UNIT – II
Requirements Analysis and Specification & Software Design

Requirements Analysis and Specification
Many projects fail:
because they start implementing the system:
without determining whether they are building what the customer really wants.
It is important to learn:
requirements analysis and specification techniques thoroughly.
Goals of requirements analysis and specification phase:
fully understand the user requirements
remove inconsistencies, anomalies, etc. from requirements
document requirements properly in an SRS document
Consists of two distinct activities:

Requirements Gathering and Analysis
Software Requirements Specification (SRS)

The person who undertakes requirements analysis and specification:
known as systems analyst:
collects data pertaining to the product
analyzes collected data to understand what exactly needs to be done.
writes the Software Requirements Specification (SRS) document.
Final output of this phase:
Software Requirements Specification (SRS) Document.
The SRS document is reviewed by the customer.
reviewed SRS document forms the basis of all future development activities.

Requirements Analysis

Requirements analysis consists of two main activities:
Requirements gathering
Analysis of the gathered requirements
Analyst gathers requirements through:
observation of existing systems,
studying existing procedures,
discussion with the customer and end-users,
analysis of what needs to be done, etc.

Requirements Gathering

If the project is to automate some existing procedures
e.g., automating existing manual accounting activities,
the task of the system analyst is a little easier
analyst can immediately obtain:
input and output formats
accurate details of the operational procedures
In the absence of a working system,
lot of imagination and creativity are required.
Interacting with the customer to gather relevant data:
requires a lot of experience.
Some desirable attributes of a good system analyst:
Good interaction skills,

imagination and creativity,
experience.

Analysis of the Gathered Requirements

After gathering all the requirements:
analyze it:
Clearly understand the user requirements,
Detect inconsistencies, ambiguities, and incompleteness.
Incompleteness and inconsistencies:
resolved through further discussions with the end-users and the customers.

Inconsistent requirement

Some part of the requirement:
contradicts with some other part.
<u>Example:</u>
One customer says turn off heater and open water shower when temperature > 100 C
Another customer says turn off heater and turn ON cooler when temperature > 100 C
Some requirements have been omitted:
due to oversight.
<u>Example:</u>
The analyst has not recorded:
when temperature falls below 90 C
heater should be turned ON
water shower turned OFF.

Analysis of the Gathered Requirements

Requirements analysis involves:
obtaining a clear, in-depth understanding of the product to be developed,
remove all ambiguities and inconsistencies from the initial customer perception of the problem.
It is quite difficult to obtain:

a clear, in-depth understanding of the problem is needed especially when there is no working model of the problem.

Experienced analysts take considerable time: to understand the exact requirements the customer has in his mind.
Experienced systems analysts know

often as a result of painful experiences, without a clear understanding of the problem, it is impossible to develop a satisfactory system.

Several things about the project should be clearly understood by the analyst:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What complexities might arise while solving the problem?

Some anomalies and inconsistencies can be very subtle:

- Escape even most experienced eyes.
- If a formal model of the system is constructed,
- Many of the subtle anomalies and inconsistencies get detected.

After collecting all data regarding the system to be developed,

- remove all inconsistencies and anomalies from the requirements,
- systematically organize requirements into a Software Requirements Specification (SRS) document.

Software Requirements Specification

Main aim of requirements specification: systematically organize the requirements arrived during requirements analysis document requirements properly.

The SRS document is useful in various contexts:

statement of user needs

contract document

reference document

definition for implementation

Software Requirements Specification: A Contract Document

Requirements document is a reference document.

SRS document is a contract between the development team and the customer.

Once the SRS document is approved by the customer,

any subsequent controversies are settled by referring the SRS document.

Once customer agrees to the SRS document:

development team starts to develop the product according to the requirements recorded in the SRS document.

The final product will be acceptable to the customer:

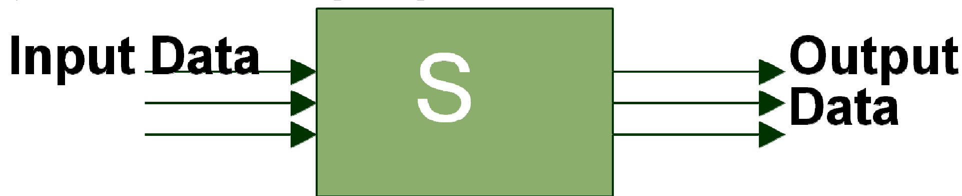
as long as it satisfies all the requirements recorded in the SRS document.

