الجامعة التكنولوجية

قسم علوم الحاسوب

Software Design

3th class –Software Branch

د. محمد غني علوان

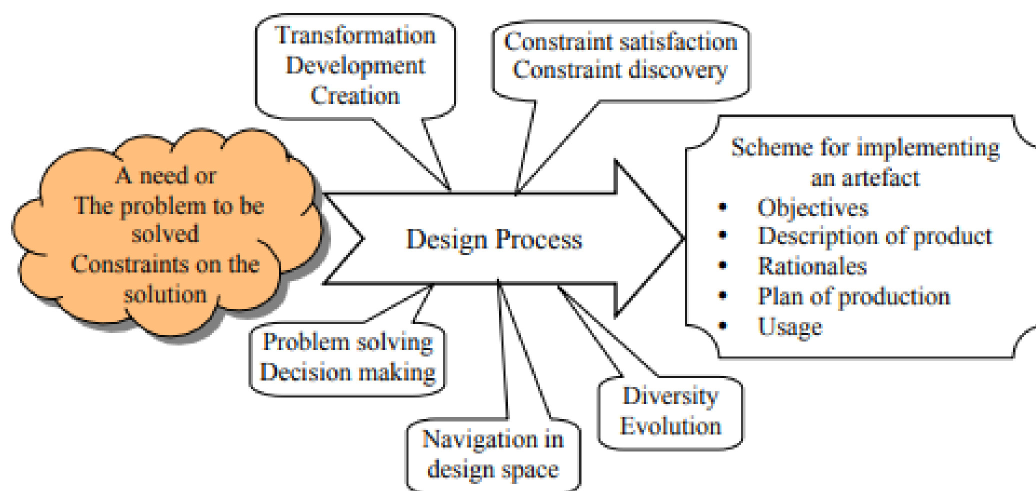
Second Course 2023-2024

Software Design

1. Definition of design Concepts:

There are two facets of the concept of design. **Firstly**, a design is a plan to bring about a man-made product. Such a plan must achieve a prescribed goal and satisfy certain constraints. **Secondly**, it is a process of the creative development of such a plan. During this process, the designer must use related scientific principles, technical information and imagination to discover constraints and to solve the design problem. Therefore, we can define engineering design as follows.

Engineering design is the use of scientific principles and technical information in the creative development of a plan to bring about a man-made product to achieve a prescribed goal with certain specified constraints. The consequence of the implementation of the design will bring changes to the environment, while the environment of the designer influences the design itself. Software design is a branch of engineering design where the product to bring about is software.



Basic concepts of design

As depicted in Figure , design activities have the following characteristics .Design starts with a need and requires intention. It results in a scheme for implementing an artefact. It involves transformations and the generation of new ideas is fundamental to all designs. Design is goal directed problem solving and decision making. It must satisfy the constraints and the requirements. The design process is also a constraint discovery process. Design is to achieve optimality in a solution space of diversity. An engineering design should contain at least **five basic elements:**

1. The objectives of the design.
2. A description of the designed product.
3. The rationale of the design.
4. A plan of the production.
5. The designated usage of the product.

There are a number of **factors** that affect design processes and their outcomes. These factors include:

1. The requirements to be satisfied by the design.
2. The design was made and evaluated.
3. The value of the product.
4. The resource available to the design, manufacture and use of the product.
5. The features, or functions, of the product.
6. The process of design.
7. The consequence of design, i.e. the change to be brought about.
8. The people involved in the design process and their working relationships.
9. The competence of the designer.
10. The service of the designed products. These factors interrelate with each other.

Design is defined as both “the process of defining the architecture, components, interfaces, and other characteristics of a system or component” and “the result of [that] process” [1]. Viewed as a process, software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software’s internal structure that will serve as the basis for its construction.

Software design plays an important role in developing software: during software design, software engineers produce various models that form a kind of blueprint of the solution to be implemented. We can analyze and evaluate these models to determine whether or not they will allow us to fulfill the various requirements.

2. A. Fundamental design issues (e.g., persistent data, storage management, and exceptions)

The concepts, notions, and terminology introduced here form an underlying basis for understanding the role and scope of software design.

A. General Design Concepts:

Software design is an important part of the software development process. To understand the role of software design, we must see how it fits in the software development life cycle. Thus, it is important to understand the major characteristics of software requirements analysis, software design, software construction, software testing, and software maintenance.

B. Context of Software Design:

Software design is an important part of the software development process. To understand the role of software design, we must see how it fits in the software development life cycle. Thus, it is important to understand the major characteristics of software requirements analysis, software design, software construction, software testing, and software maintenance.

C. Software Design Process:

Software design is generally considered a two-step process:

- Architectural design (also referred to as high-level design and top-level design) describes how software is organized into components.
- Detailed design describes the desired behavior of these components.

The output of these two processes is a set of models and artifacts that record the major decisions that have been taken, along with an explanation of the rationale for each nontrivial decision. By recording the rationale, long-term maintainability of the software product is enhanced.

D. Software Design Principles:

A principle is "a comprehensive and fundamental law, doctrine, or assumption". Software design principles are key notions that provide the basis for many different software design approaches and concepts. Software design principles include abstraction; coupling and cohesion; decomposition and modularization; encapsulation/information hiding; separation of interface and implementation; sufficiency, completeness, and primitiveness; and separation of concerns.

- **Abstraction** is "a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information" . In the context of software design, two key abstraction mechanisms are parameterization and specification. Abstraction by parameterization abstracts from the details of data representations by representing the data as named parameters. Abstraction by specification leads to three major kinds of abstraction: procedural abstraction, data abstraction, and control (iteration) abstraction.
- Coupling and Cohesion. Coupling is defined as “a measure of the interdependence among modules in a computer program,” whereas cohesion is defined as “a measure of the strength of association of the elements within a module”.[

- Decomposition and modularization. Decomposing and modularizing means that large software is divided into a number of smaller named components having well-defined interfaces that describe component interactions. Usually the goal is to place different functionalities and responsibilities in different components.
- Encapsulation and information hiding means grouping and packaging the internal details of an abstraction and making those details inaccessible to external entities.
- Separation of interface and implementation. Separating interface and implementation involves defining a component by specifying a public interface (known to the clients) that is separate from the details of how the component is realized.
- Sufficiency, completeness, and primitiveness. Achieving sufficiency and completeness means ensuring that a software component captures all the important characteristics of an abstraction and nothing more. Primitiveness means the design should be based on patterns that are easy to implement.
- Separation of concerns. A concern is an "area of interest with respect to a software design". A design concern is an area of design that is relevant to one or more of its stakeholders. Each architecture view frames one or more concerns. Separating concerns by views allows interested stakeholders to focus on a few things at a time and offers a means of managing complexity.

2.B.Key Issues in Software Design:

A number of key issues must be dealt with when designing software. Some are quality concerns that all software must address—for example, performance, security, reliability, usability, etc. Another important issue is how to decompose, organize, and package software components. This is so fundamental that all design approaches address it in one way or another . A number of these key, crosscutting issues are discussed in the following.

A. **Concurrency**:

Design for concurrency is concerned with decomposing software into processes, tasks, and threads and dealing with related issues of efficiency, atomicity, synchronization, and scheduling.

B. **Control and Handling of Events**:

This design issue is concerned with how to organize data and control flow as well as how to handle reactive and temporal events through various mechanisms such as implicit invocation and call-backs.

C. Data Persistence:

This design issue is concerned with how to handle long-lived data.

D. Distribution of Components:

This design issue is concerned with how to distribute the software across the hardware (including computer hardware and network hardware), how the components communicate, and how middleware can be used to deal with heterogeneous software.

E. **Error and Exception Handling and Fault Tolerance**:

This design issue is concerned with how to prevent, tolerate, and process errors and deal with exceptional conditions.

F. **Interaction and Presentation**:

This design issue is concerned with how to structure and organize interactions with users as well as the presentation of information (for example, separation of presentation and business logic using the Model-View-Controller approach). Note that this topic does not specify user interface details, which is the task of user interface design.

G. Security:

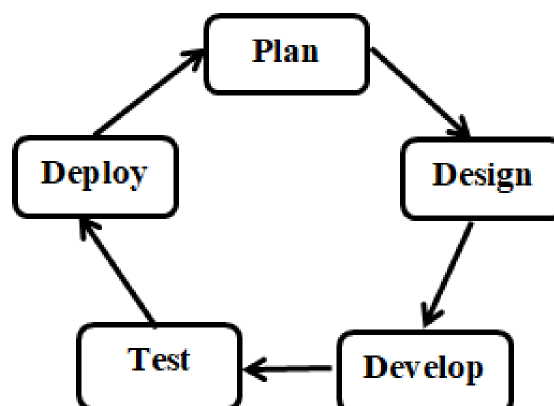
Design for security is concerned with how to prevent unauthorized disclosure, creation, change, deletion, or denial of access to information and other resources. It is also concerned with how to

tolerate security-related attacks or violations by limiting damage, continuing service, speeding repair and recovery, and failing and recovering securely. Access control is a fundamental concept of security, and one should also ensure the proper use of cryptology.

3. **Context of design within multiple software development life cycles:**

Software design is an important part of the software development process. To understand the role of software design, we must see how it fits in the software development life cycle. Thus, it is important to understand the major characteristics of software requirements analysis, software design, software construction, software testing, and software maintenance.

Software Development Life Cycle consists of details steps and activities which describes how to design, develop, maintain, replace, alter, enhance, test or even launch a software. The activities can be broken down into a very detail level but at the same time they can be grouped into five (5) core categories: Plan, Design, Develop, Test and Deploy. Below is a graphic representation which displays a typical Software Development Life Cycle.



4. **Design principles (information hiding, cohesion, and coupling):**

A principle is "a comprehensive and fundamental law, doctrine, or assumption". Software design principles are key notions that provide the basis for many different software design approaches and concepts. Software design principles include abstraction; coupling and cohesion; decomposition and modularization; encapsulation/information hiding; separation of interface and implementation; sufficiency, completeness, and primitiveness; and separation of concerns.

