inferior one, we cannot possibly design one. We investigate this issue in the next section.

- **Analysis versus design :**
  - Analysis and design activities differ in goal and scope.
  - The goal of any **analysis** technique is to elaborate the customer requirements through careful thinking and at the same time consciously avoiding making any decisions regarding the exact way the system is to be implemented.
  - The analysis results are generic and does not consider implementation or the issues associated with specific platforms.
  - The analysis model is usually documented using some graphical formalism.
  - In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart.
  - On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using unified modelling language (UML).
  - The analysis model would normally be very difficult to implement using a programming language.
  - **The design** model is obtained from the analysis model through transformations over a series of steps. In contrast to the analysis model, the design model reflects several decisions taken regarding the exact way system is to be implemented.
  - The design model should be detailed enough to be easily implementable using a programming language.

6. HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

- Coming up with an accurate characterization of a good software design that would hold across diverse problem domains is certainly not easy.
- In fact, the definition of a "good" software design can vary depending on the exact application being designed.

These characteristics are listed below:

- **Correctness:** A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
- **Understandability:** A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
- **Efficiency:** A good design solution should adequately address resource, time, and cost optimization issues.
- **Maintainability:** A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

7. COHESION AND COUPLING

- We have so far discussed that effective problem decomposition is an important characteristic of a good design.
- Good module decomposition is indicated through **high cohesion** of the individual modules and **low coupling** of the modules with each other.

Let us now define what is meant by cohesion and coupling.

- **Cohesion** is a measure of the functional strength of a module.
  - Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution.
  - When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion.

- If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion.

- **Coupling** is a measure of the degree of interaction (or interdependence) between the two modules.
  - Two modules are said to be highly coupled, if either of the following two situations arise:
    - If the function calls between two modules involve passing large volumes of shared data, the modules are **tightly coupled**.
    - If the interactions occur through some shared data, then also we say that they are **highly coupled.**
    - If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have **low coupling.**

Classification of Cohesiveness

- Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective.
- The different modules of a design can possess different degrees of freedom.
- Different classes of cohesion that modules can possess are depicted in the following diagram.

Coincidental	Logical	Temporal	Procedural	Communicational	Sequential	Functional
Low ---------- ☾ High						

- The cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible. These different classes of cohesion are elaborated below.

- **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that **relate to each other very loosely**, if at all. In this case, we can say that the module contains a random collection of functions. It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design.
  - An example of a module with coincidental cohesion has been shown in the following Figure.

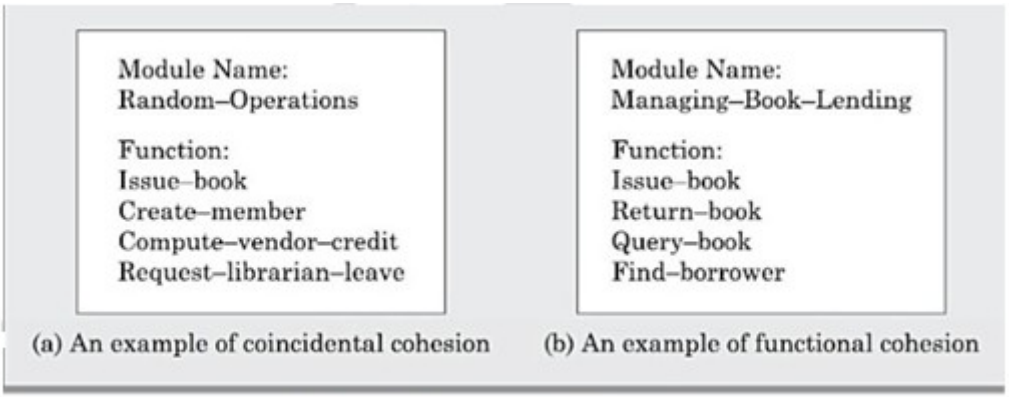


Figure 5.4: Examples of cohesion.

  - Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library

member records on one hand, and handling librarian leave request on the other.

- **Logical cohesion:** A module is said to be logically cohesive, if all elements of the module perform **similar operations**, such as error handling, data input, data output, etc.
    - As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.
- **Temporal cohesion:** When a module contains functions that are related by the fact that these functions are executed in the **same time span**, then the module is said to possess temporal cohesion.
    - As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.
- **Procedural cohesion:** A module is said to possess procedural cohesion, if the set of functions of the **module are executed one after the other**, though these functions may work towards entirely different purposes and operate on very different data.
    - Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), printbill(), place-order-on-vendor(), update-inventory(), and l ogout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.
- **Communicational cohesion:** A module is said to have communicational cohesion, if all functions of the module refer to or **update the same data structure**.
    - As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.
- **Sequential cohesion:** A module is said to possess sequential cohesion, if the different functions of **the module execute in a sequence**, and the output from one function is input to the next in the sequence.
    - As an example consider the following situation. In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions create-order(), check-item-availability(), placeorder- on-vendor() are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function create-order() creates an order that is processed by the function check-item-availability() (whether the items are available in the required quantities in the inventory) is input to place-order-on-vendor().
- **Functional cohesion:** A module is said to possess functional cohesion, if different functions of **the module co-operate to complete a single task**.
    - For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion.
    - In this case, all the functions of the module (e.g., computeOvertime(), computeWorkHours(), computeDeductions(), etc.) work together to generate the payslips of the employees.