SRS Document

The SRS document is known as black-box specification:

the system is considered as a black box whose internal details are not known.

only its visible external (i.e. input/output) behavior is documented.



The requirements are written using end-user terminology.

If necessary later a formal requirement specification may be developed from it.

Properties of a good SRS document

It should be concise and at the same time should not be ambiguous.

It should specify what the system must do and not say how to do it.

Easy to change i.e. it should be well-structured.

It should be consistent.

It should be complete.

It should be traceable: You should be able to trace which part of the specification corresponds to which part of the design and code, etc and vice versa.

It should be verifiable: e.g. "system should be user friendly" is not verifiable

SRS document, normally contains three important parts:

functional requirements,

Non-functional requirements,

Constraints on the system.

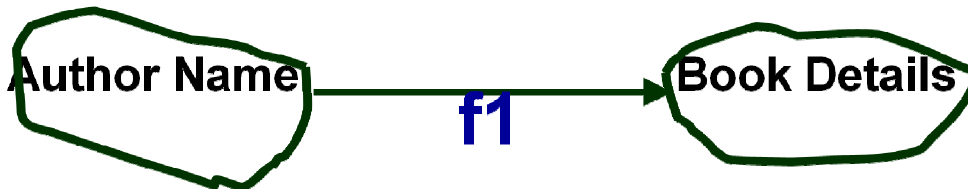
It is desirable to consider every system:

performing a set of functions {fi}.

Each function fi considered as:

transforming a set of input data to corresponding output data.

Example: Functional Requirement
Search Book
Input:
an author's name:
Output:
details of the author's books and the locations of these books in the library.



Functional requirements describe:
A set of high-level requirements
Each high-level requirement:
takes in some data from the user
outputs some data to the user
might consist of a set of identifiable functions
For each high-level requirement:
every function is described in terms of
input data set
output data set
processing required to obtain the output data set from the input data set

Nonfunctional Requirements
Characteristics of the system which can not be expressed as functions:
Non-functional requirements include:
reliability issues,
performance issues,
human-computer interface issues,
Interface with other external systems,
security, maintainability, etc.
Constraints
Constraints describe things that the system should or should not do.
For example,
standards compliance
how fast the system can produce results
so that it does not overload another system to which it supplies data, etc.
Hardware to be used,
Operating system or DBMS to be used
Capabilities of I/O devices
Standards compliance
Data representations by the interfaced system

Organization of the SRS Document

- Introduction
- Functional Requirements
- Non-functional Requirements
    - External interface requirements
    - Performance requirements
- Constraints

Examples of Bad SRS Documents

Ambiguity:

Literary expressions

Unquantifiable aspects, e.g. "good user interface"

Forward References:

References to aspects of problem

defined only later on in the text.

Wishful Thinking:

Descriptions of aspects

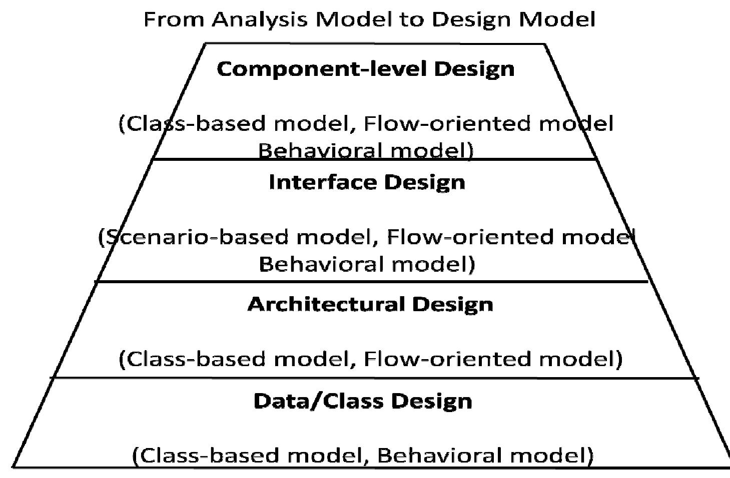
for which realistic solutions will be hard to find.