- **Abstraction** is "a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information" . In the context of software design, two key abstraction mechanisms are parameterization and specification. Abstraction by parameterization abstracts from the details of data representations by representing the data as named parameters. Abstraction by specification leads to three major kinds of abstraction: procedural abstraction, data abstraction, and control (iteration) abstraction.
- **Coupling and Cohesion**. Coupling is defined as “a measure of the interdependence among modules in a computer program,” whereas cohesion is defined as “a measure of the strength of association of the elements within a module”].
- **Decomposition and modularization**. Decomposing and modularizing means that large software is divided into a number of smaller named components having well-defined interfaces that describe component interactions. Usually the goal is to place different functionalities and responsibilities in different components.
- **Encapsulation and information hiding** means grouping and packaging the internal details of an abstraction and making those details inaccessible to external entities.

5. **The Design of User Interactions between design and requirements :**

   User interaction involves issuing commands and providing associated data to the software. User interaction styles can be classified into the following primary styles:

   - **Question-answer.** The interaction is essentially restricted to a single question-answer exchange between the user and the software. The user issues a question to the software, and the software returns the answer to the question.
   - **Direct manipulation**. Users interact with objects on the computer screen. Direct manipulation often includes a pointing device (such as a mouse, trackball, or a finger on touch screens) that manipulates an object and invokes actions that specify what is to be done with that object.
   - **Menu selection**. The user selects a command from a menu list of commands.
   - **Form fill-in**. The user fills in the fields of a form. Sometimes fields include menus, in which case the form has action buttons for the user to initiate action.
   - **Command language.** The user issues a command and provides related parameters to direct the software what to do.
   - **Natural language.** The user issues a command in natural language. That is, the natural language is a front end to a command language and is parsed and translated into software commands.

6. **Design for quality attributes (e.g., reliability, usability, maintainability, performance, testability, security, and fault tolerance):**

Software quality attributes (Bass, Clements, and Kazman 2003) refer to the nonfunctional requirements of software, which can have a profound effect on the quality of a software product. Many of these attributes can be addressed and evaluated at the time the software architecture is developed. Software quality attributes include:

1. **Maintainability.**
2. **Modifiability.**
3. **Testability.**
4. **Traceability.**
5. **Scalability.**
6. **Reusability.**
7. **Performance.**
8. **Availability.**
9. **Security.**

An introduction to software quality attributes is given in Section 4.6. This section describes each of these attributes and discusses how they are supported by the COMET design method.

Some software quality attributes are also system quality attributes because they need both the hardware and software to achieve high quality. Examples of these quality attributes are performance, availability, and security. Other software quality attributes are purely software in nature because they rely entirely on the quality of the software. Examples of these quality attributes are maintainability, modifiability, testability, and traceability.

7. **Design trade-off:**

- During software development, trade-offs are made on a daily basis by the people participating in the development project.
- Different roles in the project have to handle different tradeoffs. Some examples are that managers distribute work to developers

and while doing so they have to balance the workload between the developers and deciding how many people that should be assigned to a particular task.

- If more people are assigned to a task then the task will be completed faster, but adding more people past a certain point only serves to increase the overhead of the group and in turn increases the time it takes to complete the task.
- Developers in turn make decisions regarding design and implementation details. An example is when software architects try to balance the quality attributes of the system. A balance of functional as well as quality requirements has to be acheived so that the intended users of the system will find it useful.
- This tradeoff method helps the people that perform the tradeoff to structure the process of evaluating the alternatives. But in the end it relies on the people performing the tradeoff to make the final decision.

Design Strategies

Software Design Strategies:

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution. There are multiple variants of software design. Let us study them briefly:

Structured Design:

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately Structured design is mostly based on 'divide and conquer' strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely - **Cohesion** - grouping of all functionally related elements. **Coupling** - communication between different modules. A good structured design has **high** cohesion and **low** coupling arrangements.

1. **<u>Function Oriented Design:</u>**

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing

significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process:

- The whole system is seen as how data flows in the system by means of data flow diagram.
- Data-flow diagram (DFD) depicts how functions change the data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

2. **Object Oriented Design:**

   Object oriented design works around the entities and their characteristics instead

of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves
around the engaged entities.

Let us see the important concepts of Object Oriented Design:

- **Objects** - All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes** - A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object. In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.
- **Encapsulation** - In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance** - OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism** - OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending

upon how the function is invoked, respective portion of the code gets executed.

Design Process:

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them are defined.
- Application framework is defined.

Software Design Approaches:

There are two generic approaches for software designing:

A. Top down Design

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their one set of sub-system and components and creates hierarchical structure in the system. Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved. Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

A. Bottom-up Design:

The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system. Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

3. Data-structure centered design:

- **Structure design:**

  The aim of structured design is to transform the results of the structured analysis)i.e. a DFD representation) into a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart.

  a. Transform analysis.
  b. Transaction analysis.

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD. These are discussed in the subsequent subsections.

- **Structure Chart:**

represent hierarchical structure of modules. It breaks down the entire system into lowest functional modules, describe functions and sub-functions of each module of a system to a greater detail. Structure Chart partitions the system into black boxes (functionality of the system is known to the users but inner details are unknown). Inputs are given to the black boxes and appropriate outputs are generated.

Modules at top level called modules at low level. Components are read from top to bottom and left to right. When a module calls another, it views the called module as black box, passing required parameters and receiving results.

Symbols used in construction of structured chart:

1. **Module:**

   It represents the process or task of the system. It is of three types:

   a. **Control Module:**

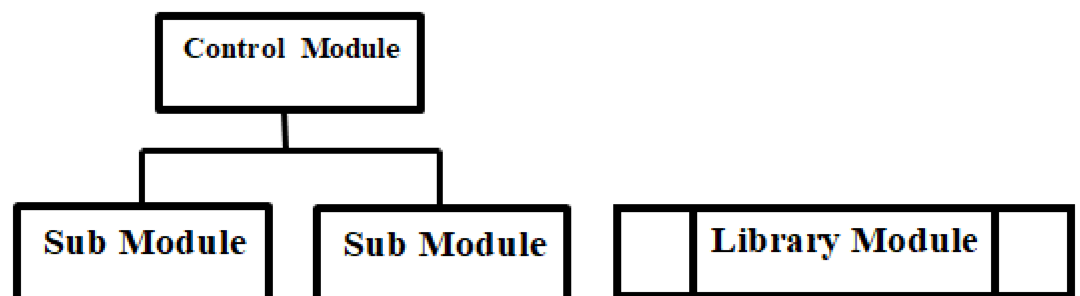
      A control module branches to more than one sub module.

   b. **Sub Module:**

      Sub Module is a module which is the part (Child) of another module.

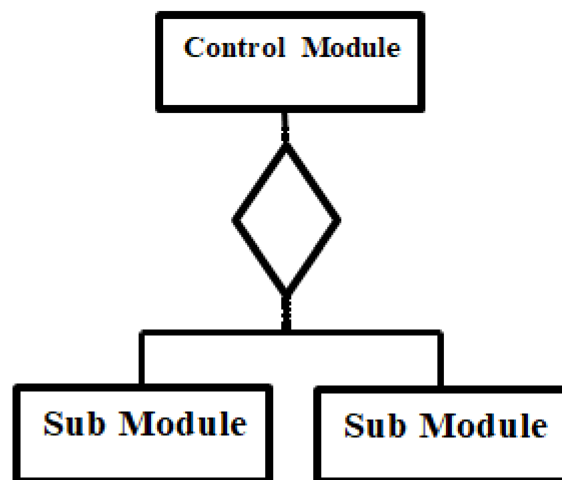
   c. **Library Module:**

      Library Module are reusable and invoable from any module.



2. **Conditional Call:**

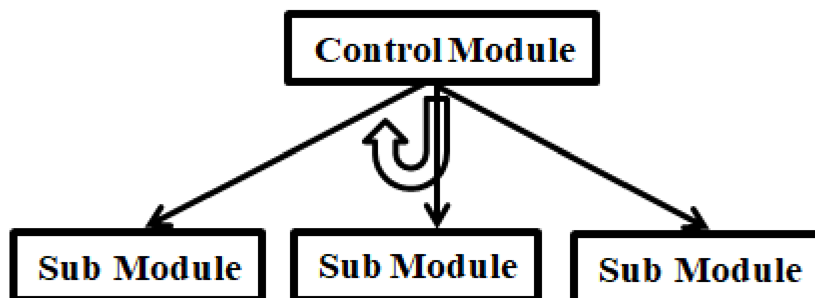
   It represents that control module can select any of the sub module on the basis of some condition.



3. **Loop (Repetitive call of module):**

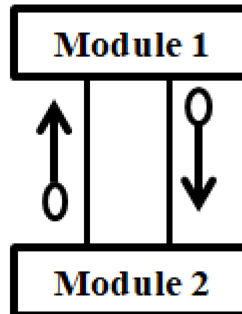
   It represents the repetitive execution of module by the sub module.

   A curved arrow represents loop in the module.



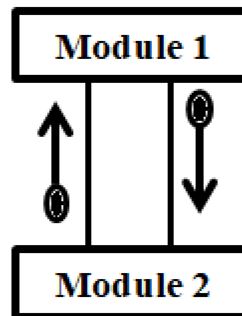
4. **Data Flow:**

It represents the flow of data between the modules. It is represented by directed arrow with empty circle at the end.



5. **Control Flow:**

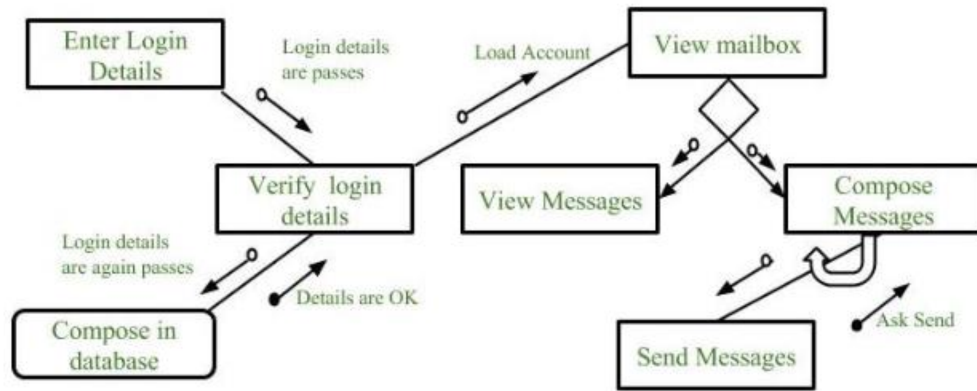
It represents the flow of control between the modules. It is represented by directed arrow with filled circle at the end.



