DEPARTMENT OF COMPUTER SCIENCE AND ENGG.
SOFTWARE ENGG.
SECTION-B

Requirement Engineering

The process to gather the software requirements from client, analyze and document them is known as requirement engineering. The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

Requirements analysis, also called **requirements engineering**, is the process of determining user expectations for a new or modified product. These features, called **requirements**, must be quantifiable, relevant and detailed. In software **engineering**, such **requirements** are often called functional specifications. **Requirements engineering** (RE) refers to the process of defining, documenting and maintaining requirements to the sub-fields of systems engineering and software engineering concerned with this process.

The activities involved in requirements engineering vary widely, depending on the type of system being developed and the specific practices of the organization(s) involved. These may include:

1. Requirements inception or requirements elicitation -
2. Requirements identification - identifying new requirements
3. Requirements analysis and negotiation - checking requirements and resolving stakeholder conflicts
4. Requirements specification (e.g., software requirements specification; SRS) - documenting the requirements in a requirements document
5. Systems modeling - deriving models of the system, often using a notation such as the Unified Modeling Language (UML) or the Lifecycle Modeling Language (LML)
6. Requirements validation - checking that the documented requirements and models are consistent and meet stakeholder needs
7. Requirements management - managing changes to the requirements as the system is developed and put into use

Requirement Engineering Process

It is a four step process, which includes –

- Feasibility Study
- Requirement Gathering
- Software Requirement Specification
- Software Requirement Validation

Feasibility study

When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software.

Referencing to this information, the analysts does a detailed study about whether the desired system and its functionality are feasible to develop.

This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity and integration ability.

The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Requirement Gathering

If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

Software Requirement Specification

SRS is a document created by system analyst after the requirements are collected from various stakeholders.

SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.

- Conditional and mathematical notations for DFDs etc.

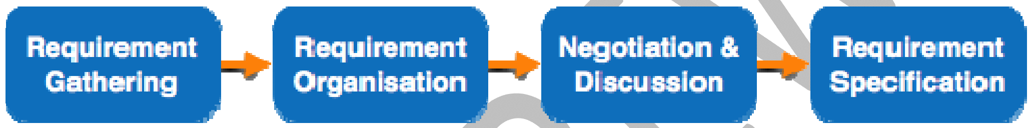
Software Requirement Validation

After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly. This results in huge increase in cost if not nipped in the bud. Requirements can be checked against following conditions -

- If they can be practically implemented
- If they are valid and as per functionality and domain of software
- If there are any ambiguities
- If they are complete
- If they can be demonstrated

Requirement Elicitation Process

Requirement elicitation process can be depicted using the following diagram:



- **Requirements gathering -** The developers discuss with the client and end users and know their expectations from the software.
- **Organizing Requirements -** The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
- **Negotiation & discussion -** If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, if they are, it is then negotiated and discussed with stakeholders. Requirements may then be prioritized and reasonably compromised.

  The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.

- **Documentation -** All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

Requirement Elicitation Techniques

Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users and others who have a stake in the software system development.

There are various ways to discover requirements

Interviews

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as:

- Structured (closed) interviews, where every single information to gather is decided in advance, they follow pattern and matter of discussion firmly.
- Non-structured (open) interviews, where information to gather is not decided in advance, more flexible and less biased.
- Oral interviews
- Written interviews
- One-to-one interviews which are held between two persons across the table.
- Group interviews which are held between groups of participants. They help to uncover any missing requirement as numerous people are involved.

Surveys

Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

Questionnaires

A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled.

A shortcoming of this technique is, if an option for some issue is not mentioned in the questionnaire, the issue might be left unattended.

Task analysis

Team of engineers and developers may analyze the operation for which the new system is required. If the client already has some software to perform certain operation, it is studied and requirements of proposed system are collected.

Domain Analysis

Every software falls into some domain category. The expert people in the domain can be a great help to analyze general and specific requirements.

Brainstorming

An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

Prototyping

Prototyping is building user interface without adding detail functionality for user to interpret the features of intended software product. It helps giving better idea of requirements. If there is no software installed at client's end for developer's reference and the client is not aware of its own requirements, the developer creates a prototype based on initially mentioned requirements. The prototype is shown to the client and the feedback is noted. The client feedback serves as an input for requirement gathering.

Observation

Team of experts visit the client's organization or workplace. They observe the actual working of the existing installed systems. They observe the workflow at client's end and how execution problems are dealt. The team itself draws some conclusions which aid to form requirements expected from the software.

Software Requirements Characteristics

Gathering software requirements is the foundation of the entire software development project. Hence they must be clear, correct and well-defined.

A complete Software Requirement Specifications must be:

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

Software Requirements

We should try to understand what sort of requirements may arise in the requirement elicitation phase and what kinds of requirements are expected from the software system.

Broadly software requirements should be categorized in two categories:

Functional Requirements

Requirements, which are related to functional aspect of software fall into this category.

They define functions and functionality within and from the software system.

Examples -

- Search option given to user to search from various invoices.
- User should be able to mail any report to management.
- Users can be divided into groups and groups can be given separate rights.
- Should comply business rules and administrative functions.
- Software is developed keeping downward compatibility intact.

Non-Functional Requirements

Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.

Non-functional requirements include -

- Security
- Logging
- Storage
- Configuration
- Performance
- Cost
- Interoperability
- Flexibility
- Disaster recovery
- Accessibility

Requirements are categorized logically as

- **Must Have** : Software cannot be said operational without them.
- **Should have** : Enhancing the functionality of software.
- **Could have** : Software can still properly function with these requirements.
- **Wish list** : These requirements do not map to any objectives of software.

While developing software, ‘Must have’ must be implemented, ‘Should have’ is a matter of debate with stakeholders and negation, whereas ‘could have’ and ‘wish list’ can be kept for software updates.

