MODULE -II

REQUIREMENT ANALYSIS AND DESIGN

Software requirement analysis and its need

- The goal of the requirements activity is to produce the Software Requirements Specification (SRS) that describes what the proposed software should do without describing how the software will do it.
- A basic purpose of the SRS is to bridge this communication gap so they have a shared vision of the software being built.
- Through SRS, the client clearly describes what it expects from the supplier, and the developer clearly understands what capabilities to build in the software.
- An SRS provides a reference for validation of the final product
- A high-quality SRS reduces the development cost

Requirement Process

The requirements process typically consists of three basic tasks: -

- Problem or Requirement analysis
- Requirements specification and
- Requirements validation

Problem analysis

In Problem analysis an effort is to

- understand the system behavior
- constraints on the system,
- its inputs and outputs, etc.

Problem analysis -steps

- Frequently, during analysis, the analyst will have a series of meetings with the clients and end users.
- In the early meetings, the clients and end users will explain to the analyst about their work, their environment, and their needs as they perceive them. Any documents describing the work or the organization may be given, along with outputs of the existing methods of performing the tasks. In these early meetings, the analyst is basically the listener, absorbing the information provided.

Problem analysis -steps

- Once the analyst understands the system to some extent, he uses the next few meetings to seek clarifications of the parts he does not understand. He may document the information or build some models, and he may do some brainstorming or thinking about what the system should do.
- In the final few meetings, the analyst essentially explains to the client what he understands the system should do and uses the meetings as a means of verifying if what he proposes the system should do is indeed consistent with the objectives of the clients.

Requirements specification

- The understanding obtained by problem analysis forms the basis of requirements specification, in which the focus is on clearly specifying the requirements in a document. As analysis produces large amounts of information and knowledge with possible redundancies, properly organizing and describing the requirements is an important goal of this activity.

Requirements validation

- Requirements validation focuses on ensuring that what have been specified in the SRS are indeed all the requirements of the software and making sure that the SRS is of good quality. The requirements process terminates with the production of the validated SRS.

Desirable Characteristics of an SRS

1. Correct
2. Complete
3. Unambiguous
4. Verifiable
5. Consistent
6. Ranked for importance and/or stability

Characteristics of an SRS

- An SRS is ***correct*** if every requirement included in the SRS represents something required in the final system.
- It is ***complete*** if everything the software is supposed to do and the responses of the software to all classes of input data are specified in the SRS.
- It is ***unambiguous*** if and only if every requirement stated has one and only one interpretation.

Characteristics of an SRS

- An SRS is ***verifiable*** if and only if every stated requirement is verifiable. A requirement is verifiable if there exists some cost-effective process that can check whether the final software meets that requirement.
- It is ***consistent*** if there is no requirement that conflicts with another.
- An SRS is ***ranked for importance and/or stability*** if for each requirement the importance and the stability of the requirement are indicated.
    - Stability of a requirement reflects the chances of it changing in the future. It can be reflected in terms of the expected change volume.

Structure of a Requirements Document

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms, and Abbreviations
   - 1.4 References
   - 1.5 Overview
2. Overall Description
   - 2.1 Product Perspective
   - 2.2 Product Functions
   - 2.3 User Characteristics
   - 2.4 General Constraints
   - 2.5 Assumptions and Dependencies
3. Specific Requirements

Structure of a Requirements Document (cont..)

- 3. Detailed Requirements
  - 3.1 External Interface Requirements
    - 3.1.1 User Interfaces
    - 3.1.2 Hardware Interfaces
    - 3.1.3 Software Interfaces
    - 3.1.4 Communication Interfaces
  - 3.2. Functional Requirements
    - 3.2.1 Mode 1
      - 3.2.1.1 Functional Requirement 1.1
      - :
      - 3.2.1.n Functional Requirement 1.n
      - :
      - 3.2.m Mode m
        - 3.2.m.1 Functional Requirement m.1
        - :
        - 3.2.m.n Functional Requirement m.n

Structure of a Requirements Document (cont..)

- 3.3 Performance Requirements
- 3.4 Design Constraints
- 3.5 Attributes
- 3.6 Other Requirements

Structure of a Requirements Document (cont..)

- The introduction section contains the purpose, scope, overview, etc., of the requirements document.
- The overall Description section gives an overall perspective of the system—how it fits into the larger system, and an overview of all the requirements of this system.
  - Product perspective is essentially the relationship of the product to other products; defining if the product is independent or is a part of a larger product, and what the principal interfaces of the product are. A general abstract description of the functions to be performed by the product is given. Schematic diagrams showing a general view of different functions and their relationships with each other can often be useful. Similarly, typical characteristics of the eventual end user and general constraints are also specified.

Structure of a Requirements Document (cont..)

- The detailed requirements section describes the details of the requirements that a developer needs to know for designing and developing the system. This is typically the largest and most important part of the document.
  - One method to organize the specific requirements is to first specify the external interfaces, followed by functional requirements, performance requirements, design constraints, and system attributes.

Structure of a Requirements Document (cont..)

- The external interface requirements section specifies all the interfaces of the software: to people, other software, hardware, and other systems.
  - User interfaces are clearly a very important component; they specify each human interface the system plans to have, including screen formats, contents of menus, and command structure.
  - In hardware interfaces, the logical characteristics of each interface between the software and hardware on which the software can run are specified. Essentially, any assumptions the software is making about the hardware are listed here.
  - In software interfaces, all other software that is needed for this software to run is specified, along with the interfaces.
  - Communication interfaces need to be specified if the software communicates with other entities in other machines.

Structure of a Requirements Document (cont..)

- In the functional requirements section, the functional capabilities of the system are described. In this organization, the functional capabilities for all the modes of operation of the software are given. For each functional requirement, the required inputs, desired outputs, and processing requirements will have to be specified.
  - For the inputs, the source of the inputs, the units of measure, valid ranges, accuracies, etc., have to be specified.
  - For specifying the processing, all operations that need to be performed on the input data and any intermediate data produced should be specified. This includes validity checks on inputs, sequence of operations, responses to abnormal situations, and methods that must be used in processing to transform the inputs into corresponding outputs.