6. **Physical Storage:**

Physical Storage is that where all the information are to be stored.

Example : Structure chart for an Email server:



- **The basic building blocks which are used to design structure charts are the following:**
  a. **Rectangular boxes:** Represents a module.
  b. **Module invocation arrows:** Control is passed from on one module to another module in the direction of the connecting arrow.
  c. **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
  d. **Library modules:** Represented by a rectangle with double edges.
  e. **Selection:** Represented by a diamond symbol.
  f. **Repetition:** Represented by a loop around the control flow arrow.

- **Types of Structure Chart:**

A. Transform Centered Structured:

These type of structure chart are designed for the systems that receives an input which is transformed by a sequence of operations being carried out by one module.

B. Transaction Centered Structure:

These structure describes a system that processes a number of different types of transaction.

4. **Aspect-oriented design:**

The focus of aspect-oriented software development is in the investigation and implementation of new structures for software modularity that provide support for explicit abstractions to modularize concerns. Aspect-oriented programming approaches provide explicit abstractions for the modular implementation of concerns in design, code, documentation, or other artifacts developed during the software life-cycle. These modularized concerns are called aspects, and aspect-oriented approaches provide methods to compose them. . Various approaches provide different flexibility with respect to composition of aspects.

Detailed Design

1. **Design patterns :**

What is a Design Pattern?

- Each pattern Describes a problem which occurs over and over again in our
  environment ,and then describes the core of the problem.
- Novelists, playwrights and other writers rarely invent new stories.

- Often ideas are reused, such as the ―Tragic Hero‖ from Hamlet or Macbeth.
- Designers reuse solutions also, preferably the ―"good" ones – Experience is what makes one an "expert".
- Problems are addressed without rediscovering solutions from scratch.

Design Patterns are the best solutions for the re-occurring problems in the application programming environment.

- Nearly a universal standard.
- Responsible for design pattern analysis in other areas, including GUIs.
- Mainly used in Object Oriented programming.

Design Pattern Elements:

1. Pattern Name:

- Handle (التعامل) used to describe the design problem.
- Increases vocabulary.
- Eases design discussions.
- Evaluation without implementation details.

2. Problem:

- Describes when to apply a pattern.
- May include conditions for the pattern to be applicable.
- Symptoms of an inflexible design or limitation.

3. Solution:

- Describes elements for the design.
- Includes relationships, responsibilities, and collaborations.
- Does not describe concrete designs or implementations.
- A pattern is more of a template.

4. **Consequences:**
    - Results and Trade Offs.
    - Critical for design pattern evaluation.
    - Often space and time trade offs.
    - Language strengths and limitations.

    (Broken into benefits and drawbacks for this discussion).

    - **Design patterns can be subjective.**

      One person‘s pattern may be another person‘s primitive building block. The focus of the selected design patterns are:

    - Object and class communication.
    - Customized to solve a general design problem.
    - Solution is context specific.

Design patterns :

One possible way of designing software is to attempt to match the problem to be solved with a preexisting software system that solves the same type of problem. This approach reduces at least a portion of the software design to pattern matching. The effectiveness depends upon a set of previously developed software modules and some way of recognizing different patterns. Following are some types of patterns that seem to repeatedly occur in software development. Our patterns are relatively high level, from These high-level patterns are:

1. A menu-driven system, where the user must pass through several steps in a hierarchy in order to perform his or her work. The menus may be of the pull-down type such as is common on personal computers or may be entirely text based.

2. An event-driven system, where the user must select steps in order to perform his or
her work. The steps need not be taken in a hierarchical order. This pattern is most
commonly with control of concurrent processes where actions may be repeated
indefinitely.
3. A system in which the actions taken depend on one of a small number of "states" and a small set of optional actions that can be taken for each state. The optional action taken depends on both the state and the value of an input "token." In this pattern, the tokens are usually presented as a stream. Once a token is processed, it is removed from the input stream.
4. A system in which a sequence of input tokens (usually in text format) is processed, one token at a time. This pattern differs from the previous pattern in that the decision about which action to take may depend on more information than is available from just the pair consisting of the state and the input token. In this pattern, the tokens may still remain in the input stream after being processed.

5. A system in which a large amount of information is searched for one or more specific pieces of information. The searches may occur once or many times.

6. A system that can be used in a variety of applications but needs adjustments to work properly in new settings.
7. System in which everything is primarily guided by an algorithm, rather than depending primarily on data.

8. A system that is distributed, with many relatively independent computational actions taking place. Some of the computational actions may communicate with other computational actions.

We note that these high-level patterns can be further broken into three groups: creational patterns, structural patterns, and behavioral patterns. These groups can be broken further into twenty-three patterns as follows:

Creational patterns:

1. Abstract factory—This pattern provides an interface for creating families of related objects without specifying their classes.
2. Builder—This pattern separates the construction of a complex object from its representation.
3. Factory method—This pattern defines an interface for creating a single object and allows subclasses to decide which class to instantiate.
4. Prototype—This pattern specifies the kinds of objects to create using a prototypical instance and creates new objects by copying this prototype. In theory, this can aid in the development of projects that use a rapid prototyping life cycle.
5. Singleton—This pattern makes sure that a class has only a single instance.

Structural patterns:

1. Adapter—This pattern converts the interface of a class into another interface that the clients of the class expect. This function is called bridgeware or glueware in
other contexts.
2. Bridge—This pattern ensures that an abstraction is separate from details of its
implementation.

3. Composite—This pattern allows objects to be grouped into tree structures where
   both individual objects and compositions of objects can be accessed.
4. Decorator—This pattern attaches additional responsibilities to an object dynamically while keeping the same interface.
5. Façade—This pattern provides a unified interface to a set of interfaces in a subsystem.
6. Flyweight—This rare pattern uses a form of sharing to treat large numbers of
   similar objects efficiently.
7. Proxy—This pattern provides a placeholder for another object to control access to it.

Behavioral patterns:

1. Chain of responsibility—This pattern allows giving more than one object a chance to handle a request by the request's sender. This can be highly useful if there are
   alternative responders, including fallback operations.
2. In Command—This rare pattern encapsulates a request as an object, thereby allowing clients to be given different requests.
3. Interpreter—This large-scale pattern allows designers to use a representation of a
   language grammar for parsing. This can be helpful for developing code

metrics.

4. Iterator—This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation, which need not be an array.

5. Mediator—This pattern defines an object that encapsulates how a set of objects
interact, preventing explicit references.
6. Memento—This pattern allows objects to capture and externalize an object's
internal state, while allowing the object to be restored. This can be very useful in
creating rollback states in the event of an unforeseen system error.
7. Observer—This useful pattern defines a one-to-many dependency between objects where a state change in one object results in all dependents being notified and updated.
8. State—This pattern allows an object to alter its behavior when its internal state
changes. This can be useful for rollbacks in the event of system errors.
9. Strategy—This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. This can be used to improve system fault tolerance when incorporated into an "N-version programming" or "multi version" scheme where correctness is evaluated by majority vote on a set of independently implemented
program versions.
10. Template method—This pattern, which is familiar to those who have taken a data
structures course, defines the skeleton of an algorithm in an operation, deferring
some steps to subclasses.
11. Visitor—This pattern is used to represent an operation to be performed on the

elements of an object structure without changing the classes of the elements on
which it operates.

2. **<u>Database Design:</u>**

Database design is the organization of data according to a database model. The designer determines what data must be stored and how the data elements interrelate. With this information, they can begin to fit the data to the database model. Database management system manages the data accordingly.

Database design involves classifying data and identifying interrelationships. This theoretical representation of the data is called an ontology. The ontology is the theory behind the database's design.

1. **Determining data to be stored:**

In a majority of cases, a person who is doing the design of a database is a person with expertise in the area of database design, rather than expertise in the domain from which the data to be stored is drawn e.g. financial information, biological information etc. Therefore, the data to be stored in the database must be determined in cooperation with a person who does have expertise in that domain, and who is aware of what data must be stored within the system.

This process is one which is generally considered part of requirements analysis, and requires skill on the part of the database designer to elicit the needed information from those with the domain knowledge. This is because those with the necessary domain knowledge frequently cannot express clearly what their system requirements for the database are as they are unaccustomed to thinking in terms of the discrete data elements which must be stored. Data to be stored can be determined by Requirement Specification.

2. **Determining data relationships:**

   Once a database designer is aware of the data which is to be stored within the database, they must then determine where dependency is within the data. Sometimes when data is changed you can be changing other data that is not visible. For example, in a list of names and addresses, assuming a situation where multiple people can have the same address, but one person cannot have more than one address, the address is dependent upon the name. When provided a name and the list the address can be uniquely determined; however, the inverse does not hold - when given an address and the list, a name cannot be uniquely determined because multiple people can reside at an address. Because an address is determined by a name, an address is considered dependent on a name.

3. **Determining data relationships:**

   Once a database designer is aware of the data which is to be stored within the database, they must then determine where dependency is within the data. Sometimes when data is changed you can be changing other data that is not visible. For example, in a list of names and addresses, assuming a situation where multiple people can have the same address, but one person cannot have more than one address, the address is dependent upon the name. When provided a name and the list the address can be uniquely determined; however, the inverse does not hold - when given an address and the list, a name cannot be uniquely determined because multiple people can reside at an address. Because an address is determined by a name, an address is considered dependent on a name.

4. **Logically structuring data:**

   Once the relationships and dependencies amongst the various pieces of information have been determined, it is possible to arrange the data into a logical structure which can then be mapped into the storage objects supported by the database management system. In the case of relational

databases the storage objects are tables which store data in rows and columns. In an Object database the storage objects correspond directly to the objects used by the Object-oriented programming language used to write the applications that will manage and access the data. The relationships may be defined as attributes of the object classes involved or as methods that operate on the object classes.