Classification of Coupling

- The coupling between two modules indicates **the degree of interdependence** between them.

- Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled.
- Let us now classify the different types of coupling that can exist between two modules.
- Between any two interacting modules, any of the following five different types of coupling can exist. These different types of coupling, in increasing order of their severities have also been shown in Figure 5.5.

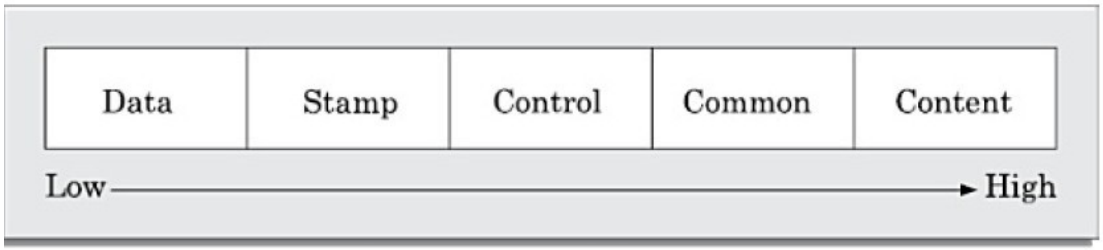


Figure 5.5: Classification of coupling.

- **Data coupling:** Two modules are data coupled, if they communicate using an **elementary data item that is passed as a parameter between the two**,
    - e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

- **Stamp coupling:** Two modules are stamp coupled, if they communicate using a **composite data** item such as a record in PASCAL or a structure in C.

- **Control coupling:** Control coupling exists between two modules, if data from one module is used to direct the order of **instruction execution** in another.
    - An example of control coupling is a flag set in one module and tested in another module.

- **Common coupling:** Two modules are common coupled, if they share some **global data** items.

- **Content coupling:** Content coupling exists between two modules, if they share **code**. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

- The different types of coupling are shown schematically in Figure 5.5. The degree of coupling increases from data coupling to content coupling. **High coupling among modules not only makes a design solution difficult to understand and maintain**, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

8. LAYERED ARRANGEMENT OF MODULES

- The control hierarchy represents the organization of program components in terms of their call relationships.
- Thus we can say that the control hierarchy of a design is determined by the order in which different modules call each other.
- Many different types of notations have been used to represent the control hierarchy.
- The most common notation is a tree-like diagram known as a structure chart
- However, other notations such as Warnier-Orr [1977, 1981] or Jackson diagrams [1975] may also be used.

- Since, Warnier-Orr and Jackson's notations are not widely used nowadays.
- In a layered design solution, the modules are arranged into several layers based on their call relationships.
- A module is allowed to call only the modules that are at a lower layer.

- That is, a module should not call a module that is either at a higher layer or even in the same layer.
- Figure 5.6(a) shows a layered design, whereas Figure 5.6(b) shows a design that is not layered.

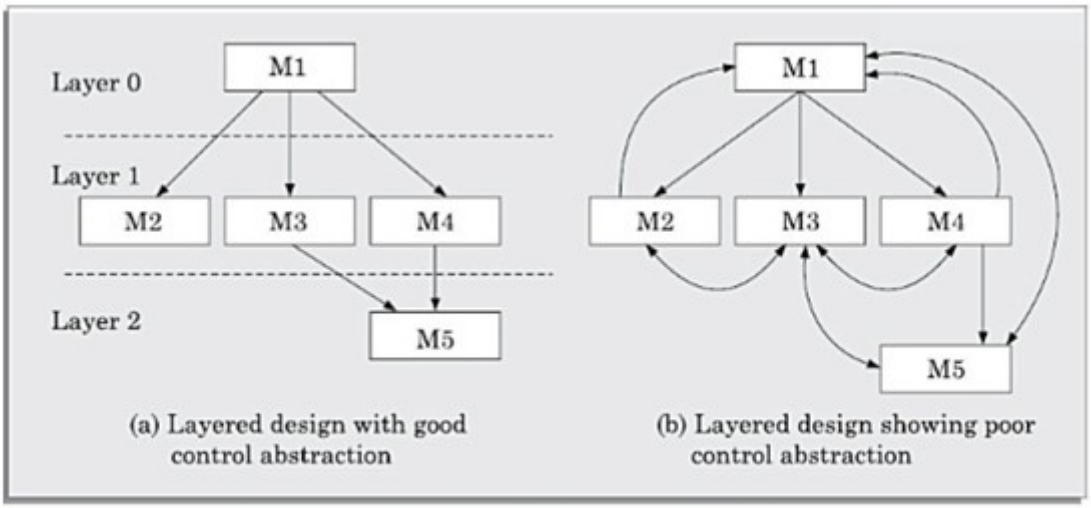


Figure 5.6: Examples of good and poor control abstraction.