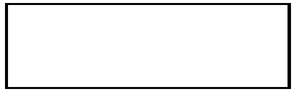
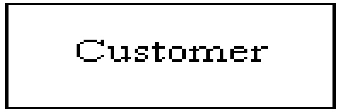
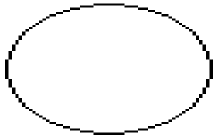
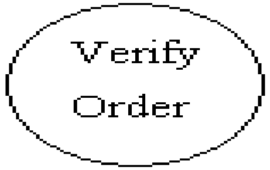
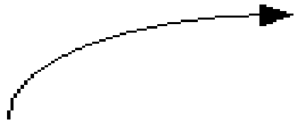
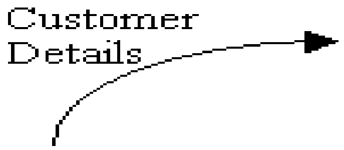
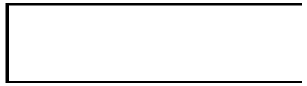
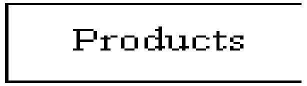
Structure of a Requirements Document (cont..)

- The performance section should specify both static and dynamic performance requirements.
- All factors that constrain the system design are described in the performance constraints section.
- The attributes section specifies some of the overall attributes that the system should have.
- Any requirement not covered under these is listed under other requirements.
- Design constraints specify all the constraints imposed on design (e.g., security, fault tolerance, and standards compliance).

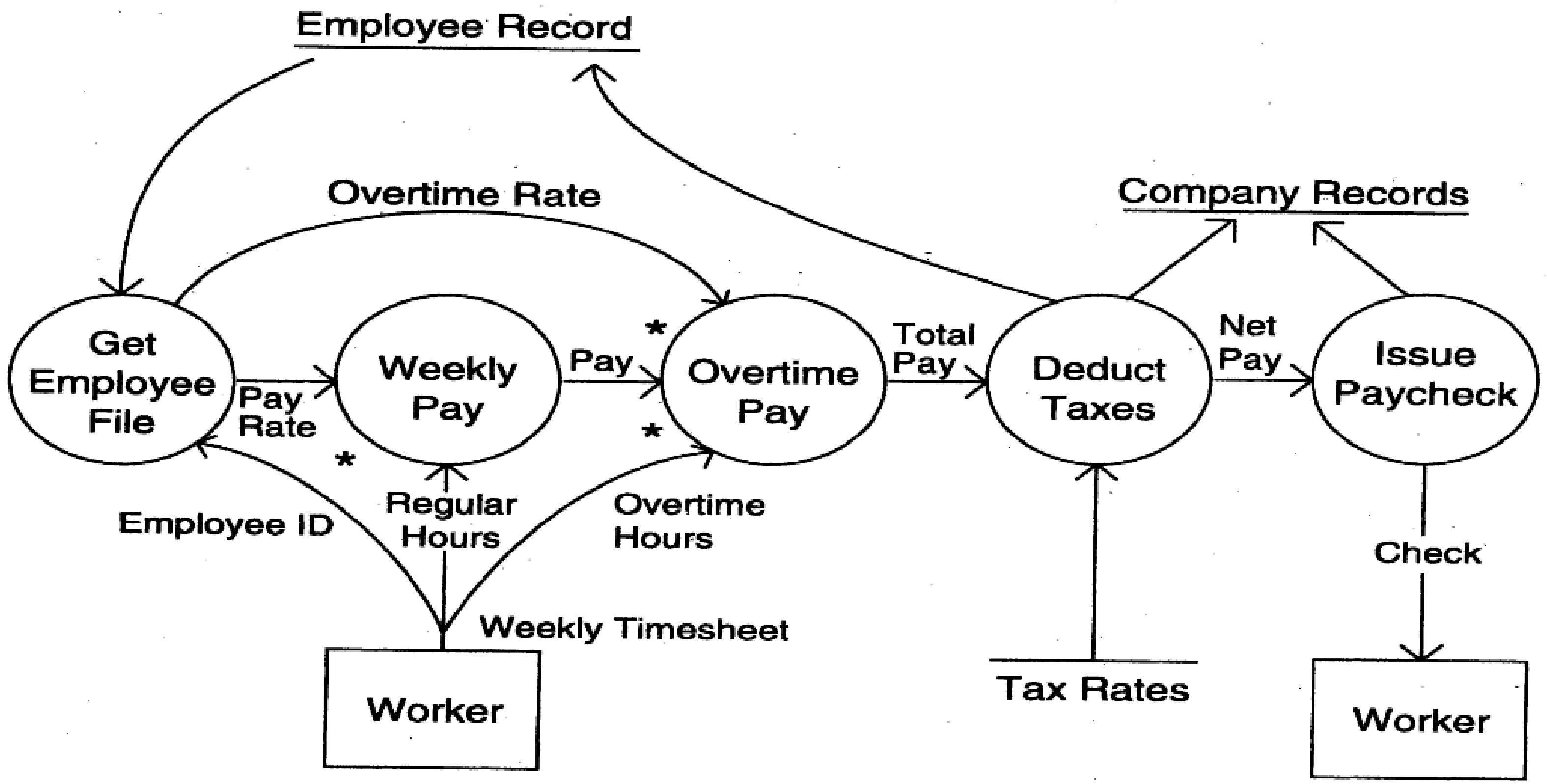
DATA FLOW DIAGRAM (DFD)

- A DFD shows the flow of data through a system. It views a system as a function that transforms the inputs into desired outputs
- The DFD aims to capture the transformations that take place within a system to the input data so that eventually the output data is produced.
- The agent that performs the transformation of data from one state to another is called a process

Symbols used in DFD

Name	Symbol	Description	Example
Entity		Used to represent people and organizations outside the system. They either input information to the system, accept output information from the system or both	
Process		These are actions that are carried out with the data that flows around the system. A process accepts input data and produces data that it passes on to another part of the DFD	
Data Flow		These represent the flow of data to or from a process	
Data Store		This is a place where data is stored either temporarily or permanently	

Example of DFD



Example DFD

- All external files such as employee record, company record, and tax rates are shown as a labeled straight line.
- The need for multiple data flows by a process is represented by a "*" between the data flows. This symbol represents the AND relationship.
    - For example, if there is a "*" between the two input data flows A and B for a process, it means that A AND B are needed for the process. In the DFD, for the process "weekly pay" the data flow "hours" and "pay rate" both are needed, as shown in the DFD.
- The OR relationship is represented by a "+" between the data flows.