5. **ER diagram (entity-relationship model):**

   Database designs also include ER (entity-relationship model) diagrams. An ER diagram is a diagram that helps to design databases in an efficient way.Attributes in ER diagrams are usually modeled as an oval with the name of the attribute, linked to the entity or relationship that contains the attribute.

6. **A design process suggestion for Microsoft Access:**

   a. Determine the purpose of the database - This helps prepare for the remaining steps.

   b. Find and organize the information required - Gather all of the types of information to record in the database, such as product name and order number.

   c. Divide the information into tables - Divide information items into major entities or subjects, such as Products or Orders. Each subject then becomes a table.

   d. Turn information items into columns - Decide what information needs to be stored in each table. Each item becomes a field, and is displayed as a column in the table. For example, an Employees table might include fields such as Last Name and Hire Date.

e. Specify primary keys - Choose each table's primary key. The primary key is a column, or a set of columns, that is used to uniquely identify each row. An example might be Product ID or Order ID.
f. Set up the table relationships - Look at each table and decide how the data in one table is related to the data in other tables. Add fields to tables or create new tables to clarify the relationships, as necessary.
g. Refine the design - Analyze the design for errors. Create tables and add a few records of sample data. Check if results come from the tables as expected. Make adjustments to the design, as needed.
h. Apply the normalization rules - Apply the data normalization rules to see if tables are structured correctly. Make adjustments to the tables, as needed.

7. Physical design:

The physical design of the database specifies the physical configuration of the database on the storage media. This includes detailed specification of data lements, data types, indexing options and other parameters residing in the DBMS data dictionary. It is the detailed design of a system that includes modules & the database's hardware & software specifications of the system. Some aspects that are addressed at the physical layer:

- Security - end-user, as well as administrative security.
- Replication - what pieces of data get copied over into another database, and how often. Are there multiple-masters, or a single one?
- High-availability - whether the configuration is active-passive, or active-active, the topology, coordination scheme, reliability targets, etc all have to be defined.

- Partitioning - if the database is distributed, then for a single entity, how is the data distributed amongst all the partitions of the database, and how is partition failure taken into account.
- Backup and restore schemes.

At the application level, other aspects of the physical design can include the need to define stored procedures, or materialized query views, OLAP cubes, et…

3. Design of networked systems:

- **Network Design Basics:** the basics of network design are:

1. Network Design Overview:

a. Network Requirements

Today, the Internet-based economy often demands around-the-clock customer service. This means that business networks must be available nearly 100 percent of the time. They must be smart enough to automatically protect against unexpected security incidents. These business networks must also be able to adjust to changing traffic loads to maintain consistent application response times. It is no longer practical to construct networks by connecting many standalone components without careful planning and design.

b. Building a Good Network

Good networks do not happen by accident. They are the result of hard work by network designers and technicians, who identify network requirements and select the best solutions to meet the needs of a business. The steps required to design a good network are as follows:

Step 1. Verify the business goals and technical requirements.

Step 2. Determine the features and functions required to meet the needs identified in Step 1.

Step 3. Perform a network-readiness assessment.

Step 4. Create a solution and site acceptance test plan.

Step 5. Create a project plan.

After the network requirements have been identified, the steps to designing a good network are followed as the project implementation moves forward.

Network users generally do not think in terms of the complexity of the underlying network. They think of the network as a way to access the applications they need, when they need them.

c. Network Requirements:

Most businesses actually have only a few requirements for their network:

1. The network should stay up all the time, even in the event of failed links, equipment failure, and overloaded conditions.
2. The network should reliably deliver applications and provide reasonable response times from any host to any host.
3. The network should be secure. It should protect the data that is transmitted over it and data stored on the devices that connect to it.
4. The network should be easy to modify to adapt to network growth and general business changes.
5. Because failures occasionally occur, troubleshooting should be easy. Finding and fixing a problem should not be too time-consuming.

d. Fundamental Design Goals:

When examined carefully, these requirements translate into four fundamental network design goals:

1. **Scalability:** Scalable network designs can grow to include new user groups and remote sites and can support new applications without impacting the level of service delivered to existing users.
2. **Availability:** A network designed for availability is one that delivers consistent, reliable performance, 24 hours a day, 7 days a week. In addition, the failure of a single link or piece of equipment should not significantly impact network performance.
3. **Security:** Security is a feature that must be designed into the network, not added on after the network is complete. Planning the location of security devices, filters, and firewall features is critical to safeguarding network resources.
4. **Manageability:** No matter how good the initial network design is, the available network staff must be able to manage and support the network. A network that is too complex or difficult to maintain cannot function effectively and efficiently.

2. <u>The Benefits of a Hierarchical Network Design:</u>

To meet the four fundamental design goals, a network must be built on an architecture that allows for both flexibility and growth.

a. Hierarchical Network Design:

In networking, a hierarchical design is used to group devices into multiple networks. The networks are organized in a layered approach. The hierarchical design model has three basic layers:

- Core layer: Connects distribution layer devices
- Distribution layer: Interconnects the smaller local networks
- Access layer: Provides connectivity for network hosts and end devices.

Hierarchical networks have advantages over flat network designs. The benefit of dividing a flat network into smaller, more manageable

hierarchical blocks is that local traffic remains local. Only traffic destined for other networks is moved to a higher layer.

Layer 2 devices in a flat network provide little opportunity to control broadcasts or to filter undesirable traffic. As more devices and applications are added to a flat network, response times degrade until the network becomes unusable. Figures 1 and 2 show the advantages of a hierarchical network design versus a flat network design.

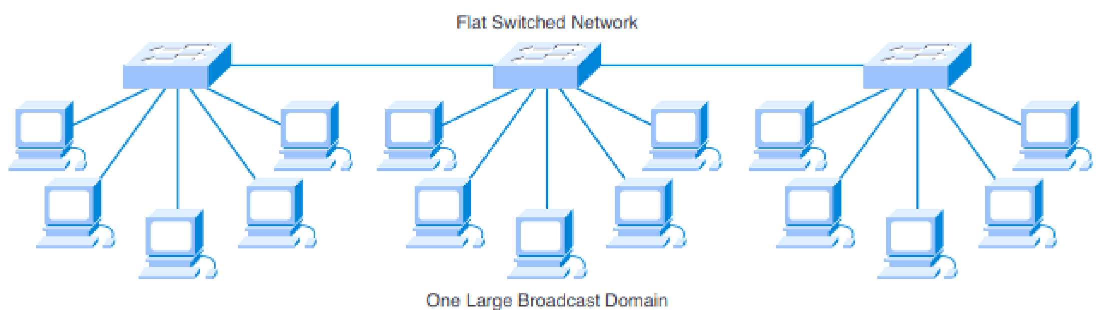


Figure 1 Flat Network

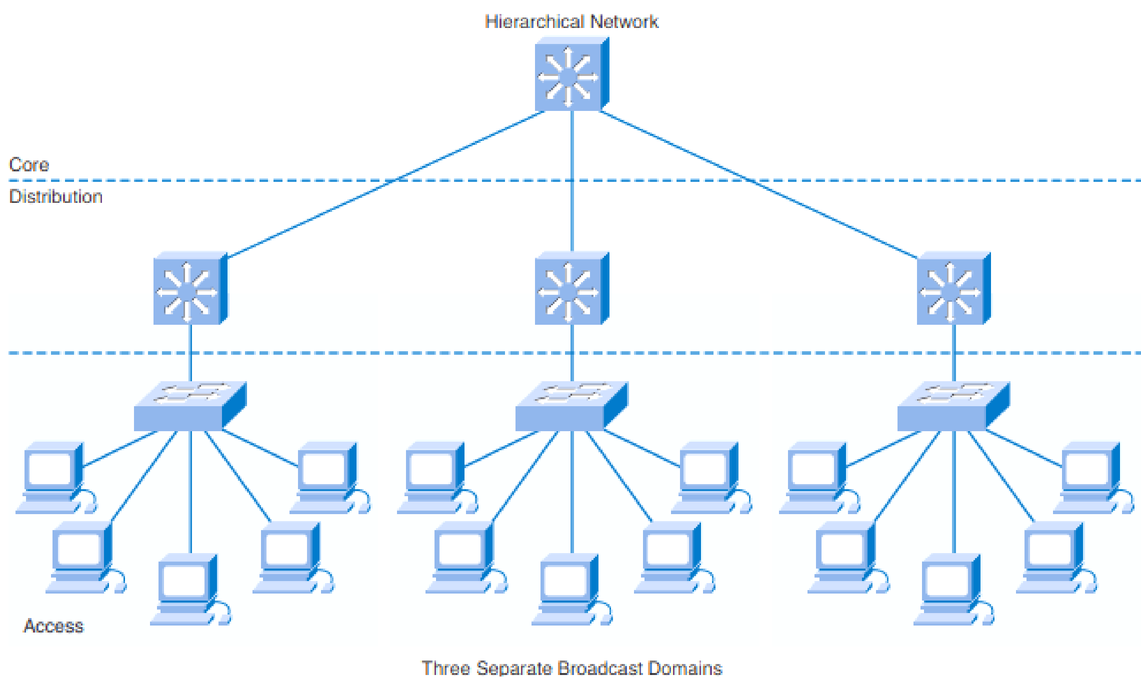


Figure 2 Hierarchical Network

b. Modular Design of Cisco Enterprise Architectures:

The Cisco Enterprise Architectures (see Figure 3) can be used to further divide the three-layer hierarchical design into modular areas. The modules represent areas that have different physical or logical connectivity. They designate where different functions occur in the network. This modularity enables flexibility in network design. It facilitates implementation and troubleshooting. Three areas of focus in modular network design are as follows:

1. Enterprise campus: This area contains the network elements required for independent operation within a single campus or branch location. This is where the building access, building distribution, and campus core are located.
2. Server farm: A component of the enterprise campus, the data center server farm protects the server resources and provides redundant, reliable high-speed connectivity.
3. Enterprise edge: As traffic comes into the campus network, this area filters traffic from the external resources and routes it into the enterprise network. It contains all the elements required for efficient and secure communication between the enterprise campus and remote locations, remote users, and the Internet.