Software Design

Software design and its activities

Software design deals with transforming the customer requirements, as described in the SRS document, into 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:
    - ✓ Identification of different modules and the control relationships among them and the definition of the interfaces among these modules
    - ✓ The outcome of high-level design is called the program structure or software architecture
    - ✓ Many different types of notations have been used to represent a high-level design.
- Detailed design
    - ✓ During detailed design, the data structure and the algorithms of the different modules are designed.
    - ✓ The outcome of the detailed design stage is usually known as the module-specification document.
- Difference between analysis and design
- The aim of analysis is to understand the problem with a view to eliminate any deficiencies in the requirement specification such as incompleteness inconsistencies, etc.
- The model which we are trying to build may be or may not be ready. (Preliminary level)
- The aim of design is to produce a model that will provide a seamless transition to the coding phase, i.e. once the requirements are analyzed and found to be satisfactory, a design model is created which can be easily implemented.



From Analysis Model to Design Model

- Each element of the analysis model provides information that is necessary to create the four design models
  - The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
  - The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
  - The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
  - The component-level design transforms structural elements of the software architecture into a procedural description of software components
- Items developed during the software design phase
- For a design to be easily implemented in a conventional programming language, the following items must be designed during the design phase.
- Different modules required to implement the design solution.
- Control relationship among the identified modules. The relationship is also known as the call relationship or invocation relationship among modules.
- Interface among different modules. The interface among different modules identifies the exact data items exchanged among the modules.
- Data structures of the individual modules.
- Algorithms required to implement each individual module.

Characteristics of a good software design

The definition of "a good software design" can vary depending on the application being designed.

- 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.
- Features of a design document
- In order to facilitate understandability, the design should have the following features:
- It should use consistent and meaningful names for various design components.
- The design should be modular. The term modularity means that it should use a cleanly decomposed set of modules.
- It should neatly arrange the modules in a hierarchy, e.g. in a tree-like diagram.

Characterize of a software design

- Complete and sufficient
  - Contains the complete encapsulation of all attributes and methods that exist for the class
  - Contains only those methods that are sufficient to achieve the intent of the class
- Primitiveness
  - Each method of a class focuses on accomplishing one service for the class
- High cohesion
  - The class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities
- Low coupling
  - Collaboration of the class with other classes is kept to an acceptable minimum
  - Each class should have limited knowledge of other classes in other subsystems
- Component independence:

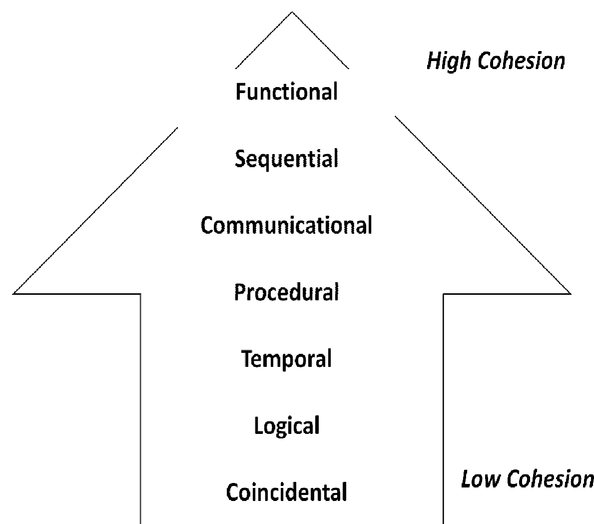
The primary characteristics of neat module decomposition are

- High cohesion
- Low coupling

Cohesion

- Definition
  - The degree to which all elements of a component are directed towards a single task.
  - The degree to which all elements directed towards a task are contained in a single component.
  - The degree to which all responsibilities of a single class are related.
- All elements of component are directed toward and essential for performing the same task.