DFD

- A DFD is not a flowchart.
- A DFD represents the flow of data, while a flowchart shows the flow of control.
- A DFD does not represent procedural information.
  - . For example, considerations of loops and decisions must be ignored.
- In drawing the DFD, the designer has to specify the major transforms in the path of the data flowing from the input to output. How those transforms are performed is not an issue while drawing the data flow graph.

DFD

- In a DFD, data flows are identified by unique names. These names are chosen so that they convey some meaning about what the data is.
- Specifying the precise structure of data flows, a data dictionary is often used.
- The associated data dictionary states precisely the structure of each data flow in the DFD.

Software Architecture

Architecture partitions the system in logical parts such that each part can be comprehended independently, and then describes the system in terms of these parts and the relationship between these parts.

Definition

The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them

Role of Software Architecture

1. Understanding and communication.
    - An architecture description is primarily to communicate the architecture to its various stakeholders, which include the users who will use the system, the clients who commissioned the system, the builders who will build the system, and, of course, the architects.
    - Through this description the stakeholders gain an understanding of some macro properties of the system and how the system intends to fulfill the functional and quality requirements. As the description provides a common language between stakeholders, it also becomes the vehicle for negotiation and agreement among the stakeholders, who may have conflicting goals.

Role of Software Architecture (contd..)

2. Reuse
   - The Architecture should have the freedom to reuse the software components already developed
   - The architecture has to be chosen in a manner such that the components which have to be reused can fit properly and together with other components that may be developed.
   - Architecture also facilitates reuse among products that are similar and building product families such that the common parts of these different but similar products can be reused.
   - Architecture helps specify what is fixed and what is variable in these different products, and can help minimize the set of variable elements such that different products can share software parts to the maximum

Role of Software Architecture (contd..)

3. Construction and Evolution
    - As Architecture partitions the system into parts, some architecture-provided partitioning can naturally be used for constructing the system, which also requires that the system be broken into parts such that different teams (or individuals) can separately work on different parts.

4. Analysis

- It is possible to analyze or predict the properties of the system being built from its architecture
    - For example, while building a website for shopping, it is possible to analyze the response time or throughput for a proposed architecture, given some assumptions about the request load and hardware. It can then be decided whether the performance is satisfactory or not, and if not, what new capabilities should be added (for example, a different architecture or a faster server for the back end) to improve it to a satisfactory level.

Software Architecture Views

- In software, the different drawings are called views. A view represents the system as composed of some types of elements and relationships between them.
- Most of the proposed views generally belong to one of these three types
  - Module
  - Component and connector
  - Allocation

Software Architecture Views (contd..)

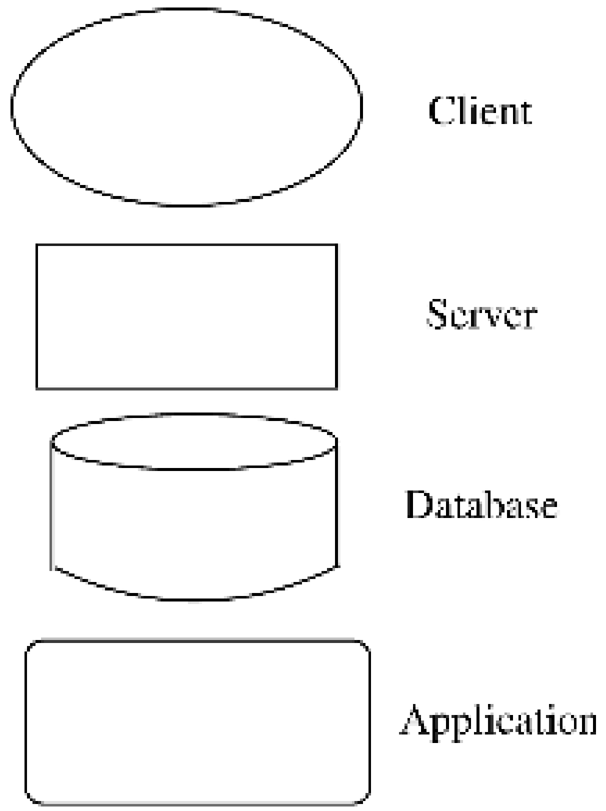
- In a module view, the system is viewed as a collection of code units, each implementing some part of the system functionality
  - Examples of modules are packages, a class, a procedure, a method, a collection of functions, and a collection of classes.
  - The relationships between these modules are also code-based and depend on how code of a module interacts with another module. Examples of relationships in this view are “is a part of” (i.e., module B is a part of module A), “uses or depends on” (a module A uses services of module B to perform its own functions and correctness of module A depends on correctness of module B), and “generalization or specialization” (a module B is a generalization of a module A)

Software Architecture Views (contd..)

- In a component and connector (C&C) view, the system is viewed as a collection of runtime entities called components. That is, a component is a unit which has an identity in the executing system.
  - Objects (not classes), a collection of objects, and a process are examples of components.
  - While executing, components need to interact with others to support the system services. Connectors provide means for this interaction.
    - Examples of connectors are pipes and sockets. Shared data can also act as a connector. If the components use some middleware to communicate and coordinate, then the middleware is a connector.

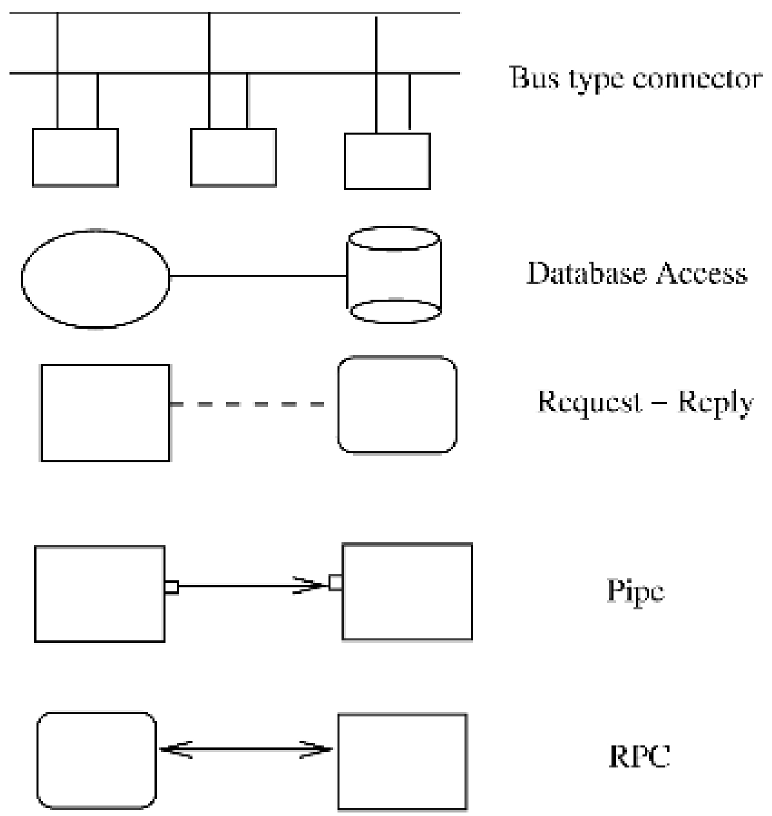
Hence, the primary elements of this view are components and connectors.