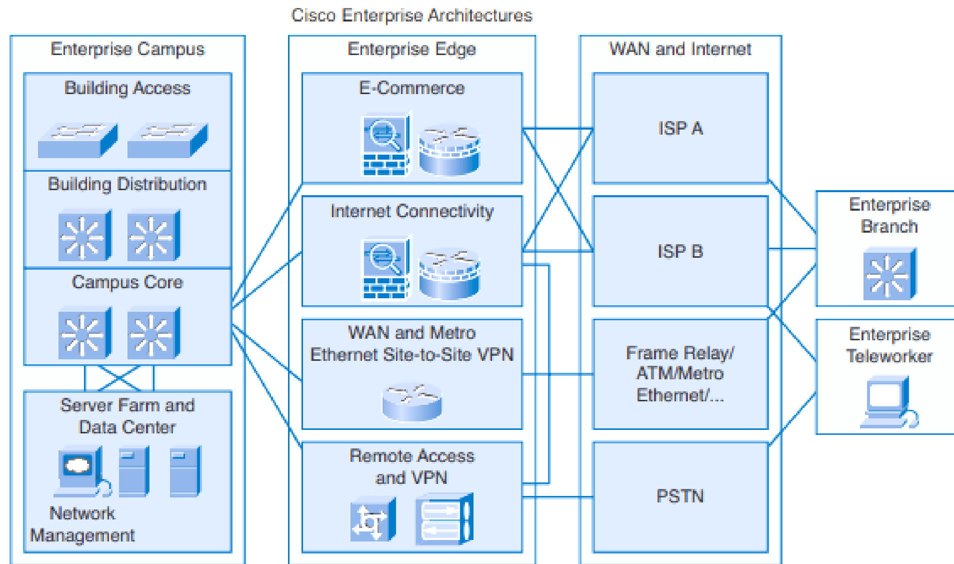


Figure 3 Cisco Enterprise Architectures

The modular framework of the Cisco Enterprise Architectures as depicted in Figure 4 has the following design advantages:

1. It creates a deterministic network with clearly defined boundaries between modules. This provides clear demarcation points so that the network designer knows exactly where the traffic originates and where it flows.
2. It eases the design task by making each module independent. The designer can focus on the needs of each area separately.
3. It provides scalability by allowing enterprises to add modules easily. As network complexity grows, the designer can add new functional modules.
4. It enables the designer to add services and solutions without changing the **underlying network design.**

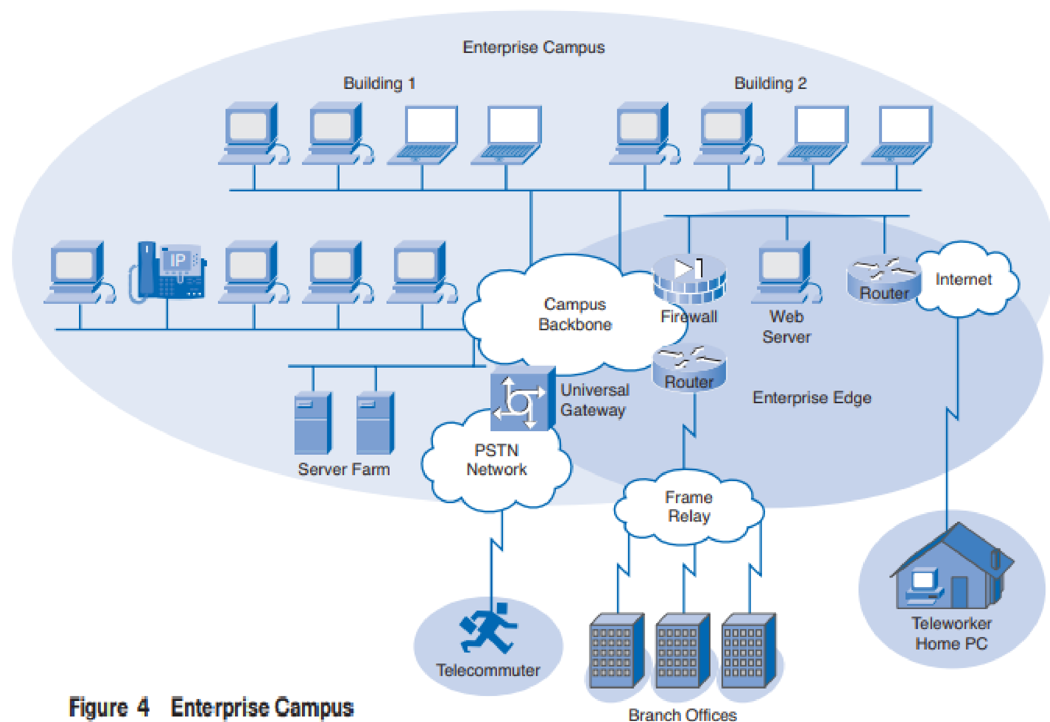


Figure 4 Enterprise Campus

4. Network Design Methodologies:

Large network design projects are normally divided into three distinct steps:

Step 1. Identify the network requirements.

Step 2. Characterize the existing network.

Step 3. Design the network topology and solutions

Step 1: Identifying Network Requirements

The network designer works closely with the customer to document the goals of the project. Figure 5 depicts a meeting between the designer and the business owner. Goals are usually separated into two categories:

a. Business goals: Focus on how the network can make the business more successful.

b. Technical requirements: Focus on how the technology is implemented within the network

Step 2: Characterizing the Existing Network

Information about the current network and services is gathered and analyzed. It is necessary to compare the functionality of the existing network with the defined goals of the new project. The designer determines whether any existing equipment, infrastructure, and protocols can be reused, and what new equipment and protocols are needed to complete the design.

Step 3: Designing the Network Topology

A common strategy for network design is to take a top-down approach. In this approach, the network applications and service requirements are identified, and then the network is designed to support them. When the design is complete, a prototype or proof-of-concept test is performed.

Figure 5 Client Interaction

A common mistake made by network designers is the failure to correctly determine the scope of the network design project.

Determining the Scope of the Project While gathering requirements, the designer identifies the issues that affect the entire network and those that affect only specific portions. By creating a topology similar to Figure 6, the designer can isolate areas of concern and identify the scope of the project. Failure to understand the impact of a particular requirement often causes a project scope to

expand beyond the original estimate. This oversight can greatly increase the cost and time required to implement the new design.

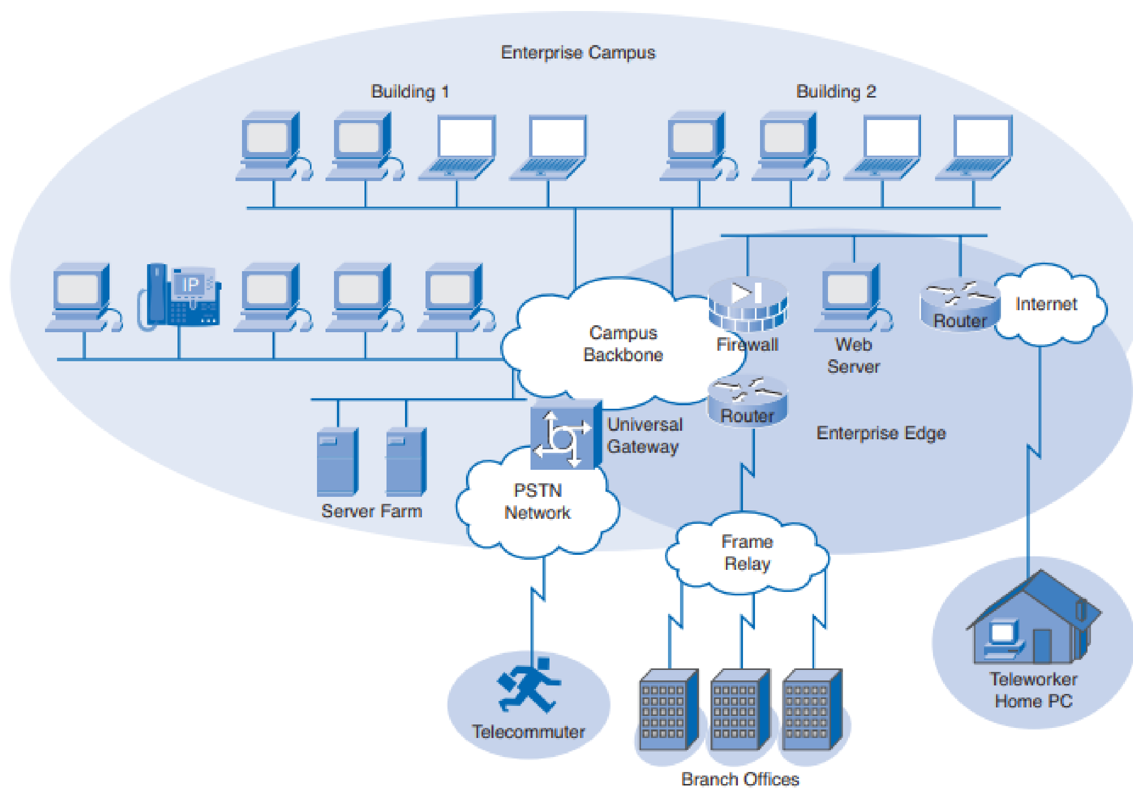


Figure 6 Enterprise Campus

Impacting the Entire Network

Network requirements that impact the entire network include the following:

1. Adding new network applications and making major changes to existing. applications, such as database or Domain Name System (DNS) structure changes.

2. Improving the efficiency of network addressing or routing protocol changes

3. Integrating new security measures.

4. Adding new network services, such as voice traffic, content networking, and storage networking.

5. Relocating servers to a data center server farm.

Impacting a Portion of the Network:

Requirements that may only affect a portion of the network include the following:

1. Improving Internet connectivity and adding bandwidth;
2. Updating access layer LAN cabling.
3. Providing redundancy for key services.
4. Supporting wireless access in defined areas.
5. Upgrading WAN bandwidth.

4. Design notations (e.g., class and object diagrams, UML, state diagrams, and formal specification):

UML is popular for its diagrammatic notations. We all know that UML is for visualizing, specifying, constructing and documenting the components of software and non-software systems. Hence, visualization is the most important part which needs to be understood and remembered.