User Interface requirements

UI is an important part of any software or hardware or hybrid system. A software is widely accepted if it is -

- easy to operate
- quick in response
- effectively handling operational errors
- providing simple yet consistent user interface

User acceptance majorly depends upon how user can use the software. UI is the only way for users to perceive the system. A well performing software system must also be equipped with attractive, clear, consistent and responsive user interface. Otherwise the functionalities of software system can not be used in convenient way. A system is said be good if it provides means to use it efficiently. User interface requirements are briefly mentioned below -

- Content presentation
- Easy Navigation
- Simple interface
- Responsive
- Consistent UI elements
- Feedback mechanism
- Default settings
- Purposeful layout
- Strategical use of color and texture.
- Provide help information
- User centric approach
- Group based view settings.

Software System Analyst

System analyst in an IT organization is a person, who analyzes the requirement of proposed system and ensures that requirements are conceived and documented properly & correctly. Role of an analyst starts during Software Analysis Phase of SDLC. It is the responsibility of analyst to make sure that the developed software meets the requirements of the client.

System Analysts have the following responsibilities:

- Analyzing and understanding requirements of intended software
- Understanding how the project will contribute in the organization objectives
- Identify sources of requirement
- Validation of requirement
- Develop and implement requirement management plan
- Documentation of business, technical, process and product requirements
- Coordination with clients to prioritize requirements and remove and ambiguity
- Finalizing acceptance criteria with client and other stakeholders

Software Metrics and Measures

Software Measures can be understood as a process of quantifying and symbolizing various attributes and aspects of software.

Software Metrics provide measures for various aspects of software process and software product.

Software measures are fundamental requirement of software engineering. They not only help to control the software development process but also aid to keep quality of ultimate product excellent.

According to Tom DeMarco, a (Software Engineer), “You cannot control what you cannot measure.” By his saying, it is very clear how important software measures are.

Let us see some software metrics:

- **Size Metrics -** LOC (Lines of Code), mostly calculated in thousands of delivered source code lines, denoted as KLOC.

  Function Point Count is measure of the functionality provided by the software. Function Point count defines the size of functional aspect of software.

- **Complexity Metrics -** McCabe’s Cyclomatic complexity quantifies the upper bound of the number of independent paths in a program, which is perceived as complexity of the program or its modules. It is represented in terms of graph theory concepts by using control flow graph.
- **Quality Metrics -** Defects, their types and causes, consequence, intensity of severity and their implications define the quality of product.

  The number of defects found in development process and number of defects reported by the client after the product is installed or delivered at client-end, define quality of product.

- **Process Metrics -** In various phases of SDLC, the methods and tools used, the company standards and the performance of development are software process metrics.
- **Resource Metrics -** Effort, time and various resources used, represents metrics for resource measurement.

Modeling and simulation

Modeling and simulation (**M&S**) refers to using models – physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process – as a basis for simulations – methods for implementing a model (either statically or) over time – to develop data as a basis for managerial or technical decision making. M&S helps getting information about how something will behave without actually testing it in real life. For instance, to determine which type of spoiler would improve traction the most while designing a race car, a computer simulation of the car could be used to estimate the effect of different spoiler shapes on the coefficient of friction in a turn. Useful insights about different decisions in the design could be gleaned without actually building the car.

The use of M&S within engineering is well recognized. Simulation technology belongs to the tool set of engineers of all application domains and has been included in the body of knowledge of engineering management. M&S helps to reduce costs, increase the quality of products and systems, and document and archive lessons learned.

M&S is a discipline on its own. Its many application domains often lead to the assumption that M&S is pure application. This is not the case and needs to be recognized by engineering management experts who want to use M&S. To ensure that the results of simulation are applicable to the real world, the engineering manager must understand the assumptions, conceptualizations, and implementation constraints of this emerging field.

Development steps
• Model-based control engineering
• Modeling and simulation
• Systems platform: hardware, systems software.

Controls development cycle
• Analysis and modeling
– Control algorithm design using a simplified model
– System trade study - defines overall system design
• Simulation
– Detailed model: physics, or empirical, or data driven
– Design validation using detailed performance model
• System development
– Control application software
– Real-time software platform
– Hardware platform
• Validation and verification
– Performance against initial specs
– Software verification
– Certification/commissioning

<u>PARTITIONING SOFTWARE:</u>

Problem Partitioning

- When solving a small problem, the entire problem can be tackled at once. The complexity of large problems and the limitations of human minds do not allow large problems to be treated as huge monoliths.
- As basic aim of problem analysis is to obtain a clear understanding of the needs of the clients and the users.

- Frequently the client and the users do not understand or know all their needs, because the potential of the new system is often not fully appreciated.
- The analysts have to ensure that the real needs of the clients and the users are uncovered, even if they don't know them clearly.
- That is, the analysts are not just collecting and organizing information about the client's organization and its processes, but they also act as consultants who play an active role of helping the clients and users identify their needs.
- For solving larger problems, the basic principle is the time-tested principle of "divide and conquer.

"divide into smaller pieces, so that each piece can be conquered separately."

- For software design, partition the problem into sub problems and then try to understand each sub problem and its relationship to other sub problems in an effort to understand the total problem.
- That is goal is to divide the problem into manageably small pieces that can be solved separately, because the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.
- The different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem.
- Problem partitioning also aids design verification.
- The concepts of state and projection can sometimes also be used effectively in the partitioning process.

Software prototyping

Software prototyping is the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.