Software Architecture Views (contd..)

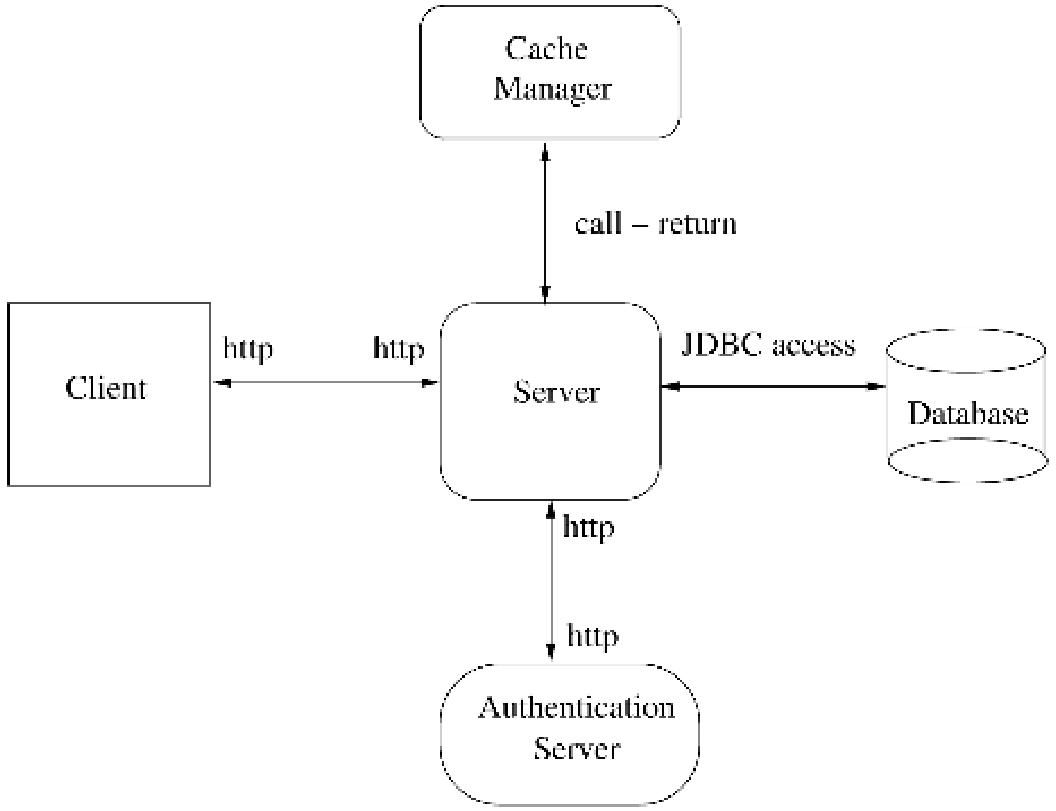


Component examples.

Software Architecture Views (contd..)



Software Architecture Views (contd..)



Software Architecture Views (contd..)

- An allocation view focuses on how the different software units are allocated to resources like the hardware, file systems, and people. They expose structural properties like which processes run on which processor, and how the system files are organized on a file system.

Software Architecture

An architecture description consists of views of different types, with each view exposing some structure of the system. **Module views** show how the software is structured as a set of implementation units, **C&C views** show how the software is structured as interacting runtime elements, and **allocation views** show how software relates to non-software structures. These three types of view of the same system form the architecture of the system.

Planning a Software Project

- Planning is the most important project management activity. It has two basic objectives—establish reasonable cost, schedule, and quality goals for the project, and to draw out a plan to deliver the project goals
- The inputs to the planning activity are the requirements specification and maybe the architecture description.
- There are generally two main outputs of the planning activity: the overall ***Project management plan document*** that establishes the project goals on the cost, schedule, and quality fronts, and defines the plans for managing risk, monitoring the project, etc.; and the detailed plan, often referred to as ***The detailed project schedule***, specifying the tasks that need to be performed to meet the goals, the resources who will perform them, and their schedule.

Planning a Software Project

- Effort Estimation
- Project Schedule and Staffing
- Quality Planning
- Risk Management Planning
- Project Monitoring Plan
- Detailed Scheduling

Software Design

- The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected.
- In the second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided. This design level is often called detailed design or logic design. Detailed design essentially expands the system design to contain a more detailed description of the processing logic and data structures so that the design is sufficiently complete for coding.

Design Concepts

- The goal of the design is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.
- To evaluate a design, we have to specify some evaluation criteria. We will focus on modularity of a system, which is decided mostly by design, as the main criterion for evaluation.
- A system is considered modular if it consists of discrete modules so that each module can be implemented separately, and a change to one module has minimal impact on other modules.