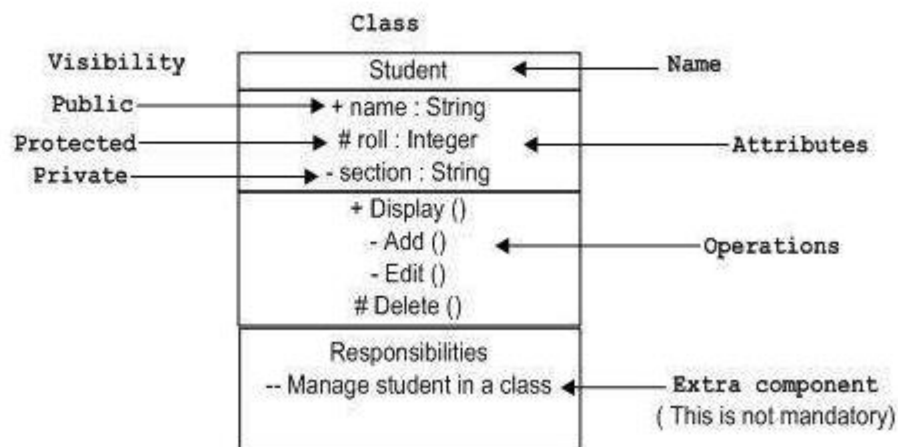
UML notations are the most important elements in modeling. Efficient and appropriate use of notations is very important for making a complete and meaningful model. The model is useless, unless its purpose is depicted properly.

Hence, learning notations should be emphasized from the very beginning. Different notations are available for things and relationships. UML diagrams are made using the notations of things and relationships. Extensibility is another important feature which makes UML more powerful and flexible.

- **Class Notation**

UML class is represented by the following figure. The diagram is divided into four parts.

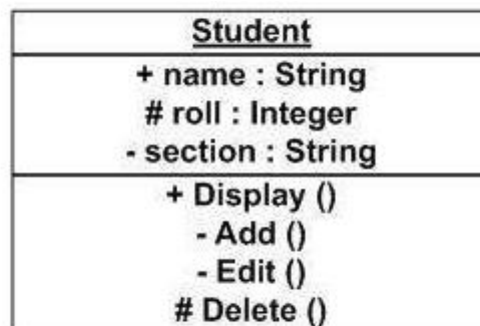
1. The top section is used to name the class.
2. The second one is used to show the attributes of the class.
3. The third section is used to describe the operations performed by the class.
4. The fourth section is optional to show any additional components.



Classes are used to represent objects. Objects can be anything having properties and responsibility.

- **Object Notation:**

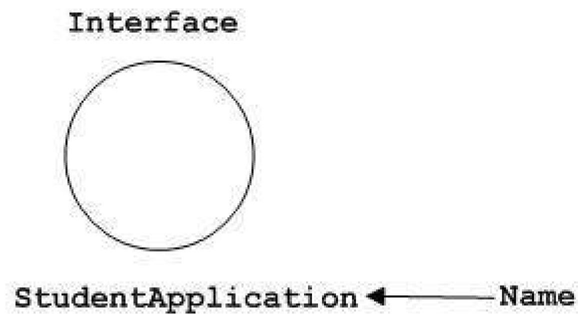
The object is represented in the same way as the class. The only difference is the name which is underlined as shown in the following figure.



As the object is an actual implementation of a class, which is known as the instance of a class. Hence, it has the same usage as the class.

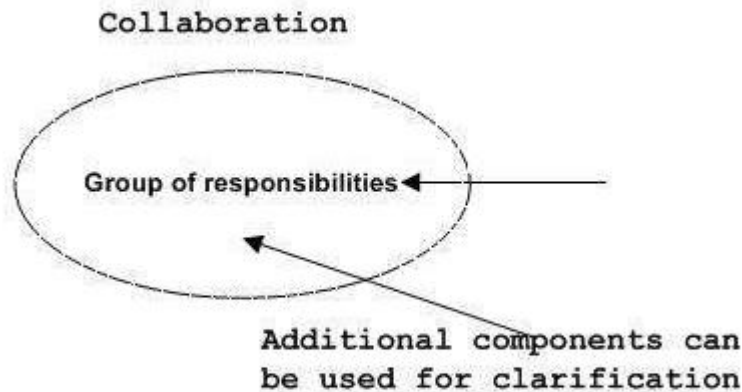
- **Interface Notation:**

Interface is represented by a circle as shown in the following figure. It has a name which is generally written below the circle.



- **Collaboration Notation:**

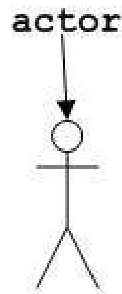
Collaboration is represented by a dotted eclipse as shown in the following figure. It has a name written inside the eclipse:



Collaboration represents responsibilities. Generally, responsibilities are in a group.

- **Actor Notation**:

An actor can be defined as some internal or external entity that interacts with the system.



The usage of Initial State Notation is to show the starting point of a process.

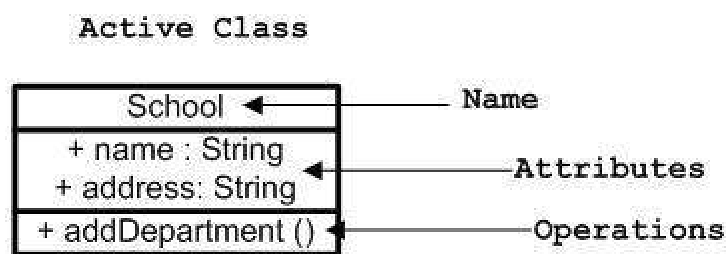
- **Final State Notation:**

Final state is used to show the end of a process. This notation is also used in almost all diagrams to describe the end.

The usage of Final State Notation is to show the termination point of a process

- **Active Class Notation:**

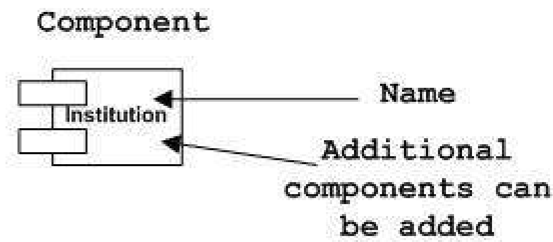
Active class looks similar to a class with a solid border. Active class is generally used to describe the concurrent behavior of a system.



Active class is used to represent the concurrency in a system.

- **Component Notation:**

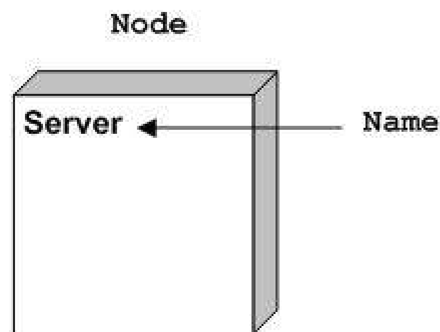
A component in UML is shown in the following figure with a name inside. Additional elements can be added wherever required.



Component is used to represent any part of a system for which UML diagrams are made.

- **Node Notation:**

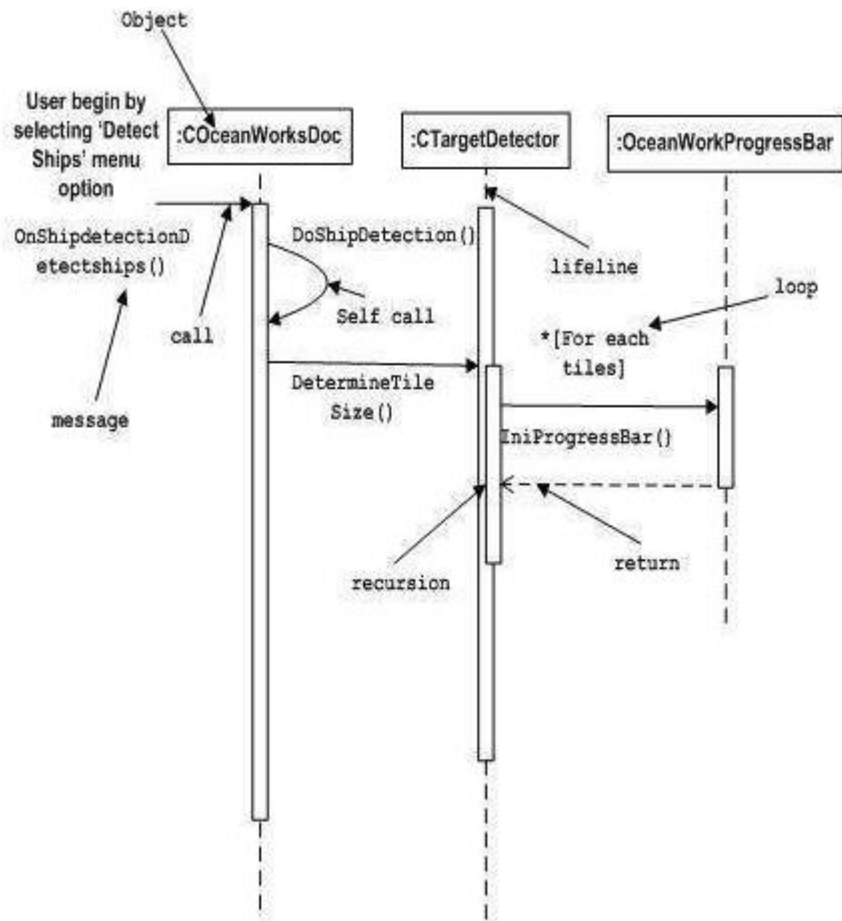
A node in UML is represented by a square box as shown in the following figure with a name. A node represents the physical component of the system.