- Observe that the design solution shown in Figure 5.6(b), is actually not layered since all the modules can be considered to be in the same layer.
- An important characteristic feature of a good design solution is layering of the modules.
- A layered design achieves control abstraction and is easier to understand and debug.
- In a layered design, the *top-most module in the hierarchy can be considered as a manager* that only invokes the services of the lower level module to discharge its responsibility.
- *The modules at the intermediate layers offer services to their higher layer by invoking the services* of the lower layer modules and also by doing some work themselves to a limited extent.
- The modules at the lowest layer are the **worker modules**. These do not invoke services of any module and entirely carry out their responsibilities by themselves.
- Understanding a layered design is easier since to understand one module, one would have to at best consider the modules at the lower layers (that is, the modules whose services it invokes).
- Besides, in a layered *design errors are isolated*, since an error in one module can affect only the higher layer modules. As a result, in case of any failure of a module, only the modules at the lower levels need to be investigated for the possible error.
- *Thus, debugging time reduces significantly in a layered design.*
- In the following, we discuss some important concepts and terminologies associated with a layered design:
- **Super-ordinate and sub-ordinate modules:** In a control hierarchy, a module that controls another module is said to be super-ordinate to it. Conversely, a module controlled by another module is said to be sub-ordinate to the controller.
- **Visibility:** A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.
- **Control abstraction:** In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules

at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

- **Depth and width:** Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.
- **Fan-out:** Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.
- **Fan-in:** Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

9. APPROACHES TO SOFTWARE DESIGN

- There are two fundamentally different approaches to software design that are in use today—
  - function-oriented design, and
  - object-oriented design.
- These two design approaches are different, they are complementary rather than competing techniques.
- The object oriented approach is a relatively newer technology and is still evolving.
- For development of large programs, the object- oriented approach is becoming increasingly popular due to certain advantages that it offers.
- Function-oriented designing is a mature technology and has a large following.

Function-oriented Design

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

- **Top-down decomposition:** A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.
  - In top-down decomposition, starting at a high-level view of the system, each high-level 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 high-level function may be refined into 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.
- **Centralized system state:** The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event.
  - For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system.
  - Such data in procedural programs usually have global scope and are shared by many modules.

- The system state is centralized and shared among different functions. For example, in the library management system, several functions such as the following share data such as member-records for reference and updating:
  - create-new-member
  - delete-member
  - update-member-record

A large number of function-oriented design approaches have been proposed in the past. A few of the well-established function-oriented design approaches are as following:

- Structured design by Constantine and Yourdon, [1979]
- Jackson's structured design by Jackson [1975]
- Warnier-Orr methodology [1977, 1981]
- Step-wise refinement by Wirth [1971]
- Hatley and Pirbhai's Methodology [1987]

Object-oriented Design

- In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities).
- Each object is associated with a set of functions that are called its methods.
- Each object contains its own data and is responsible for managing it.
- The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object.
- The system state is decentralized since there is no globally shared data in the system and data is stored in each object.
  - For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data.
- The methods defined for one object cannot directly refer to or change the data of other objects.
- The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition as explained below.
- Objects decompose a system into functionally independent modules.
- Objects can also be considered as instances of abstract data types (ADTs).
- The ADT concept did not originate from the object-oriented approach. In fact, ADT concept was extensively used in the ADA programming language.
- ADT is an important concept that forms an important pillar of object orientation.
- Let us now discuss the important concepts behind an ADT. There are, in fact, three important concepts associated with an ADT—
  - data abstraction,
  - data structure,
  - Data type.

  We discuss these in the following subsection:

- **Data abstraction:** The principle of data abstraction implies that how data is exactly stored is abstracted away.
  - This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organized, and manipulated inside the object.
  - The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object.

- Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and can access data of a stack object only through the supported operations such as push and pop.

- **Data structure:** A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organized collection of primitive data items such as integer, floating point numbers, characters, etc.
- **Data type:** A type is a programming language terminology that refers to anything that can be instantiated.
  - For example, int, float, char etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.
  - In object-orientation, classes are ADTs. But, what is the advantage of developing an application using ADTs?