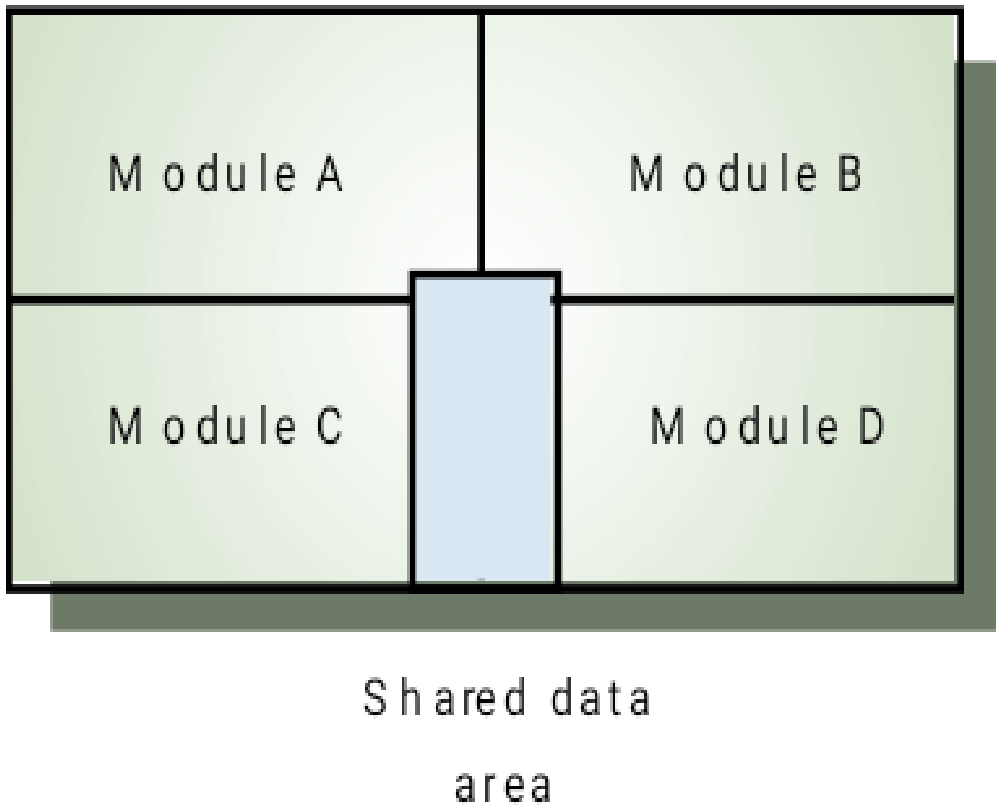
Software Design (contd..)

- To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately.
- Modularization criteria used are
    - Coupling
    - Cohesion
    - open-closed principle

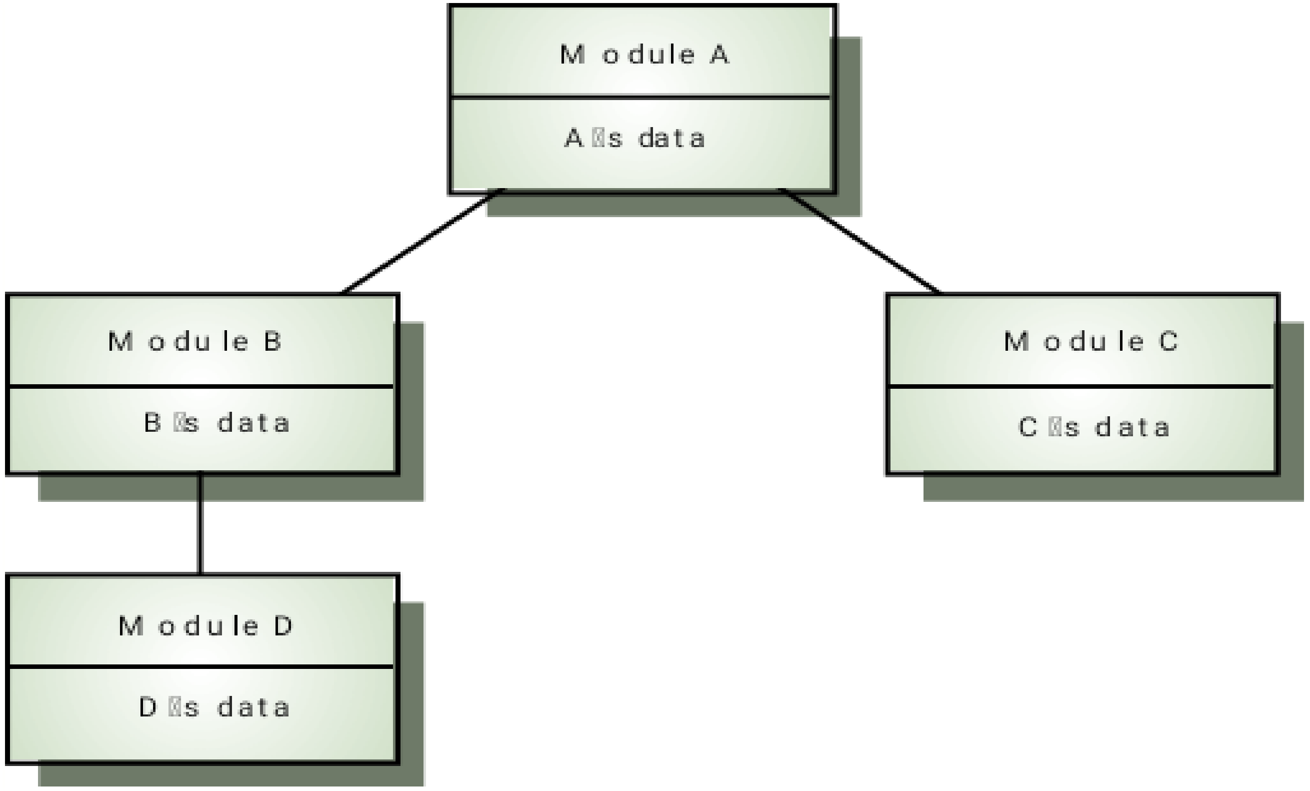
Coupling

- A measure of the strength of the inter-connections between system components.
- **Loose coupling** means component changes are unlikely to affect other components.
  - Shared variables or control information exchange lead to **tight coupling**.
  - Loose coupling can be achieved by state decentralization (as in objects) and component communication via parameters or message passing

Tight Coupling



Loose Coupling



Coupling

- In OO systems, three different types of coupling exist between modules
  - Interaction coupling
    - Interaction coupling occurs due to methods of a class invoking methods of other classes.
  - Component coupling
    - Component coupling refers to the interaction between two classes where a class has variables of the other class.
  - Inheritance coupling
    - Inheritance coupling is due to the inheritance relationship between classes. Two classes are considered inheritance coupled if one class is a direct or indirect subclass of the other.