Type of Cohesion



Coincidental Cohesion

- Def: Parts of the component are unrelated (unrelated functions, processes, or data)
- The module contains a random collection of functions
- The functions have been put in the module out of pure coincidence without any thought or design
- Elements needed to achieve some functionality are scattered throughout the system.
- Accidental
- Worst form

Example:

1. Print next line
2. Reverse string of characters in second argument
3. Add 7 to 5th argument
4. Convert 4th argument to float

The grouping does not have any relevance to the structure of the problem.

Logical Cohesion

- Def: Elements of component are related logically and not functionally.
- A module is said to be logically cohesive, if all elements of the module perform similar operations
- Example: An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module
- A component reads inputs from tape, disk, and network.
- All the code for these functions are in the same component.
- Operations are related, but the functions are significantly different.

Temporal Cohesion

- Def: When a module contains functions that are related by the fact that all the functions must be executed in the same time span
- Elements are grouped by when they are processed.
- Example:

A system initialization routine: this routine contains all of the code for initializing all of the parts of the system.

An exception handler:

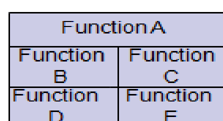
  - Closes all open files, Creates an error log, Notifies user,
  - Lots of different activities occur, all at same time

Procedural Cohesion

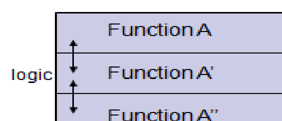
Def: if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.

- Actions are still weakly connected and unlikely to be reusable.
- Example:
  - Write output record
  - Read new input record
  - Pad input with spaces
  - Return new record

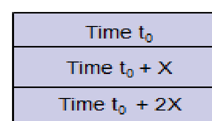
Examples of Cohesion



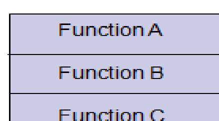
Coincidental
Parts unrelated



Logical
Similar functions



Temporal
Related by time



Procedural
Related by order of functions

Communicational Cohesion

- Def: Functions performed on the same data or to produce the same data.
- if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.
- Examples:
  - Update record in data base and send it to the printer
    - Update a record on a database
    - Print the record
  - Fetch unrelated data at the same time.
    - To minimize disk access

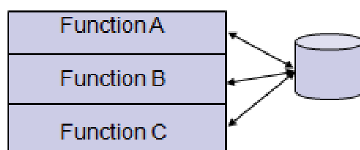
Sequential Cohesion

Def: if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.

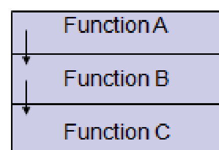
- *Data flows* between parts
- Occurs naturally in functional programming languages
- Good situation
- For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

Functional Cohesion

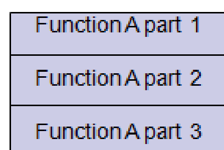
- Def: Every essential element to a single computation is contained in a component.
- Every element in the component is essential to the computation.
- Ideal situation
- For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion
- Able to describe it using a single sentence.
- What is a functionally cohesive component?
  - One that not only performs the task for which it was designed and performs by only that function



Communicational
Access same data



Sequential
Output of one is input to another



Functional
Sequential with complete, related functions

Object cohesion (strong)

Each operation provides functionality which allows object attributes to be modified or inspected

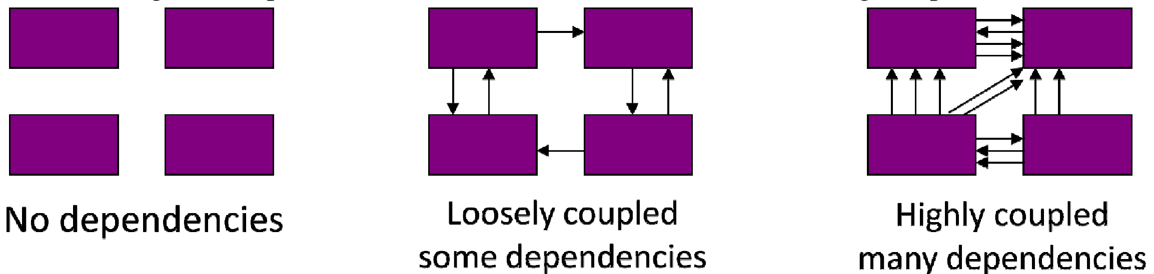
Exercise: Cohesion for Each Module?