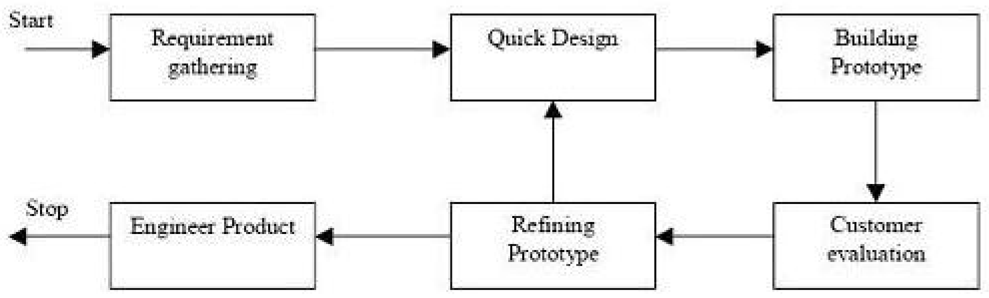
A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Prototyping has several benefits: The software designer and implementer can get valuable feedback from the users early in the project. The client and the contractor can compare if the software made matches the software specification, according to which the software program is built. It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met

The basic idea in **Prototype model** is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. Prototype model is a software development model. By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system. Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.

The prototype are usually not complete systems and many of the details are not built in the prototype. The goal is to provide a system with overall functionality.

Diagram of Prototype model:



Prototyping Model

Advantages of Prototype model:

- Users are actively involved in the development
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily
- Confusing or difficult functions can be identified.
- Requirements validation, Quick implementation of, incomplete, but functional, application.

Disadvantages of Prototype model:

- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Incomplete application may cause application not to be used as the full system was designed.
- Incomplete or inadequate problem analysis.

When to use Prototype model:

- Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
- Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
- Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

SPECIFICATION PRINCIPLES:

These are:
(1) manage using a phased life-cycle plan;
(2) perform continuous validation;
(3)maintain disciplined product control;
(4) use modern programming practices;
(5) maintain clear account ability for results
(6) use better and fewer people;
(7)maintain a commitment to improve the process.

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

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

Data Flow Diagram

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

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

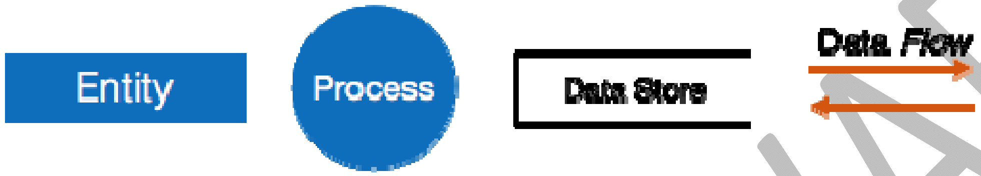
Types of DFD

Data Flow Diagrams are either Logical or Physical.

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

DFD Components

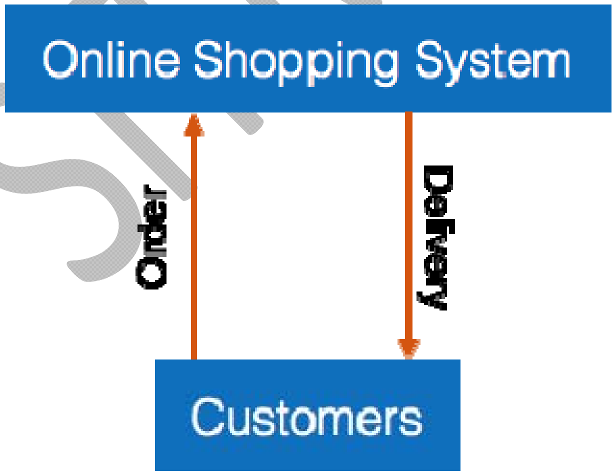
DFD can represent Source, destination, storage and flow of data using the following set of components -



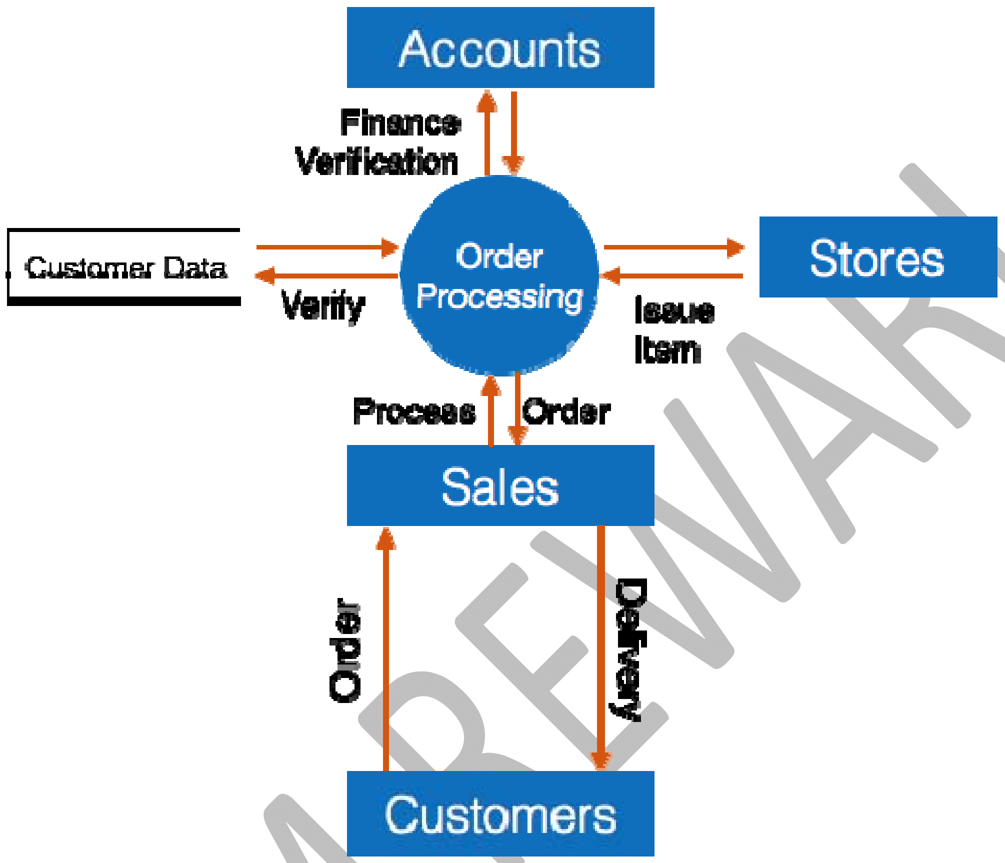
- **Entities** - Entities are source and destination of information data. Entities are represented by a rectangles with their respective names.
- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
- **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

Levels of DFD

- **Level 0** - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



- **Level 1** - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.