Node is used to represent the physical part of a system such as the server, network, etc.,

- **Interaction Notation:**

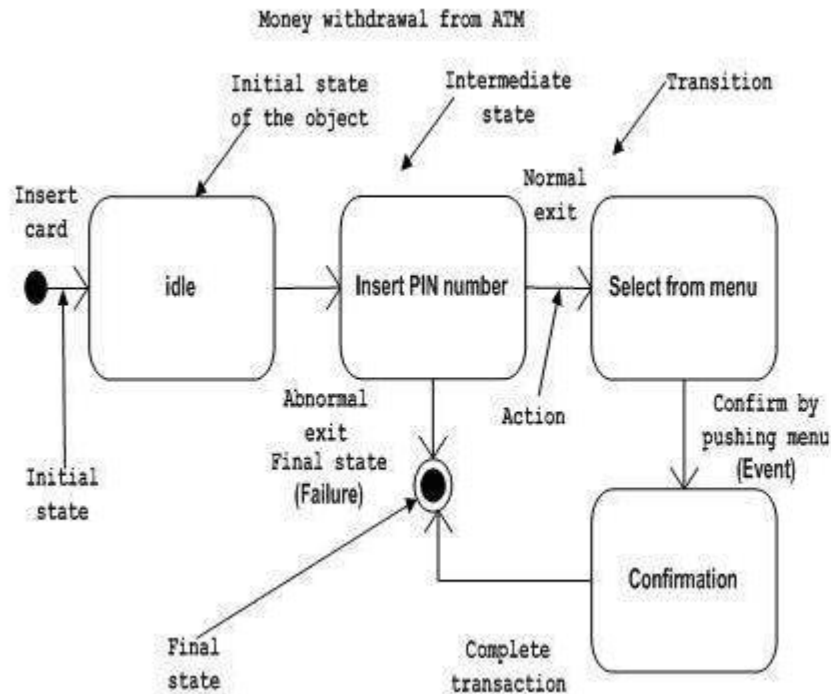
Interaction is basically a message exchange between two UML components. The following diagram represents different notations used in an interaction.



Interaction is used to represent the communication among the components of a system.

- **State Machine Notation:**

State machine describes the different states of a component in its life cycle. The notations are described in the following diagram.



State machine is used to describe different states of a system component. The state can be active, idle, or any other depending upon the situation.

Design Evaluation

1. **Design Attributes (e.g., Coupling, Cohesion, Information Hiding, and Separation of Concerns) :**

   A. Cohesion:

   In computer programming, cohesion refers to the degree to which the elements inside a module belong together. sense, it is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class. In another sense, it is a measure of the strength of relationship between the class's method and data themselves.

   So, cohesion focuses on how single module/class is designed. Higher the cohesiveness of the module/class, better is the OO design.

If our module performs one task and nothing else or has a clear purpose, our module has high cohesion. On the other hand, if our module tries to encapsulate more than one purpose or has an unclear purpose, our module has low cohesion.

Modules with high cohesion tend to be preferable, simple because high cohesion is associated with several desirable traits of software including:

- Robustness.
- Reliability.
- understandability.

Low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

Cohesion is often contrasted with coupling. High cohesion often correlates with loose coupling, and vice versa.

Single Responsibility Principle aims at creating highly cohesive classes.

Cohesion is increased if:

- The functionalities embedded in a class, accessed through its methods, have much in common.
- Methods carry out a small number of related activities, by avoiding coarsely grained or unrelated sets of data.

Advantages of high cohesion:

- Reduced module complexity (they are simpler, having fewer operations).
- Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules.

- Increased module reusability, because application developers will find the component they need more easily among the cohesive set of operations provided by the module.

Types of cohesion: There are some types of cohesion that we need to know:

- **Coincidental cohesion(worst):**
  Coincidental cohesion is when parts of a module are grouped arbitrarily; the only relationship between the parts is that they have been grouped together**.** For example - Utilities class**.**
- **Logical cohesion:**
  Logical cohesion is when parts of a module are grouped because they are logically categorized to do the same thing even though they are different by nature.For example - grouping all mouse and keyboard input handling routines.
- **Temporal cohesion:**
  Temporal cohesion is when parts of a module are grouped by when they are processed - the parts at a particular time in program execution.
  For example - A function which is called after catching an exception which closes open files, creates an error log, and notifies the user.

- **Procedural cohesion:**
  Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution. For example - a function which checks file permissions and then opens the file**.**

- **Communicational / Informal cohesion**

  Communicational cohesion is when parts of a module are grouped because they operate on the same data. There are cases where communicational cohesion is the highest level of cohesion that can be attained under the circumstances. For example - a module which operates on the same record of information.

- **Sequential cohesion:**

  Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line.

  For example: a function which reads data from a file and processes the data.

- **Functional cohesion (best)**

  Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module. While functional cohesion is considered the most desirable type of cohesion for a software module, it may no be achievable.

- **Perfect cohesion (atomic).**

B. Coupling:

is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

Coupling increases between two classes A and B if:

- A has an attribute that refers to (is of type) B.

- A calls on services of an object B.
- A has a method that reference B (via return type or parameter.(
- A is a subclass of (or implements) class B.

Low coupling refers to a relationship in which one module interacts with another module through a simple and stable interface and does not need to be concerned with the other module's internal implementation.

Some properties that need to consider in coupling: In Coupling, we need to consider some properties:

- **Degree:**

Degree is the number of connections between the module and others. With coupling, we want to keep the degree small. For instance, if the module needed to connect to other modules through a few parameters or narrow interfaces, then the degree would be small, and coupling would be loose.

- **Ease:**

Ease is how obvious are the connections between the module and others. With coupling, we want the connections to be easy to make without needing to understand the implementations of the other modules.

- **Flexibility:**

Flexibility is how interchangeable the other modules are for this module. With coupling, we want the other modules easily replaceable for something better in the future.

Disadvantages of tightly coupling:

- A change in one module usually forces a ripple effect of changes in other modules.

- Assembly of modules might require more effort or time due to the increased inter-module dependency.
- A particular module might be harder to reuse or test because dependent modules must be included.

Difference between cohesion and coupling:

☒ **Cohesion:**

- Cohesion is the indication of the relationship within module.
- Cohesion shows the module's relative functional strength;
- Cohesion is a degree (quality) to which a component / module focuses on the single thing.
- While designing we should strive for high cohesion. Ex: cohesive component/module focus on a single task with little interaction with other modules of the system.
- Cohesion is the kind of natural extension of data hiding, for example, class having all members visible with a package having default visibility.
- Cohesion is Intra – Module Concept.

☒ **Coupling:**

- Coupling is the indication of the relationships between modules.
- Coupling shows the relative independence among the modules.
- Coupling is a degree to which a component / module is connected to the other modules.
- While designing we should strive for low coupling. Ex: dependency between modules should be less.
- Making private fields, private methods and non public classes provides loose coupling.
- Coupling is Inter -Module Concept.

C. Information Hiding:

- Only the operations of a class are allowed to manipulate its attributes.
    - Access attributes only via operations.
- Hide external objects at subsystem boundary.
    - Define abstract class interfaces which mediate between the external world and the system as well as between subsystems.
- Do not apply an operation to the result of another operation.
    - Write a new operation that combines the two operations.

D. Separation of concerns:

Separation of concerns is the idea that each module or layer in an application should only be responsible for one thing and should not contain code that deals with other things. Separating concerns reduces code complexity by breaking a large application down into many smaller units of encapsulated functionality.

Separation of concerns can be expressed as functions, modules, controls, widgets, layers, tiers, services, and so on. The various units of concern vary from one app to the next, and each different app may use a different combination. Functions and modules have already been discussed.

A control is a reusable GUI input that enables user interaction with your application. For example, combo boxes, calendar inputs, sliders, buttons, switches, and knobs are all controls.

A widget is a small application which is intended to be embedded in other applications. For example, WordPress allows developers to offer embeddable units of functionality to blog owners through its plug-in ecosystem. There are many widgets to manage calendars, comments, maps, and all sorts of services from third-party providers.

Layers are logical groupings of functionality. For example, a data layer might encapsulate functionality related to data and state, while a presentation layer handles display concerns, such as rendering to the DOM and binding UI behaviors.

Tiers are the runtime environments that layers get deployed to. A runtime environment usually consists of at least one physical computer, an operating system, the runtime engine (e.g., Node, Java, or Ruby), and any configuration needed to express how the application should interact with its environment.

It's possible to run multiple layers on the same tier, but tiers should be kept independent enough that they can easily be deployed to separate machines or even separate data centers. For large-scale applications, it's usually also necessary that tiers can scale horizontally, meaning that as demand increases, you can add machines to a tier in order to improve its capacity to meet that demand.

- **Client-Side Concerns:**

There are several client-side concerns that almost every mature JavaScript application might deal with at some point:

- Module management.
- Events.
- Presentation and DOM manipulation.
- Internationalization.
- Data management and IO (including Ajax(
- Routing (translating URLs to script actions(
- Logging.
- Analytics tracking.

- Authentication.
- Feature toggling (decouple code deployment and feature release).

2. **<u>design metrics:</u>**

Design metrics fall into two categories**:**

- **Product metrics:**

Derived from design representations, these can be used to predict the extent of a future activity in a software project, as well as assessing the quality of the design in its own right. Product metrics are to be further divided into network, stability and information flow metrics.

- **Process metrics::**

Metrics derived from the various activities that make up the design phase. They include effort, timescale metrics, fault and change metrics. These are normally used for error detection, the time spent at each phase of development, measuring the cost etc. When they are recorded on a unit basis, they can also be used for unit quality control.

Of the two types, product metrics are the most suitable for evaluating software design quality, and so these are discussed further.

- **Network metrics:**

These metrics sometimes referred to as call graph metrics, are based on the shape of the calling hierarchy within the software system. Their complexity metric as based on measuring how far a design deviates from a tree structure with neither common calls to modules nor common access to a database. The theory on which this metric was based is that both common calls and common database access increase the coupling between the modules.

- **Stability metrics:**

Stability metrics are based on the resistance to change that occurs

in a software system during maintenance. The principle behind this type of metric is that a poor system is one where a change to one module has a high probability of giving rise to changes in other modules. This, in turn, has a high probability of giving rise to further changes in other modules. The work is an expansion of a metric, which relies on the subjective estimation of the effect that a change to one module had on another.

This early work has now been refined . Design stability measures can now
be obtained at any point in the design process, allowing examination of the
program early in its life-cycle for possible maintenance problems. Design
stability measurement requires a more in-depth analysis of the interfaces of
modules and an account of the 'ripple effect' as a consequence of program
modifications (stability of the program). The potential 'ripple effect' is defined as the total number of assumptions made by other modules, which invoke a module whose stability is being measured, share global data or files with modules, or are invoked by the module.