- Compute average daily temperatures at various sites
- Initialize sums and open files
- Create new temperature record

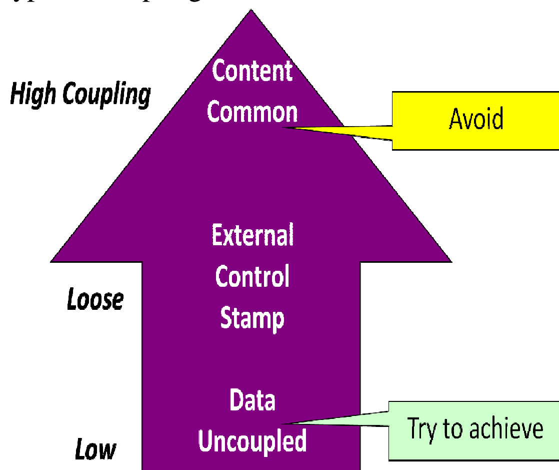
- Store temperature record
- Close files and print average temperatures
- Read in site, time, and temperature
- Store record for specific site
- Edit site, time, or temperature field

Coupling

The degree of dependence such as the amount of interactions among components



Type of Coupling



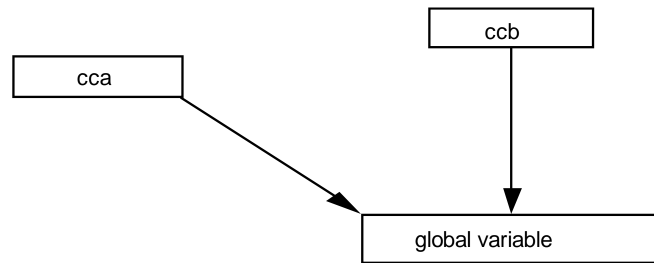
Content Coupling

- Def: one module directly references contents of the other
  - Component directly modifies another's data
  - Component modifies another's code, e.g., jumps (goto) into the middle of a routine
- Question
  - Language features allowing this?
- Example
  - module **a** modifies statements of module **b**
  - When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.
- Why is this bad?
  - any change to **b** requires changes to **a**

Common Coupling

- Def.
  - two modules have write access to the same global data
- Example
  - two modules have access to same database, and can both read and write same record
- Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks. Each source process writes directly to global data store. Each sink process reads directly from global data store.

```
while (global variable == 0)
    if (argument xyz > 25)
        module 3 ();
    else
        module 4 ();
```



- Usually a poor design choice because
  - Lack of clear responsibility for the data
  - Reduces readability
  - Difficult to determine all the components that affect a data element (reduces maintainability)
  - Difficult to reuse components
  - Reduces ability to control data accesses i.e. module exposed to more data than necessary

External Coupling

- Def: Two components share something externally imposed, e.g.,
  - External file
  - Device interface
  - Protocol
  - Data format

Control Coupling

- Def.
  - one module passes an element of control to the other
- Example
  - control-switch passed as an argument
  - Good example: sort that takes a comparison function as an argument.
- Why is this bad?
  - modules are not independent
    - module **b** must know the internal structure of module **a**
    - affects reusability

Stamp Coupling

- Def.
  - Component passes a data structure to another component that does not have access to the entire structure.
  - Example

To calculate withholding (employee record)

- Why is this bad?
  - affects understanding
    - not clear, without reading entire module, which fields of record are accessed or changed
  - unlikely to be reusable
    - other products have to use the same higher level data structures
  - passes more data than necessary
    - e.g., uncontrolled data access can lead to computer crime

Customer Billing System

- The print routine of the customer billing accepts customer data structure as an argument, parses it, and prints the name, address, and billing information.

Improvement --- OO Solution

- Use an interface to limit access from clients

Customer
get_name ( ) get_address( ) get_billing info( ) get_other_info ( ) …

Data Coupling

- Def.
  - every argument is either a simple argument or a data structure in which all elements are used by the called module
  - Component passes data (not data structures) to another component
  - Good, if it can be achieved.
- Example

display time of arrival (flight number)

- Why is this good?
  - maintenance is easier
  - good design has high cohesion & weak coupling

Modules to Objects