- **Level 2** - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

  Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

Structure Charts

Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.Structure chart represents hierarchical structure of modules. At each layer a specific task is performed.

Here are the symbols used in construction of structure charts -

- **Module** - It represents process or subroutine or task. A control module branches to more than one sub-module. Library Modules are re-usable and invokable from any module.

Principles of Software Design & Concepts in Software Engineering

Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system. **IEEE** defines software design as 'both a process of defining, the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.'

In the design phase, many critical and strategic decisions are made to achieve the desired functionality and quality of the system. These decisions are taken into account to successfully develop the software and carry out its maintenance in a way that the quality of the end product is improved.

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice.

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Design Levels

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design-** The high-level design breaks the ‘single entity-multiple component’ concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

Concurrency

Back in time, all software are meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.

It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Example

The spell check feature in word processor is a module of software, which runs along side the word processor itself.

Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

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

Coupling

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

There are five levels of coupling, namely -

- **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

Design Verification

The output of software design process is design documentation, pseudo codes, detailed logic diagrams, process diagrams, and detailed description of all functional or non-functional requirements.

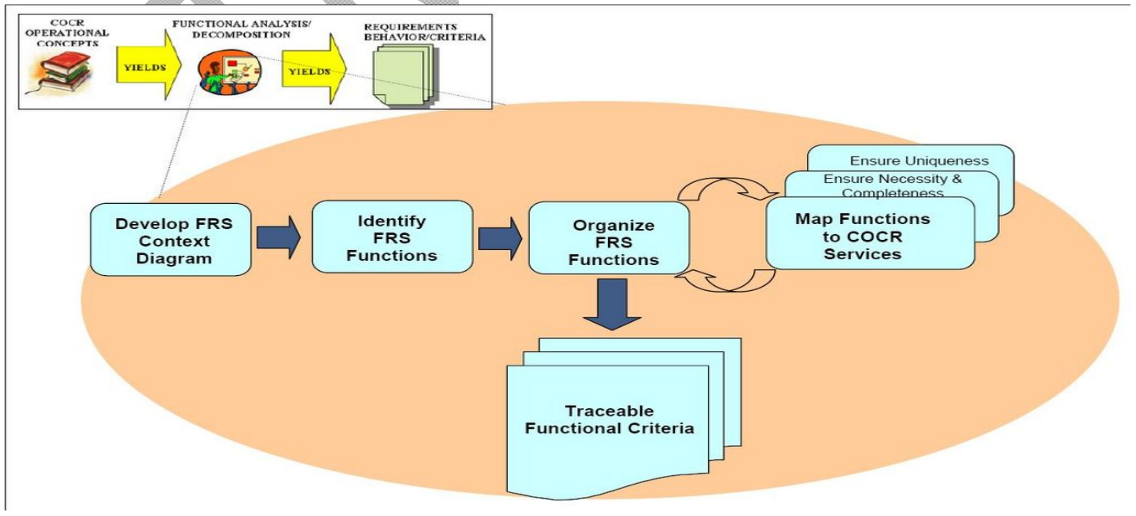
The next phase, which is the implementation of software, depends on all outputs mentioned above.

It is then becomes necessary to verify the output before proceeding to the next phase. The early any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of design phase are in formal notation form, then their associated tools for verification should be used otherwise a thorough design review can be used for verification and validation.

By structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy and quality.

Structured analysis

In software engineering, **structured analysis** (SA) and **structured design** (SD) are methods for analyzing business requirements and developing specifications for converting practices into computer programs, hardware configurations, and related manual procedures.


COCR OPERATIONAL CONCEPTS
YIELDS
FUNCTIONAL ANALYSIS/ DECOMPOSITION
YIELDS
REQUIREMENTS BEHAVIOR/CRITERIA
Develop FRS Context Diagram
Identify FRS Functions
Organize FRS Functions
Ensure Uniqueness
Ensure Necessity & Completeness
Map Functions to COCR Services
Traceable Functional Criteria

Approach

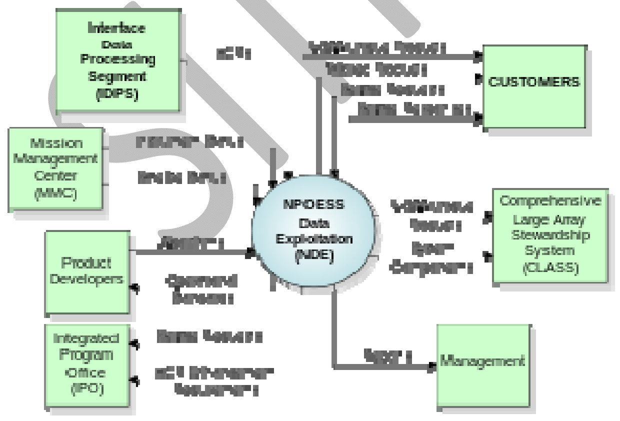
Structured Analysis views a system from the perspective of the data flowing through it. The function of the system is described by processes that transform the data flows. Structured analysis takes advantage of information hiding through successive decomposition (or top down) analysis. This allows attention to be focused on pertinent details and avoids confusion from looking at irrelevant details. As the level of detail increases, the breadth of information is reduced. The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements.

De Marco's approach consists of the following objects:

- Context diagram
- Data flow diagram
- Process specifications
- Data dictionary

Hereby the data flow diagrams (DFDs) are directed graphs. The arcs represent data, and the nodes (circles or bubbles) represent processes that transform the data. A process can be further decomposed to a more detailed DFD which shows the subprocesses and data flows within it. The subprocesses can in turn be decomposed further with another set of DFDs until their functions can be easily understood. Functional primitives are processes which do not need to be decomposed further. Functional primitives are described by a process specification (or mini-spec). The process specification can consist of pseudo-code, flowcharts, or structured English. The DFDs model the structure of the system as a network of interconnected processes composed of functional primitives. The data dictionary is a set of entries (definitions) of data flows, data elements, files, and databases. The data dictionary entries are partitioned in a top-down manner. They can be referenced in other data dictionary entries and in data flow diagrams.

Context diagram

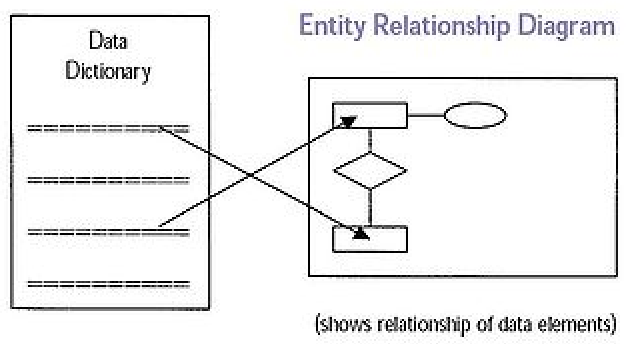

Interface Data Processing Segment (IDPS)
Mission Management Center (MMC)
Product Developers
Integrated Program Office (IPO)
NPOESS Data Exploitation (NDE)
CUSTOMERS
Comprehensive Large Array Stewardship System (CLASS)
Management

Example of a system context diagram.

Context diagrams are diagrams that represent the actors outside a system that could interact with that system. This diagram is the highest level view of a system, similar to block diagram, showing a, possibly software-based, system as a whole and its inputs and outputs from/to external factors.

This type of diagram according to Kossiakoff (2003) usually "pictures the system at the center, with no details of its interior structure, surrounded by all its interacting systems, environment and activities. The objective of a system context diagram is to focus attention on external factors and events that should be considered in developing a complete set of system requirements and constraints".[13] System context diagrams are related to data flow diagram, and show the interactions between a system and other actors which the system is designed to face. System context diagrams can be helpful in understanding the context in which the system will be part of software engineering.

Data dictionary

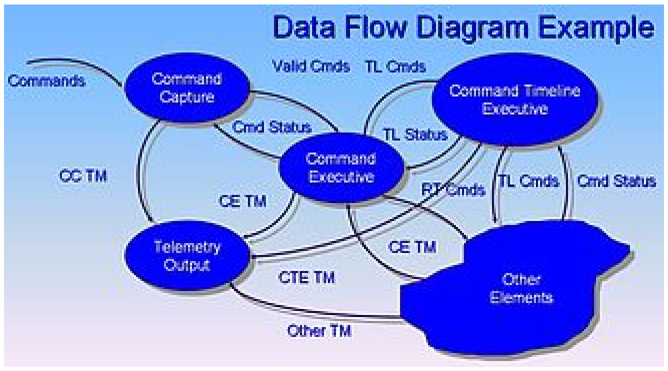

Data Dictionary
Entity Relationship Diagram
(shows relationship of data elements)

Entity relationship diagram, essential for the design of database tables, extracts, and metadata.

A data dictionary or *database dictionary* is a file that defines the basic organization of a database.[14] A database dictionary contains a list of all files in the database, the number of records in each file, and the names and types of each data field. Most database management systems keep the data dictionary hidden from users to prevent them from accidentally destroying its contents. Data dictionaries do not contain any actual data from the database, only bookkeeping information for managing it. Without a data dictionary, however, a database management system cannot access data from the database.[14]

Database users and application developers can benefit from an authoritative data dictionary document that catalogs the organization, contents, and conventions of one or more databases. This typically includes the names and descriptions of various tables and fields in each database, plus additional details, like the type and length of each data element. There is no universal standard as to the level of detail in such a document, but it is primarily a distillation of metadata about database structure, not the data itself. A data dictionary document also may include further information describing how data elements are encoded. One of the advantages of well-designed data dictionary documentation is that it helps to establish consistency throughout a complex database, or across a large collection of federated databases.

Data flow diagrams



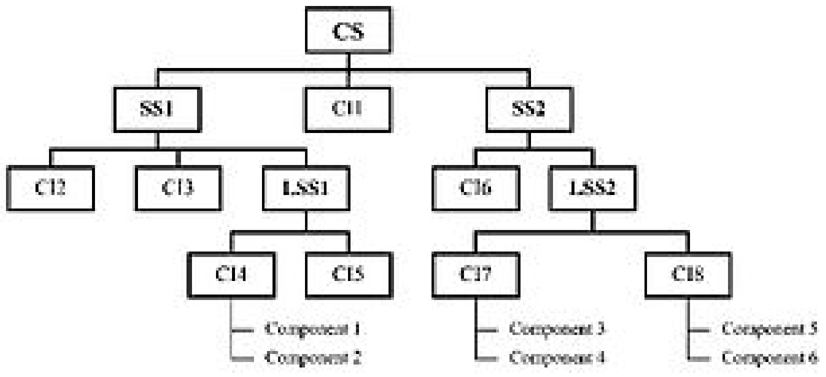
Data flow diagram example.

A data flow diagram (DFD) is a graphical representation of the "flow" of data through an information system. It differs from the system flowchart as it shows the flow of data through processes instead of computer hardware. Data flow diagrams were invented by Larry Constantine, developer of structured design, based on Martin and Estrin's "data flow graph" model of computation.[18]

It is common practice to draw a system context diagram first which shows the interaction between the system and outside entities. The DFD is designed to show how a system is divided into smaller portions and to highlight the flow of data between those parts. This context-level data flow diagram is then "exploded" to show more detail of the system being modeled.

Data flow diagrams (DFDs) are one of the three essential perspectives of structured systems analysis and design method (SSADM). The sponsor of a project and the end users will need to be briefed and consulted throughout all stages of a system's evolution. With a data flow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. The old system's data flow diagrams can be drawn up and compared with the new system's data flow diagrams to draw comparisons to implement a more efficient system. Data flow diagrams can be used to provide the end user with a physical idea of where the data they input ultimately has an effect upon the structure of the whole system from order to dispatch to recook. How any system is developed can be determined through a data flow diagram.

Structure chart



A configuration system structure chart.

A structure chart (SC) is a chart that shows the breakdown of the configuration system to the lowest manageable levels. This chart is used in structured programming to arrange the program modules in a tree structure. Each module is represented by a box which contains the name of the modules. The tree structure visualizes the relationships between the modules.

Structure charts are used in structured analysis to specify the high-level design, or architecture, of a computer program. As a design tool, they aid the programmer in dividing and conquering a large software problem, that is, recursively breaking a problem down into parts that are small enough to be understood by a human brain. The process is called top-down design, or functional decomposition. Programmers use a structure chart to build a program in a manner similar to how an architect uses a blueprint to build a house. In the design stage, the chart is drawn and used as a way for the client and the various software designers to communicate. During the actual building of the program (implementation), the chart is continually referred to as the master-plan.

Structured design

Structured design (SD) is concerned with the development of modules and the synthesis of these modules in a so-called "module hierarchy". In order to design optimal module structure and interfaces two principles are crucial:

- *Cohesion* which is "concerned with the grouping of functionally related processes into a particular module", and
- *Coupling* relates to "the flow of information or parameters passed between modules. Optimal coupling reduces the interfaces of modules and the resulting complexity of the software".[10]

Structured design was developed by Larry Constantine in the late 1960s, then refined and published with collaborators in the 1970s;has proposed his own approach which consists of three main objects :

- structure charts
- module specifications
- data dictionary.

The structure chart aims to show "the module hierarchy or calling sequence relationship of modules. There is a module specification for each module shown on the structure chart. The module specifications can be composed of pseudo-code or a program design language. The data dictionary is like that of structured analysis. At this stage in the software development lifecycle, after analysis and design have been performed, it is possible to automatically generate data type declarations", and procedure or subroutine templates.

Structured query language

The structured query language (SQL) is a standardized language for querying information from a database. SQL was first introduced as a commercial database system in 1979 and has since been the favorite query language for database management systems running on minicomputers and mainframes. Increasingly, however, SQL is being supported by PC database systems because it supports distributed databases (see definition of distributed database). This enables several users on a computer network to access the same database simultaneously

Process specification

Process specification is a generic term for the specification of a process. Its context is not unique to "business activity" but can be applied to any organizational activity.

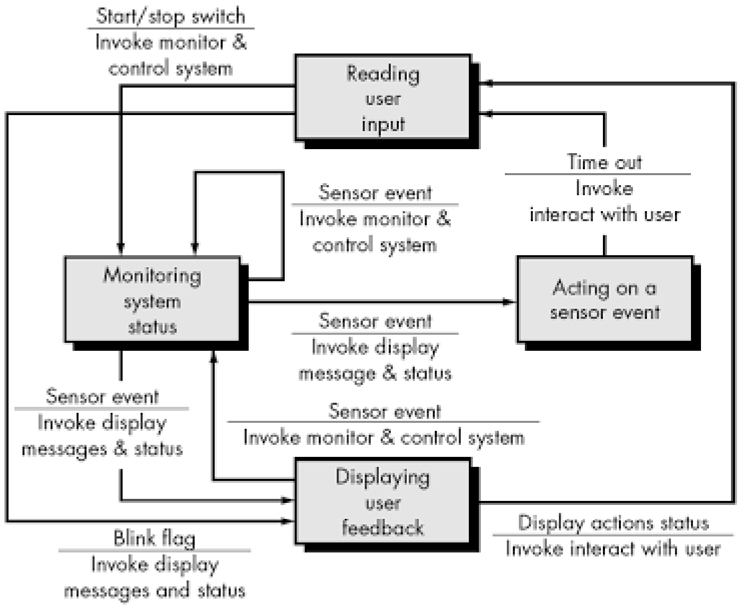
Within some structured methods, the capitalized term **Process Specification** refers to a description of the procedure to be followed by an actor within an elementary level business activity, as represented on a process model such as a dataflow diagram or IDEF0 model. A common alias is minispec short for miniature specification.

The process specification defines what must be done in order to transform inputs into outputs. It is a detailed set of instructions outlining a business procedure that each elementary level business activity is expected to carry out. Process specifications are commonly included as an integral component of a requirements document in systems development.

There are a variety of approaches that can be used to produce a process specification including

- decision tables
- structured English (favored technique of most systems analysts)
- pre/post conditions
- use cases, basic course or events/alternate paths in use cases
- flowcharts.
- Nassi–Shneiderman diagram
- UML Activity diagram

- The control specification (CSPEC) represents the behavior of the system (at the level from which it has been referenced) in two different ways. The CSPEC contains a state transition diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior. It is now time to consider an example of this important modeling notation for structured analysis.
- Figure below depicts a state transition diagram for the level 1 control flow model for SafeHome. The labeled transition arrows indicate how the system responds to events as it traverses the four states defined at this level. By studying the STD, a software engineer can determine the behavior of the system and, more important, can ascertain whether there are "holes" in the specified behavior. For example, the STD indicates that the only transition from the reading user input state occurs when the start/stop switch is encountered and a transition to the monitoring system status state occurs. Yet, there appears to be no way, other than the occurrence of sensor event, that will allow the system to return to reading user input. This is an error in specification and would, we hope, be uncovered during review and corrected.


Start/stop switch
Invoke monitor & control system
Reading user input
Time out
Invoke interact with user
Sensor event
Invoke monitor & control system
Monitoring system status
Acting on a sensor event
Sensor event
Invoke display message & status
Sensor event
Invoke display messages & status
Sensor event
Invoke monitor & control system
Displaying user feedback
Display actions status
Invoke interact with user
Blink flag
Invoke display messages and status

Data dictionary

A **data dictionary** is a collection of descriptions of the **data** objects or items in a **data** model for the benefit of programmers and others who need to refer to them. A first step in analyzing a system of objects with which users interact is to identify each object and its relationship to other objects

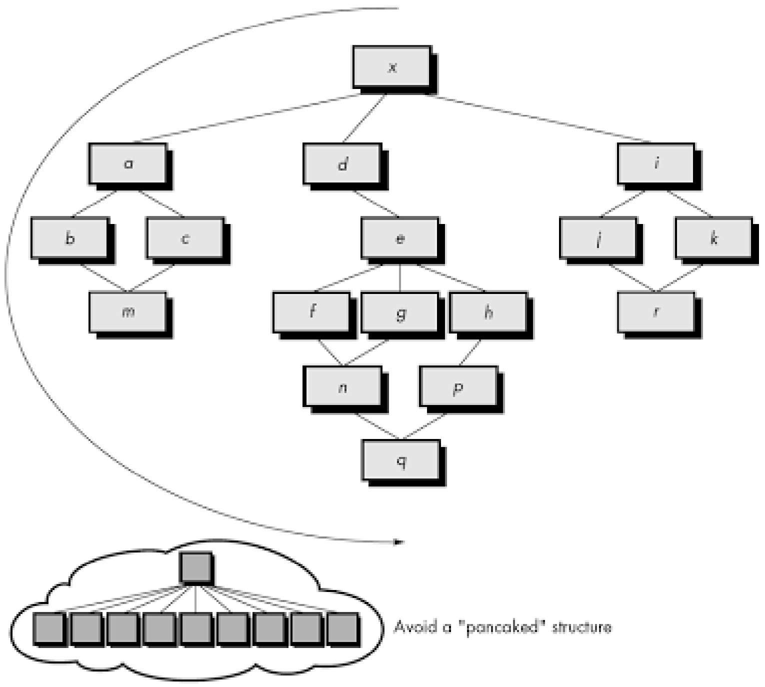
A **data dictionary**, or metadata repository, as defined in the *IBM Dictionary of Computing*, is a "centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format." *Oracle* defines it as a collection of tables with metadata. The term can have one of several closely related meanings pertaining to databases and database management systems (DBMS):

- A document describing a database or collection of databases
- An integral component of a DBMS that is required to determine its structure
- A piece of middleware that extends or supplants the native data dictionary of a DBMS

Software Engineering-Design Heuristics For Effective Modularity

Once program structure has been developed, effective modularity can be achieved by applying the design concepts introduced earlier in this chapter. The program structure can be manipulated according to the following set of heuristics:

1. Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion. Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence. An exploded module becomes two or more modules in the final program structure.
An imploded module is the result of combining the processing implied by two or more modules. An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.



2. Attempt to minimize structures with high fan-out; strive for fan-in as depth increases. The structure shown inside the cloud in figure does not make effective use of factoring. All modules are “pancaked” below a single control module. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

3. Keep the scope of effect of a module within the scope of control of that module. The scope of effect of module e is defined as all other modules that are affected by a decision made in module e. The scope of control of module e is all modules that are subordinate and ultimately subordinate to module e. Referring to figure, if module e makes a decision that affects module r, we have a violation of this heuristic, because module r lies outside the scope of control of module e.

4. Evaluate module interfaces to reduce complexity and redundancy and improve consistency. Module interface complexity is a prime cause of software errors .Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly unrelated data passed via an argument list or other technique) is an indication
of low cohesion. The module in question should be reevaluated.

5. Define modules whose function is predictable, but avoid modules that are overly restrictive. A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use.

A module that restricts processing to a single subfunction exhibits high cohesion and is viewed with favor by a designer. However, a module that arbitrarily restricts the size of a local data structure, options within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

6. Strive for “controlled entry” modules by avoiding "pathological connections." This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. Pathological connection refers to branches or references into the middle of a module.

Software Design Documentation (SDD)

The information that the software design document should describe depends on various factors including the type of software being developed and the approach used in its development. A number of standards have been suggested to develop a software design document. However, the most widely used standard is by IEEE, which acts as a general framework. This general framework can be customized and adapted to meet the needs of a particular organization. This template consists of several sections, which are listed below.

1. **Scope:** Identifies the release or version of the system being designed. The system is divided into modules; the relationship between them and functionalities will be defined. Every iteration of the SDD document describes and identifies the software modules to be added or changed in a release.

2. **References:** Lists references (both hardware and software documents and manuals) used in the creation of the SDD that may be of use to the designer, programmer, user, or management personnel. This document is also considered useful for the readers of the document. In this section, any references made to the other documents including references to related project documents, especially the SRS are also listed. The existing software documentation (if any) is also listed.
3. **Definition:** Provides a glossary of technical terms used in the document along with their definitions.
4. **Purpose:** States the purpose of this document and its intended audience. This is meant primarily for individuals who will be implementing the system.
5. **Design description information content:** Consists of the following subsections.

. **Introduction:** Since SDD represents the software design that is to be implemented, it should describe the design entities into which the system has been partitioned along with their significant properties and relationships.

1. **Design entity:** It is a software design component that is different from other design entities in terms of structure and function. The objective of creating design entities is to partition the system into a set of components that can be implemented and modified independently. Note that each design entity is assigned with a unique name and serves a specific purpose and function but all possess some common characteristics.
2. **Design entity attributes:** They are properties of the design entity and provide some factual information regarding the entity. Every attribute has an attached description, which includes references and design considerations.

Table Attributes and Description

Attributes	Description
Identification	Identifies name of the entity. All the entities have a unique name.
Type	Describes the kind of entity. This specifies the nature of the entity.
Purpose	Specifies why the entity exists.
Function	Specifies what the entity does.
Subordinates	Identifies sub-ordinate entity of an entity.
Dependencies	Describes relationships that exist between one entity and other entities.
Interface	Describes how entities interact among themselves.
Resources	Describes elements used by the entity that are external to the design.
Processing	Specifies rules used to achieve the specified functions.
Data	Identifies data elements that form part of the internal entity.

6. Design description organization: Consists of the following subsection.

7. Design views: They describe the software design in a comprehensive manner so that the process of information access and integration is simplified. The design of software can be viewed in multiple ways and each design view describes a distinct aspect of the system. Table lists various design views and their attributes.

Table Design Views and their Description

Design View	Description	Attribute
Decomposition description	Partitions the system into design entities.	Identification, type, purpose, function, and subordinate
Dependency description	Describes relationships between entities.	Identification, type, purpose, dependencies, and resources
Interface description	Consists of list that is required by the stakeholders (designers, developers, and testers) in order to design entities.	Identification, function and interfaces
Detail description	Describesinternal details of the design entity.	Identification, processing, and data

Software quality

In the context of software engineering, **software quality** refers to two related but distinct notions that exist wherever quality is defined in a business context:

- Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product;
- Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

Structural quality is evaluated through the analysis of the software inner structure, its source code, at the unit level, the technology level and the system level, which is in effect how its architecture adheres to sound principles of software architecture outlined in a paper on the topic by OMG. In contrast, functional quality is typically enforced and measured through software testing.

In the context of software engineering, **software quality** measures how well software is designed (*quality of design*), and how well the software conforms to that design (*quality of conformance*), although there are several different definitions. It is often described as the 'fitness for purpose' of a piece of software.

Whereas *quality of conformance* is concerned with implementation (see Software Quality Assurance), *quality of design* measures how valid the design and requirements are in creating a worthwhile product

One of the challenges of software quality is that "everyone feels they understand it".

Software quality may be defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

The three key points in this definition:

1. Software requirements are the foundations from which quality is measured.

   Lack of conformance to requirement is lack of quality.

2. Specified standards define a set of development criteria that guide the manager is software engineering.

   If criteria are not followed lack of quality will usually result.

3. A set of implicit requirements often goes unmentioned, for example ease of use, maintainability etc.

   If software confirms to its explicit requirement but fails to meet implicit requirements, software quality is suspected.

A definition in Steve McConnell's *Code Complete* divides software into two pieces: **internal** and **external quality characteristics**. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.

Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better." This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality.

<u>Software product quality</u>

- Correctness

- Product quality
    - conformance to requirements or program specification; related to Reliability
- Scalability
- Completeness
- Absence of bugs
- Fault-tolerance
    - Extensibility
    - Maintainability
- Documentation

The Consortium for IT Software Quality (CISQ) was launched in 2009 to standardize the measurement of software product quality. The Consortium's goal is to bring together industry executives from Global 2000 IT organizations, system integrators, outsourcers, and package vendors to jointly address the challenge of standardizing the measurement of IT software quality and to promote a market-based ecosystem to support its deployment.

Software reliability

Software **reliability** is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

Goal of reliability

The need for a means to objectively determine software reliability comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both lay-persons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviour, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Reliability in program development

Requirements

A program cannot be expected to work as desired if the developers of the program do not, in fact, know the program's desired behavior in advance, or if they cannot at least determine its desired behaviour in parallel with development, in sufficient detail. What level of detail is considered sufficient is hotly debated. The idea of perfect detail is attractive, but may be impractical, if not actually impossible. This is because the desired behavior tends to change as the possible range of the behaviour is determined through actual attempts, or more accurately, failed attempts, to achieve it.

Design

While requirements are meant to specify what a program should do, design is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program.

Programming

The history of computer programming language development can often be best understood in the light of attempts to master the complexity of computer programs, which otherwise becomes more difficult to understand in proportion (perhaps exponentially) to the size of the programs. (Another way of looking at the evolution of programming languages is simply as a way of getting the computer to do more and more of the work, but this may be a different way of saying the same thing). Lack of understanding of a program's overall structure and functionality is a sure way to fail to detect errors in the program, and thus the use of better languages should, conversely, reduce the number of errors by enabling a better understanding.

Software Build and Deployment

Many programming languages such as C and Java require the program "source code" to be translated in to a form that can be executed by a computer. This translation is done by a program called a compiler. Additional operations may be involved to associate, bind, link or package files together in order to create a usable runtime configuration of the software application. The totality of the compiling and assembly process is generically called "building" the software.

The software build is critical to software quality because if any of the generated files are incorrect the software build is likely to fail. And, if the incorrect version of a program is inadvertently used, then testing can lead to false results.