Cohesion

- Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives the designer an idea about whether the different elements of a module belong together in the same module.
- Cohesion and coupling are clearly related. Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is.

Cohesion Levels

- Coincidental cohesion (weak)
  - Parts of a component are simply bundled together.
- Logical association (weak)
  - Components which perform similar functions are grouped.
- Temporal cohesion (weak)
  - Components which are activated at the same time are grouped.

Cohesion Levels

- Communicational cohesion (medium)
  - All the elements of a component operate on the same input or produce the same output.
- Sequential cohesion (medium)
  - The output for one part of a component is the input to another part.
- Functional cohesion (strong)
  - Each part of a component is necessary for the execution of a single function.
- Object cohesion (strong)
  - Each operation provides functionality which allows object attributes to be modified or inspected

Cohesion

- Cohesion in object-oriented systems has three aspects
  - Method cohesion

    It focuses on why the different code elements of a method are together within the method.
  - Class cohesion

    focuses on why different attributes and methods are together in this class
  - Inheritance cohesion

    Inheritance cohesion focuses on the reason why classes are together in a hierarchy

Structured Design

As a summary-

- **Cohesion** - grouping of all functionally related elements.
- **Coupling** - communication between different modules.

A good structured design has ***high*** cohesion and ***low*** coupling arrangements

Function Oriented Design

- 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.

Design Process

- The whole system is seen as how data flows in the system by means of 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.

Design Notation and Specification

- For function-oriented design, a variety of notations exist. We will look at one notation called **structure charts** which is used with a design methodology known as structured design

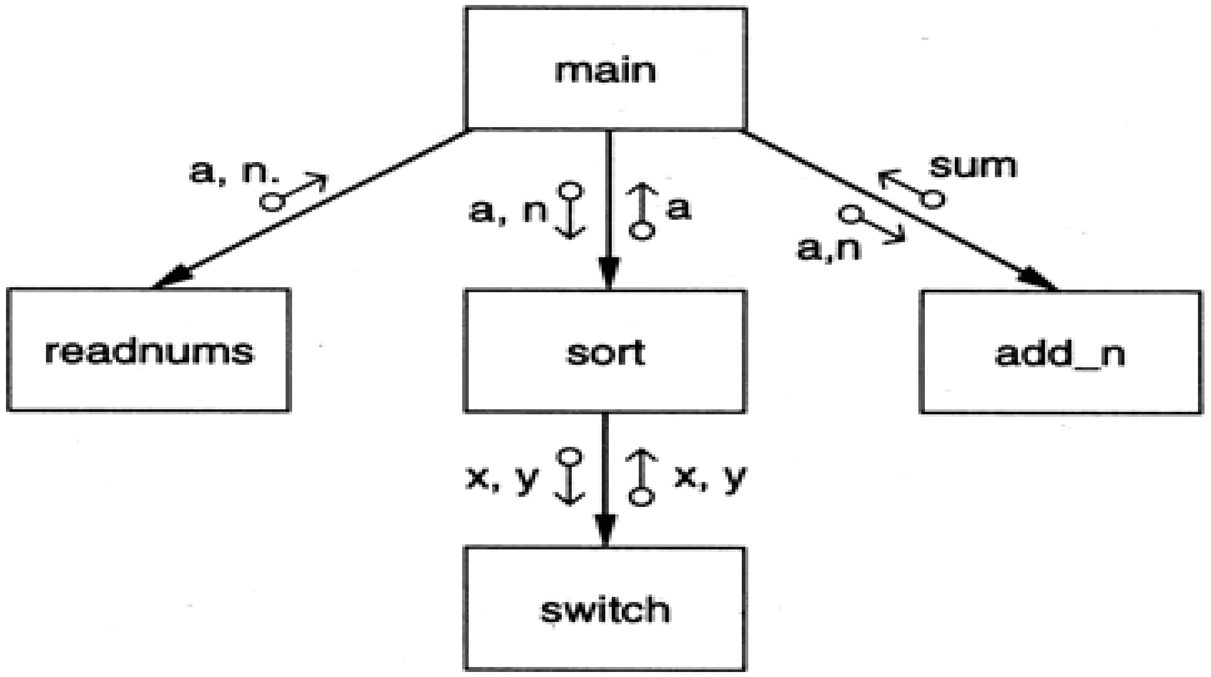
Structure Charts

- A structure chart is a graphical representation of a system's structure; in particular, its modules and their interconnections
  - *Reminder: this is different from object-oriented design: here each module has a well defined function and we are showing how functions work together to achieve a particular objective*
- Each module is represented by a box
- If A uses B, then an arrow is drawn from A to B
  - *B is called the **subordinate** of A*
  - *A is called the **superordinate** of B*

Structure Charts

- An arrow is labeled with the parameters received by B as input and the parameters returned by B as output
    - *Arrows indicate the direction in which parameters flow*
    - *Parameters can be data (shown as unfilled circles at the tail of a label) or control information (filled circles at the tail)*

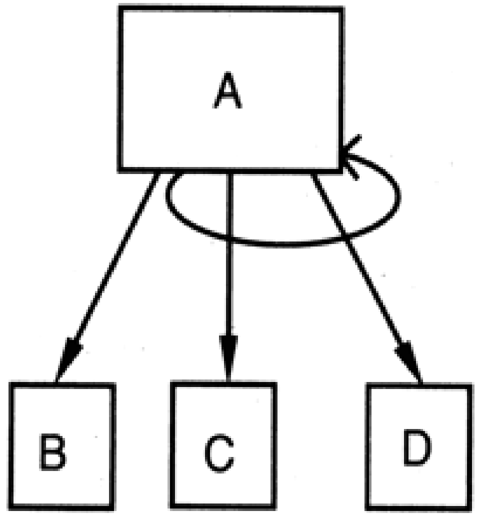
Structure Chart Example

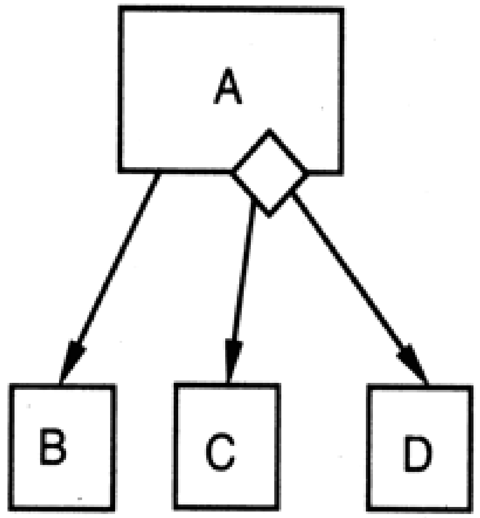


Supporting Iteration and Branching

- Structure charts can also support the specification of module relationships that involve iteration and/or branching
- On the left, A invokes modules C and D in a loop. On the right, A will call either C or D based on some sort of decision. Note: the presence of a decision is indicated by the diamond, but details of the decision (at least above) are documented elsewhere.
-

Supporting Iteration and Branching





Verification

- Designs should be checked for internal consistency and for completeness with respect to the SRS
- If a formal design notation is used, then tools may be able to perform some of these checks
- Otherwise, design reviews (as part of your inspection process) are required to ensure that the finished design is of high quailty

Design reviews - **Metrics**

- Size: Number of Modules x Average LOC expected per module
    - *Or you can generate LOC estimates for each individual module*
- Complexity
    - *Network Metrics*
    - *Stability Metrics*
    - *Information Flow Metrics*

Network Metrics

- Network metrics focus on the structure chart of a system
- They attempt to define how “good” the structure or network is in an effort to quantify the complexity of the call graph
- The simplest structure occurs if the call graph is a tree, with each node having two children
    - *As a result, the graph impurity metric is defined as* ***nodes - edges - 1***
    - *In the case of a tree, this metric produces the result zero since there is always one more node in a tree than edges*
    - *This metric is designed to make you examine nodes that have high coupling and see if there are ways to reduce this coupling*

Stability Metrics

- Stability of a design is a metric that tries to quantify the resistance of a design to the potential ripple effects that are caused by changes in modules
- The creators of this metric argue that the higher the stability of a design, the easier it is to maintain the resulting system
- The basic idea is to define stability in terms of the number of assumptions made by other components about a particular component.
  - *This provides a stability value for each particular module*
  - *A formula is then given to combine the stability value of each module into a value for the entire system*
- In essense, the lower the amount of coupling between modules, the higher the stability of the overall system

Information Flow Metrics

- Information flow metrics attempt to define the complexity of a system in terms of the total amount of information flowing through its modules
- *Approach 1*
  - A module's complexity depends on its intramodule complexity and its intermodule complexity
  - intramodule complexity is approximated by the (estimated) size of the module in lines of code
  - intermodule complexity is determined by the total amount of information (abstract data elements) flowing into a module (inflow) and the total amount of information flowing out of a module (outflow)

Information Flow Metrics(contd..)

- The module design complexity D_c is defined as **D_c = size * (inflow*outflow)^2**
- The term **(inflow * outflow)^2** refers to the total number of input and output combinations, and this number is squared since the interconnections between modules are considered more important to determining the complexity of a module than its code size

Information Flow Metrics(contd..)

- *Approach 2*
  - Approach 1 depends largely on the amount of information flowing in and out of the module
  - Approach 2 is a variant that also considers the number of modules connected to a particular module; in addition, the code size of a module is considered insignificant with respect to a module's complexity
  - The module design complexity D_C is defined as **D_C = (fan_in * fan_out) + (inflow*outflow)**
  - fan_in above refers to the number of modules that call this module, fan_out is the number of modules called by this module

Information Flow Metrics(contd..)

- *Classification*
  - Neither of these metrics is any good, unless they can tell us when to consider a module “too complex”
  - To this end, an approach was developed to compare a module's complexity against the complexity of the other modules in its system
  - **avg_complexity** is defined as the average complexity of the modules in the current design
  - **std_deviation** is defined as the standard deviation in the design complexity of the modules in the current design

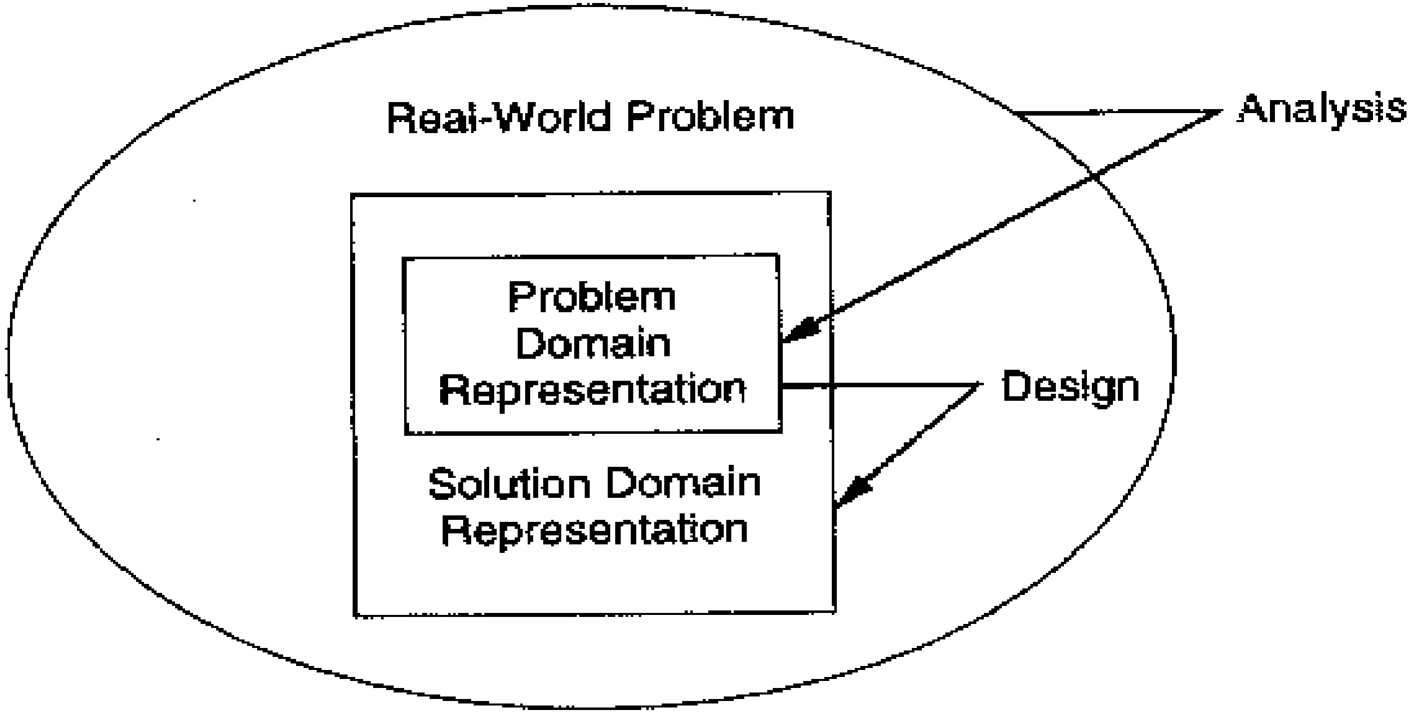
Information Flow Metrics(contd..)

- *Classification (contd..)*
  - A module can be classified as *error prone*, *complex*, or *normal* using the following conditions D_c is the complexity of a particular module
    - A module is error prone if D_c > avg_complexity + std_deviation
    - A module is complex if avg_complexity < D_c < avg_complexity + std_deviation
    - Otherwise a module is considered normal

Object-Oriented Design

- The purpose of Object-Oriented (OO) design is to define the classes (and their relationships) that are needed to build a system that meets the requirements contained in the SRS
- OO techniques can be used in analysis (requirements) as well as design
    - The methods and notations are similar
- In OO analysis we model the problem domain, while in OO design we model the solution domain
- Often structures created during OO analysis are subsumed (reused, extended) in the structures produced by OO design
    - The line between OO analysis and OO design is blurry, as analysis structures will transition into model elements of the target system

Relationship of OO A&D



Design Methodology

- An approach for creating an OO design consists of the following sequence of steps:
  - – Identify classes and relationships between them.
  - – Develop the dynamic model and use it to define operations on classes.
  - – Develop the functional model and use it to define operations on classes.
  - – Identify internal classes and operations.
  - – Optimize and package

Identifying Classes and Relationships

- Identifying the classes and their relationships requires identification of object types in the problem domain, the structures between classes (both inheritance and aggregation), attributes of the different classes, associations between the different classes, and the services each class needs to provide to support the system. Basically, in this step we are trying to define the initial class diagram of the design.

Dynamic Modeling

- The dynamic model of a system aims to specify how the state of various objects changes when events occur. An event is something that happens at some time instance. For an object, an event is essentially a request for an operation. An event typically is an occurrence of something and has no time duration associated with it. Each event has an initiator and a responder. Events can be internal to the system, in which case the event initiator and the event responder are both within the system. An event can be an external event, in which case the event initiator is outside the system (e.g., the user or a sensor).

Functional Modeling

- A functional model of a system specifies how the output values are computed in the system from the input values, without considering the control aspects of the computation. This represents the functional view of the system—the mapping from inputs to outputs and the various steps involved in the mapping. Generally, when the transformation from the inputs to outputs is complex, consisting of many steps, the functional modeling is likely to be useful. In systems where the transformation of inputs to outputs is not complex, functional model is likely to be straightforward.

Defining Internal Classes and Operations

- The final design is a blueprint for implementation. Hence, implementation issues have to be considered. While considering implementation issues, algorithm and optimization issues arise. These issues are handled in this step.
- Then the implementation of operations on the classes is considered. For this, rough algorithms for implementation might be considered. While doing this, a complex operation may get defined in terms of lower-level operations on simpler classes.
- Once the implementation of each class and each operation on the class has been considered and it has been satisfied that they can be implemented, the system design is complete.

Optimize and Package

- During design, some inefficiencies may have crept in. In this final step, the issue of efficiency is considered, keeping in mind that the final structures should not deviate too much from the logical structure produced. Various optimizations are possible and a designer can exercise his judgment keeping in mind the modularity aspects also.

Complexity Metrics for OO Design

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Coupling between Classes (CBC)
- Response for a Class (RFC)

Weighted Methods per Class (WMC)

- The effort in developing a class will be determined by the number of methods the class has and the complexity of the methods. Hence, a complexity metric that combines the number of methods and the complexity of methods can be useful in estimating the overall complexity of the class.
- The weighted methods per class (WMC) metric does precisely this.
  - Suppose a class C has methods M1,M2, ...,Mn defined on it. Let the complexity of the method Mi be ci.
- The WMC is defined as WMC = $\sum_{i=1}^{i=n} ci$
  - the complexity of each method is considered to be 1, WMC gives the total number of methods in the class.

Depth of Inheritance Tree (DIT)

- The DIT of a class C in an inheritance hierarchy is the depth from the root class in the inheritance tree. In other words, it is the length of the shortest path from the root of the tree to the node representing C or the number of ancestors C has. In case of multiple inheritance, the DIT metric is the maximum length from a root to C.

Coupling between Classes (CBC)

- Coupling between classes (CBC) is a metric that tries to quantify coupling that exists between classes. The CBC value for a class C is the total number of other classes to which the class is coupled. Two classes are considered coupled if methods of one class use methods or instance variables defined in the other class

Response for a Class (RFC)

- The RFC value for a class C is the cardinality of the response set for a class. The response set of a class C is the set of all methods that can be invoked if a message is sent to an object of this class. This includes all the methods of C and of other classes to which any method of C sends a message

Detailed Design

The internal logic that will implement the given specifications is the focus of this section

Logic/Algorithm Design

State Modeling of Classes

Logic/Algorithm Design

- The most common method for designing algorithms or the logic for a module is to use the stepwise refinement technique. The stepwise refinement technique breaks the logic design problem into a series of steps, so that the development can be done gradually. The process starts by converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements. In each step, one or several statements in the algorithm developed so far are decomposed into more detailed instructions. The successive refinement terminates when all instructions are sufficiently precise that they can easily be converted into programming language statements.

State Modeling of Classes

- A method to understand the behavior of a class is to view it as a finite state automaton, which consists of states and transitions between states. When modeling an object, the state is the value of its attributes, and an event is the performing of an operation on the object. A state diagram relates events and states by showing how the state changes when an event is performed. A state diagram for an object will generally have an initial state, from which all states are reachable

State Modeling of Classes

- The finite state modeling of objects is an aid to understand the effect of various operations defined on the class on the state of the object. A good understanding of this can aid in developing the logic for each of the operations. To develop the logic of operations, regular approaches for algorithm development can be used. The model can also be used to validate if the logic for an operation is correct