During program maintenance, if changes are made that affect these assumptions, a 'ripple effect' may occur through the program, requiring additional costly changes. It is possible to calculate the 'ripple effect' consequent on modifying the module.

The design stability of a piece of software will be calculated on the basis

of the total potential ‘ripple effect’ of all its modules. This approach allows the calculation of design stability measures at any point in the design process. Areas of the program with poor stability can then be redesigned to improve the situation.

- **Evolution processes:**

  Software evolution processes vary depending on the type of software being
  maintained, the development processes used in an organization and the skills of the people involved. In some organizations, evolution may be an informal process where change requests mostly come from conversations between the system users and developers. In other companies, it is a formalized process with structured documentation produced at each stage in the process.

  System change proposals are the driver for system evolution in all organizations. Change proposals may come from existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team. The processes of change identification and system evolution are cyclic and continue throughout the lifetime of a system.
  Change proposals should be linked to the components of the system that have to be modified to implement these proposals. This allows the cost and the impact of the change to be assessed. This is part of the general process of change management, which also should ensure that the correct versions of components are included in each system release.

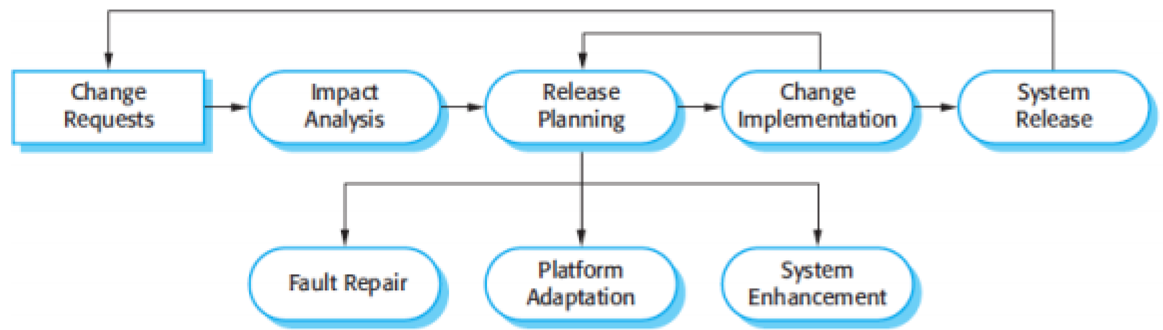


Figure : The software evolution process

The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. The cost

and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change. If

the proposed changes are accepted, a new release of the system is planned.

During release planning, all proposed changes (fault repair, adaptation, and new

functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and

validated, and a new version of the system is released. The process then iterates

with a new set of changes proposed for the next release. You can think of change implementation as an iteration of the development process, where the revisions to the system are designed, implemented, and tested.

However, a critical difference is that the first stage of change implementation may involve program understanding, especially if the

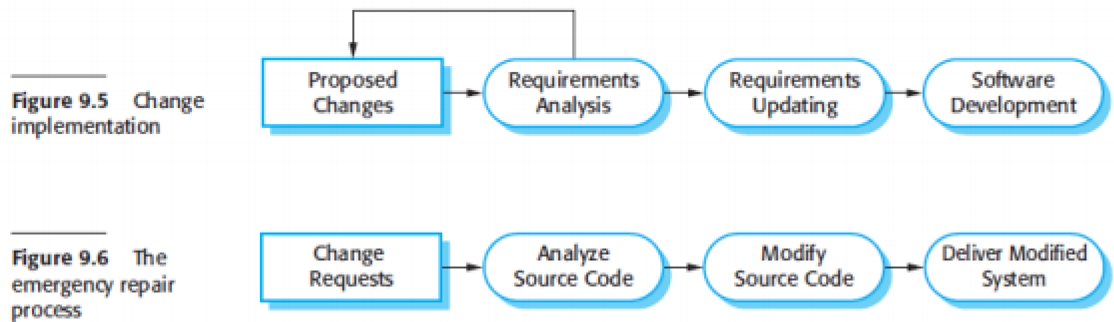
original system developers are not responsible for change implementation. During this program understanding phase, you have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. You need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system. Ideally, the change implementation stage of this process should modify the system specification, design, and implementation to reflect the changes to the system.

New requirements that reflect the system changes are proposed, analyzed, and validated. System components are redesigned and implemented and the system is retested. If appropriate, prototyping of the proposed changes may be carried out as part of the change analysis process.

During the evolution process, the requirements are analyzed in detail and implications of the changes emerge that were not apparent in the earlier change analysis process. This means that the proposed changes may be modified and further customer discussions may be required before they are implemented.

Change requests sometimes relate to system problems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault occurs that has to be repaired to allow normal peration to continue.



2. If changes to the systems operating environment have unexpected effects that
   disrupt normal operation
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system. In these cases, the need to make the change quickly means that you may not be able to follow the formal change analysis process.

   Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem . However, the danger is that the requirement, the software design, and the code become inconsistent. Although you may intend to document the change in the requirements and design, additional emergency fixes to the software may then be needed. These take priority over documentation. Eventually, the original change is forgotten and the system documentation and code are never realigned.

- **Program evolution dynamics:**

Program evolution dynamics is the study of system change. In the 1970s and 1980s, Lehman and Belady (1985) carried out several empirical studies of system
change with a view to understanding more about characteristics of software

evolution.

Lehman and Belady claim these laws are likely to be true for all types of large organizational software systems (what they call E-type systems). These are systems in which the requirements are changing to reflect changing business needs. New releases of the system are essential for the system to provide business
value.

The first law states that system maintenance is an inevitable process. As the system's environment changes, new requirements emerge and the system must be modified. When the modified system is reintroduced to the environment, this promotes more environmental changes, so the evolution process starts again.

The second law states that, as a system is changed, its structure is degraded. The
only way to avoid this happening is to invest in preventative maintenance. You
spend time improving the software structure without adding to its functionality.
Obviously, this means additional costs, over and above those of implementing
required system changes.

Law	Description
Continuing change	A program that is used in a real-world environment must necessarily change, or else become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release.
Organizational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will decline unless they are modified to reflect changes in their operational environment.
Feedback system	Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

The third law is, perhaps, the most interesting and the most contentious of Lehman's laws. It suggests that large systems have a dynamic of their own that is established at an early stage in the development process.

This determines the gross trends of the system maintenance process and limits the number of possible system changes. Lehman and Belady suggest that this law is a consequence of structural factors that influence and constrain system change, and organizational factors that affect the evolution process.

The structural factors that affect the third law come from the complexity of large systems. As you change and extend a program, its structure tends to degrade. This

is true of all types of system (not just software) and it occurs because you are adapting a structure intended for one purpose for a different purpose.

This degradation, if unchecked, makes it more and more difficult to make further
changes to the program.
Making small changes reduces the extent of structural degradation and so lessens the risks of causing serious system dependability problems. If you try and make large changes, there is a high probability that these will introduce new faults.
This law confirms that large software development teams are often unproductive
because communication overheads dominate the work of the team.Lehman's fifth law is concerned with the change increments in each system release. Adding new functionality to a system inevitably introduces new system faults. The more functionality added in each release, the more faults there will be.
Therefore, a large increment in functionality in one system release means that this
will have to be followed by a further release in which the new system faults are repaired. Relatively little new functionality should be included in this release. This law suggests that you should not budget for large functionality increments
in each release without taking into account the need for fault repair. The first five laws were in Lehman's initial proposals; the remaining laws were added after further work. The sixth and seventh laws are similar and essentially say that users of software will become increasingly unhappy with it unless it is maintained and new functionality is added to it. The final law reflects the most recent work on feedback processes, although it is not yet clear how this can be applied in ractical software development.

- **Software maintenance:**

Software maintenance is the general process of changing a system after it has been delivered. The term is usually applied to custom software in which separate development groups are involved before and after delivery. The changes made to
the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or accommodate new requirements. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system. There are three different types of software maintenance:

a. **Fault repairs** Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.
b. **Environmental adaptation** This type of maintenance is required when some
aspect of the system's environment such as the hardware, the platform operating
system, or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.
c. **Functionality addition** This type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

In practice, there is not a clear-cut distinction between these types of maintenance. When you adapt the system to a new environment, you may add functionality to take advantage of new environmental features. Software faults are often exposed because users use the system in

unanticipated ways. Changing the system to accommodate their way of working is the best way to fix these faults.

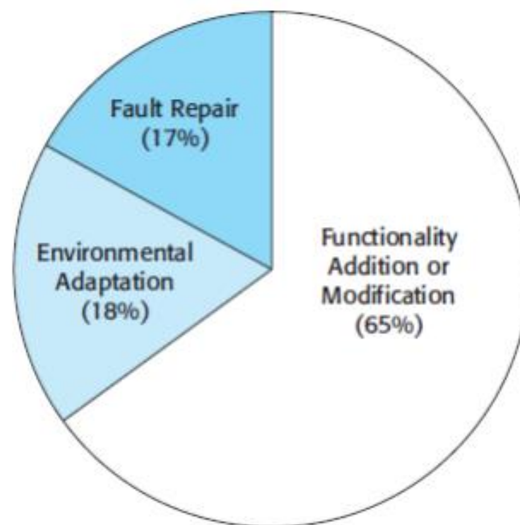


Figure : software Maintenance effort distribution

3. **Formal design analysis:**

- **Formal methods:**

  Formal methods of software design, means using mathematics to write error-free programs. The mathematics needed is not complicated; it's just basic logic. The word "formal" means the use of a formal language, so that the program logic can be machine checked. Our compilers already tell us if we make a syntax error, or a type error, and they tell us what and where the error is. Formal methods take the next step, telling us if we make a logic error, and they tell us what and where the error is. And they tell us this as we make the error, not after the program is finished. It is good to get any program correct while writing it, rather than waiting for bug reports from users. It is absolutely essential for programs that lives will depend on.

In computer science, specifically software engineering and hardware engineering, formal methods are a particular kind of mathematically rigorous techniques for the specification, development and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design.

Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, discrete event dynamic system and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification.