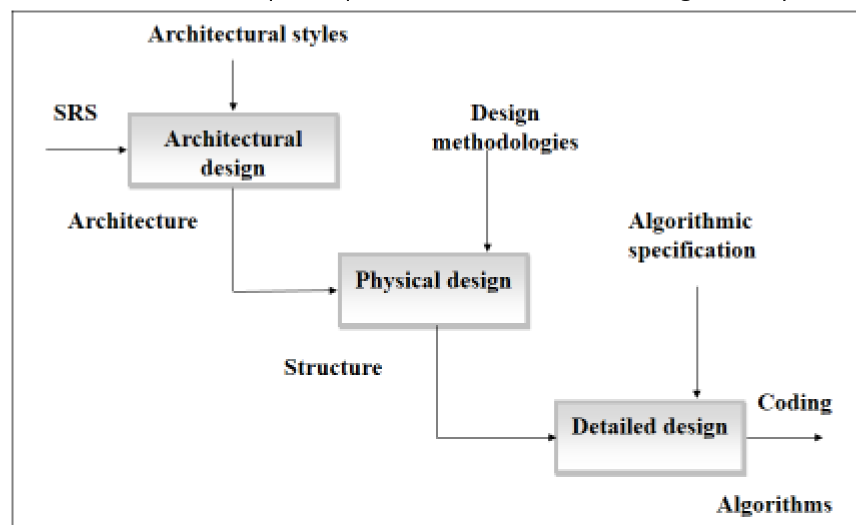
UNIT 3

Software Design

- Software design is a crucial phase of the software development life cycle that focuses on the solution domain of a system.
- Software design is the process of describing the blueprint or sketch of the final software product in the form of a design model.
- Design aims at producing software architecture, establishing structural relationships among modules, and describing the algorithmic details of each module.
- The design model encompasses an architecture on which it is developed, interfaces with other modules, and a design document.

Software Design Process

- A software design process is a set of design activities carried out in the design phase to produce a design model from the SRS.
- The software design process basically consists of three design phases or design levels, viz., *architectural design, physical design,* and *detailed design*.
- *Architectural design* is an external design which considers the external behavior of a software product.
- The external design considers the architectural aspects related to business, technology, major data stores, structure of data and modules, reports, performance criteria, and high-level process structure of the product.



- *Physical design* is a high-level design or structural design which is concerned with refining the conceptual view of the system; identifying the major modules; decomposing the modules into sub-modules, interconnections among modules, data structure, and data store in the system.
- Software design methodologies (e.g., structured design, object oriented, Jackson structured design, W-Orr, etc.) are used to produce the physical design.
- *Detailed design* is the algorithmic design of each module in the software. It is also called logical design.

- The detailed design concentrates on the specification of algorithms and data structures of each module, the actual interface descriptions and data stores of the modules, and package specifications of the system.

Characteristics of a Good Software Design

The desirable characteristics that a good software design should have are as follows:

- *Correctness*
- *Efficiency*
- *Understandability*
- *Maintainability*
- *Simplicity*
- *Completeness*
- *Verifiability*
- *Portability*
- *Modularity*
- *Reliability*
- *Reusability*

Design Principles

- There are certain design concepts and principles that govern the building of quality software designs.
- Some of the common concepts of software design are:
  - ❖ *Abstraction*
  - ❖ *information hiding*
  - ❖ *functional decomposition*
  - ❖ *design strategies*
  - ❖ *Modularity*
  - ❖ *and modular design*

Abstraction

- The process of describing a problem at a higher level of representation without bothering about its internal details.
- Problem partitioning and abstraction are closely related in a software design.
- There are two levels of abstraction
  - *high-level abstraction*
  - *low-level abstraction*

- Software engineering practitioners think software development is the movement in different levels of abstraction.
- There are three types of abstraction, namely, *functional abstraction, data abstraction, and control abstraction*.
- Functional abstraction specifies the functions that a module performs in the system.
- Function prototype, function call, closed subroutine are some examples of functional abstraction.
- Data abstraction specifies the entities or data objects that provide certain services to the external environment.
- Abstract data types (ADTs), such as structures in C, classes in C++, and packages in Java are the examples of data abstraction. A more detailed example for the ADT of stack in C++ is as follows:
- Control abstraction provides the operational characteristics of the system without describing their implementation details.
- For example, loops, iterations, frameworks, and multithreading describe control abstraction.

Advantages of abstraction:

- It separates design from implementation, which is easy to understand and manage.
- It helps in problem understanding and software maintenance.
- It reduces the complexity of modern computer programming for software users and engineers.
- It helps in program organization that can be generalized for recovering common problems and therefore it promotes software reuse.
- It also promotes scalability and helps in making early design decisions.

Information Hiding

- Information hiding is an important design principle which is expressed through encapsulation and abstraction.
- It helps in modularization of software projects into small components.
- It allows the programmers to change the implementation of the application for better performance.

Functional Decomposition

- Functional decomposition is the process of partitioning a large and complex problem into small, manageable, and understandable pieces.
- It is performed using abstraction and information hiding.
- The decomposition process uses “divide and conquer” approach to divide the software into independent parts.
- Decomposition is performed in different manners in different design methodologies (e.g., structured, object oriented, component-based development, etc.).
- The decomposed parts are organized into a hierarchy of components

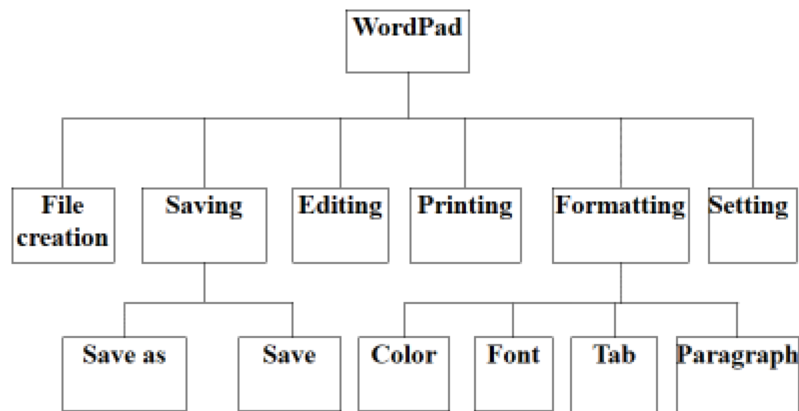


Figure 6.2: Functional decomposition of WordPad

Design Strategies: Top-down and Bottom-up

- A hierarchical organization helps in taking design decisions and performing design activity.
- A design activity varies with design techniques, such as structured design, Jackson structured design, object-oriented design, etc.
- The design techniques are based on design strategies that reflect the quality of design.
- *Top-down* and *bottom-up* are the most popular design strategies used in the industry.
- In the top-down strategy, the system is viewed as a single "black-box" program with a high-level interface with the external environment. It starts with a general level of specification and moves to a specific level of specifications.
- In the bottom-up strategy, specific levels of details are designed and further these are linked together to design the final system.

Design Strategies: Top-down strategy

- A top-down strategy (also referred to as *stepwise refinement*) is essentially partitioning a system to elaborate on its subsystems.
- It starts with the global view defined at a high level of abstraction of the overall system.
- The system is refined and decomposed into the next lower-level subsystems.
- Each subsystem is again decomposed into the specific level of detail to identify the concrete level of the subsystems.
- The process of elaboration is continued until we reach at the concrete level of detail.
- At this level, design decisions are taken easily and subsystems can easily be developed and managed.

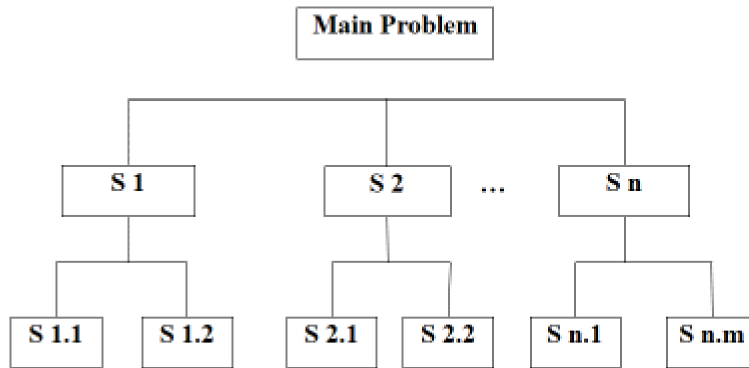


Figure 6.3: Top-down strategy

Design Strategies: Bottom-up strategy

- A bottom-up strategy (also referred to as *layers of abstraction*) is piecing together the subsystems to form the whole system.
- It starts with the lower-level subsystems at the bottom level, i.e., the individual elements in the system.
- The systems are then combined together to form the upper level of abstraction, i.e., subsystems.
- In turn, subsystems are again put together to design the next level of the system.
- This process of layering the abstraction levels is continued until the top level of the system is reached.
- Each level of abstraction performs services to its upper level of the system.

Design Strategies: Bottom-up strategy

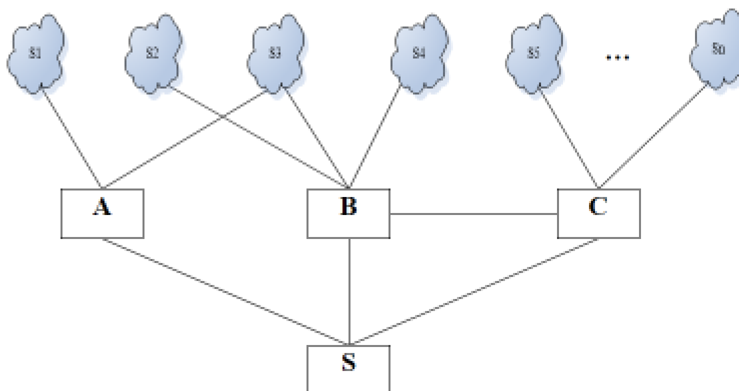


Figure 6.4: Bottom-up strategy

Modularity

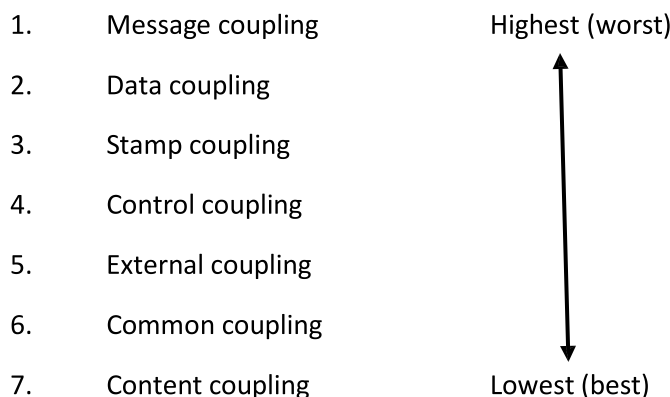
- Modularization is the process of breaking a system into pieces called modules so that these can be easily managed and implemented.

- A module is a part of a software system that can be separately implemented and a change in a module has a minimal effect on other modules.
- A module can be a function, procedure, program, subroutine, class, package, framework, library files, templates, components, etc.
- A modular system consists of various modules linked via interfaces.
- An *interface* is a kind of link or relationship that combines two or more modules together.
- Modularity is the measurement of modularization of a system into pieces.
- A module has the following characteristics:
- Modularity measures the independency of the parts of a system and enhances separation of concerns.
- Modularity enhances quality factors, such as portability, extensibility, compatibility, scalability, etc.
- A module contains data structures, input/output statements, instructions, and processing logic.
- A module can be called into another module.
- A module can be reused within other modules.
- A modular system can be easily developed, maintained, and debugged.
- Modularity reduces design complexities in a system through distributed software architectures.
- Modularity uses abstraction, which helps in defining a subsystem.
- Modularity improves design clarity and understandability.
- A modular design focuses on minimizing the interconnections between modules.
- In a modular system design, several independent and executable modules are composed together to construct an executable application program.
- The programming language support, interfaces, and the information-hiding principles ensure modular system design.
- The most common criterions are functional independency; levels of abstraction; information hiding; functional diagrams, such as DFD, modular programming languages, coupling, and cohesion.
- *An effective modular system has low coupling and high cohesion*

Coupling

- It is the strength of interconnection between modules. It is the measure of the degree of interdependency between modules.
- Modules are either loosely coupled or strongly coupled.
- In strong coupling, two modules are dependent on each other. Strong coupling can be observed in assembly language programs where change in one part or data requires changes in other parts of the system.
- Also, these are difficult to reuse, test, and release for the operation.

- In loose coupling, there are weak interconnections between modules.
- Interdependency between modules increases as coupling increases.
- Interconnection between modules can be measured by the number of function calls, number of parameters passed, return values, data types, sharing of data files or data items, etc.
- The strength of interconnection between modules is influenced by the level of *complexity of the interfaces, type of connection, and the type of communication*.
- The complexity of an interface is the measure of the type of parameters, number of parameters, common sharing of data or code communicated by modules.
- The connection between modules is established by relating one module to another through parts of the system or through certain data value.
- Communication is the data passed, type of data, and the control information passed to another module.
- Types of coupling:



Types of coupling

Message Coupling: Message coupling is the lowest (i.e. best) type of coupling which exists between modules. For example, in C++ objects communicate with each other by sending a message through parameters in the function call.

Data Coupling: Data coupling exists between modules when data are passed as parameters in the argument list of the function call. Each datum is a primary data item (e.g., integer, character, float, etc.) that can be used between modules.

Stamp Coupling: It occurs between modules when data are passed by parameters using complex data structures, which may use parts or the entire data structure by other modules. For example, structures in C, records in Pascal, etc.

Control Coupling: It exists when one module controls the flow of another by passing control information such as flag set or switch statements. For example, a flag variable decides what function or module is to be executed next.

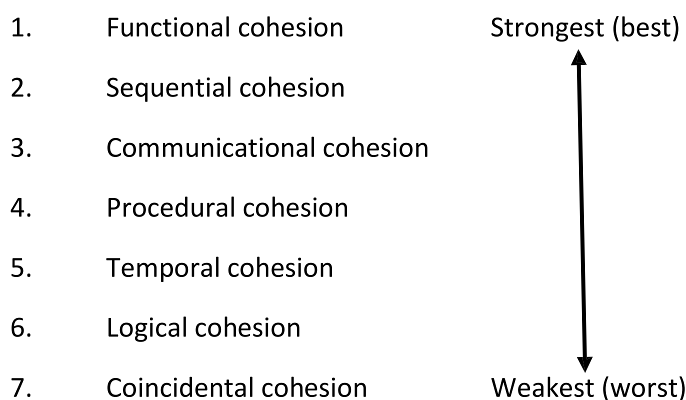
External Coupling: External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

Common Coupling: Common coupling is when two modules share common data (e.g., a global variable). In C language, external data items are accessed by all modules in the program. If there is any change in the shared resource, it influences all the modules using it.

Content Coupling : It is the highest coupling (worst). Content coupling exists between two modules when one module refers or shares its internal working with another module. Accessing local data items or instructions of another module is an example of content coupling.

<u>Cohesion:</u>

- Cohesion of a single module is the degree to which the elements of a single module are functionally related to achieve an objective.
- Module cohesion represents how tightly bound the internal elements of the module are to one another.
- High cohesion is often characterized by more understandability, modifiability, and maintainability of the modules in a system.
- Low cohesive modules are highly undesirable and should be modified or replaced to meet the objectives of modular design.
- The important goal of designers is to maximize cohesion and minimize coupling.
- Cohesion between the elements of a module is measured in terms of the strength of the hiding of the elements within the module itself.
- A functionally independent module has higher cohesion as compared to dependent modules.
- Levels of cohesion:



<u>Levels of cohesion</u>

Functional Cohesion: In a functionally cohesive module, all the elements of the module perform a single function. For example, “log” computes the logarithm of a number and “printf” prints the results.

Sequential Cohesion: Sequential cohesion exists when the output from one element of a module becomes the input for some other element.. For example, "withdraw money" and "update balance" both are bound together to withdraw money from an account.

Communicational Cohesion: In communicational cohesion, all the elements of a module operate on the same input or output data. For example, "print and punch the output file" can be communicational cohesion. The binding of elements in a module is higher than procedural cohesion.

Procedural Cohesion: Procedural cohesion contains the elements which belong to a common procedural unit. The functions are executed in a certain order. For example, "entering, reading, and verifying the ATM password" are bound in an ordered manner for the procedurally cohesive module "enter password."

Temporal Cohesion: Sometimes, a module performs several functions in a sequence but their execution is related to a certain time. For example, "a database trigger is activated on executing a certain procedure."

Logical Cohesion: Logical cohesion exists when logically-related elements of a module are placed together. All the parts communicate with each other by passing control information such as flag variable, using some shared source code, etc.

Coincidental Cohesion: It occurs when the elements within a given module have no meaningful relationship to each other.

Design Methodologies:

- A design methodology provides the techniques and guidelines for the design process of a system.
- The goal of all design methodologies is to produce a design for the solution of a system.
- A design process consists of various design activities.
- The most popular design methodologies are:
    - Function-oriented design
    - Object-oriented design

Function-oriented design

- Function-oriented design is a mature design methodology for software design. It begins with the requirements document, i.e., SRS to understand different modules.
- It follows the top-down design strategy in which focus is initially given on the global perspectives (main file, global data, records, external interaction, etc.) of the overall system.
- Thereafter, the system is decomposed into subsystems using the top-down strategy.
- The subsystems are again refined into more detailed functional levels.
- Decomposition is continued until we reach the concrete level.
- The concrete level states are easy to convert into programming languages.
- Some function-oriented design methodologies:
    - Structured design methodology [Yourdon – 1979]
    - Jackson structured design methodology [Jackson –1975]

- Warnier-Orr methodology [Warnier-Orr 1977 , Warnier-Orr 1981]
- Step-wise refinement methodology [Wirth – 1971]
- Hately and Pirbhai's methodology [Hatley – 1987]

Object-Oriented Design

- Object- oriented design deals with the real world entities of the environment for problem solving.
- The entities are characterized by objects.
- Similar objects are combined into a group called a class.
- A class contains data and its related functions that it will perform.
- The object-oriented approach follows the bottom-up strategy for design.
- Object-oriented design methodologies:
    - *Shlaer/Mellor methodology [Shlaer – 1988]*
    - *Coad/Yourdon methodology [Coad – 1991]*
    - *Booch methodology [Booch – 1991]*
    - *OMT methodology [Rumbaugh – 1991]*
    - *Wirfs-Brock methodology [Wirfs-Brock – 1990]*
    - *OOSE objectory methodology [Jacobson – 1992]*
    - *UML (Unified Modeling Language) [Rational – 1997]*

Structured Design

- Structural design is one of the most widely used function-oriented design methodology which follows mainly the top-down design strategy .
- The basic approach of structured design is to transform the data flow diagrams of structural analysis into structure charts.
- Structured design is represented through the structure chart and it follows the transform analysis and transaction analysis to produce the design of the system.
- It is based on functional decomposition, which concentrates on identifying the conceptual view of the system and its elaboration and refinement in a top-down manner.
- The modularization criteria (i.e. coupling and cohesion) are used to represent the design decisions.
- The structure chart is a graphical representation of procedural programs in the structured design methodology.
- It represents the modules of a system in a hierarchical fashion.
- It shows the dependency between the modules and the parameters that are passed among the different modules.
- It produces a software structure that can be easily implemented in programming languages.

- The building blocks in the software are represented through modules and these are linked together to produce the final design.

Symbols used to represent the structure chart:

- **Affluent module:** The affluent module, also known as the input module, receives information from a subordinate module and passes to a superordinate module. This module is used to collect the input data.
- **Efferent module:** The efferent module, also known as the output module, passes information from a superordinate to a subordinate module. This module is used to produce intermediate or final outcomes.
- **Transform module:** The transform module performs data transformation. It receives data, performs some operations, and represents them into another form.
- **Coordinate module:** The coordinate module manages the transformations communication between subordinate modules.
- **Composite module:** The composite module can perform functions at more than one module.
- **Selection:** The selection of a module is represented through a diamond box. The control couple decides which subordinate module is to be invoked for further operation.
- **Repetition:** The repetition of a module is represented by a looping arrow around the modules to be iterated in the program.
- **Library module:** The library module is represented by a rectangle with double edges. The frequently called modules are iterated in the design.

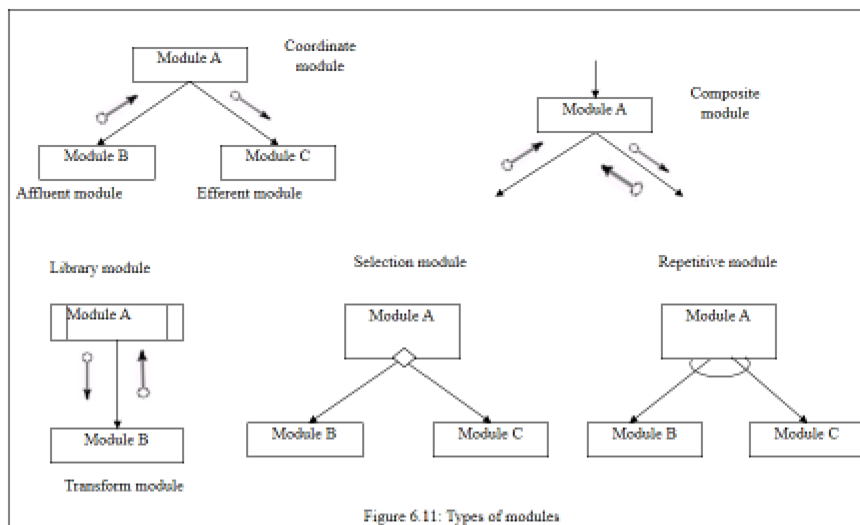


Figure 6.11: Types of modules

An example of structure chart

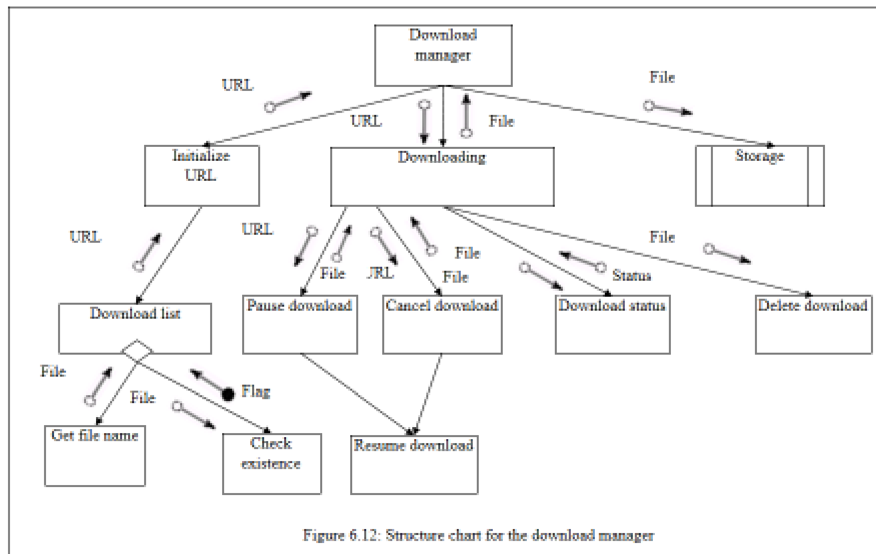


Figure 6.12: Structure chart for the download manager

Structure Chart versus Flowchart:

- Although flowcharts and structure charts are used for problem solving of procedural programs, both are different in the representation of their designs layouts.
- The structure chart shows the task to be performed by the program, whereas a flowchart shows how the program will perform the task.
- Also, a flowchart does not state the algorithmic details of the system.
- In a structure chart, modules may be implemented separately, which can be modified in the later stages, while a flowchart represents a single snapshot of the problem and its flow of information.
- Therefore, it becomes difficult to identify and separate program execution in modules.
- The structured design methodology (SDM) is a systematic approach for decomposing and organizing different modules in a structure chart.
- The SDM uses data flow diagrams constructed during structured analysis to perform structured design.
- There are two important strategies used for transforming data into the structure charts:
    - *Transform analysis*
    - *Transaction analysis*

Transform analysis:

- Transform analysis is applicable in the situations where there is a single path of any transformation, i.e., all input data are incoming to the transform module.
- It is applicable for small functional problems such as computational, scientific, and engineering computations.

Transaction analysis

It is used when there are multiple paths emerging at any moment. During transaction analysis, transform analysis can be applied in individual modules for refinements

- The structured design methodology using transform analysis is as follows:

    1. *Review and refine the data flow diagram.*
    2. *Identify boundaries between input, process, and output segments.*
    3. *Apply first-level factoring using design principles and modularization criteria.*
    4. *Perform additional factoring on input, process, and output segments.*

1. Review and Refine Data Flow Diagram

- This DFD is drawn to understand the problem domain. It covers the external environment and the flow of data in the existing system.
- A functional DFD is extracted from the existing DFD by refining it to hold into transformational form.

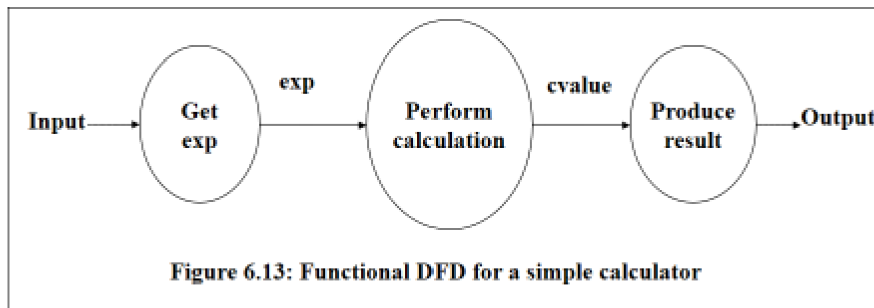


Figure 6.13: Functional DFD for a simple calculator

- The important functions with its major input and output segments are considered in it.
- The input and output segments have several subparts inherent in them.
- The functional part is also known as *central transform*. Central transform exists between inputs and outputs.

2. Identify Boundaries between Input, Process, and Output Segments

- The boundary between input stream, central transform, and output stream is marked by identifying the most abstract input data and the most abstract output data.
- The most abstract input in a DFD is the input stream that can no longer be identified.
- Similarly most abstract output is the first output stream identified in the DFD.
- The control which performs the basic transformation for the system exists between the most abstract input and the most abstract output.
- The boundary is drawn as arcs between the most abstract input and central transform; and central transform and the most abstract output.

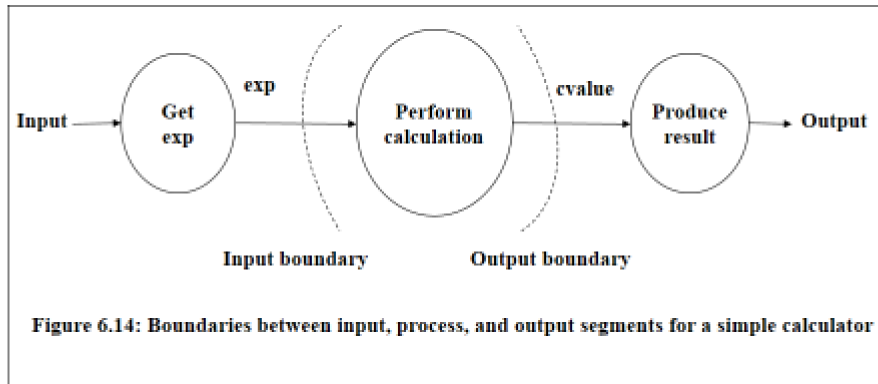


Figure 6.14: Boundaries between input, process, and output segments for a simple calculator

3. Apply First-Level Factoring using Design Principles and Modularization Criteria

- First-level factoring is performed after identifying the most abstract input and the most abstract output
- Central transform is connected to the main module and it is considered a coordinate module between the input and output data streams.

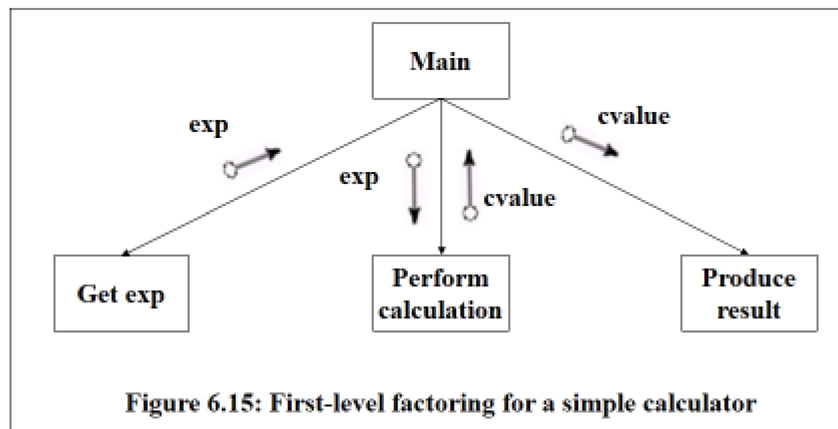


Figure 6.15: First-level factoring for a simple calculator

- The input and output modules become the subordinate modules to the main module.
- There may be many transform, input, and output modules in the system.

4. Perform Additional Factoring on Input, Output, and Transform Module

- Additional factoring is done on each input module, central transform, and output module depending upon the complexity and size of the modules.
- Factoring of modules is continued until we reach the modules that are corresponding to the source or data stores, or until it becomes sufficient to transfer them to the implementation level.
- Factoring is performed using all the design principles such as abstraction, functional decomposition, modularity, information hiding, etc.
- Finally, other modules such as error handling, security, backup etc., are added to the modules wherever they are required in the design.

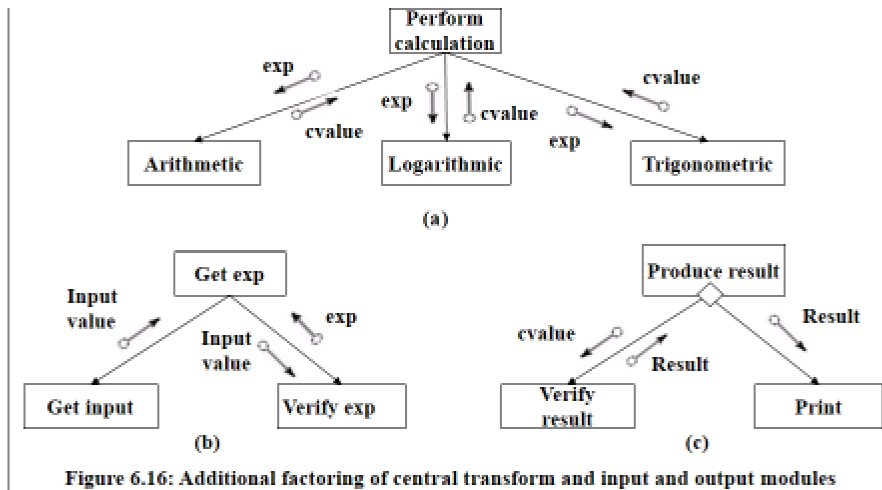


Figure 6.16: Additional factoring of central transform and input and output modules

Transaction Analysis:

- Transaction analysis is similar to transform analysis but there may be several paths from any transaction in the DFD.
- The action of each path is dependent on the input command.

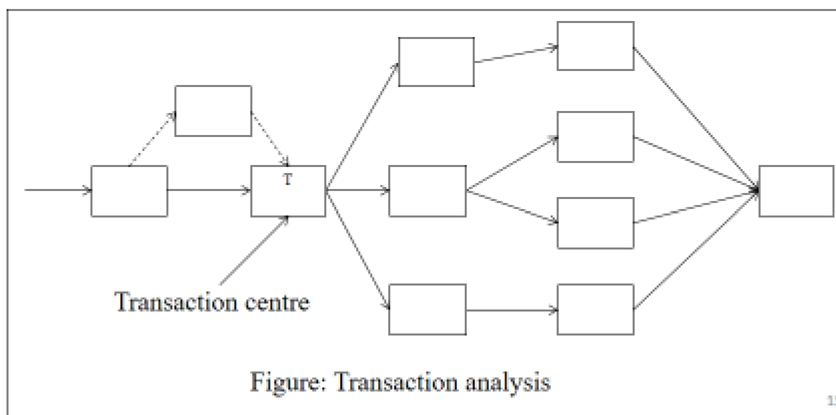


Figure: Transaction analysis

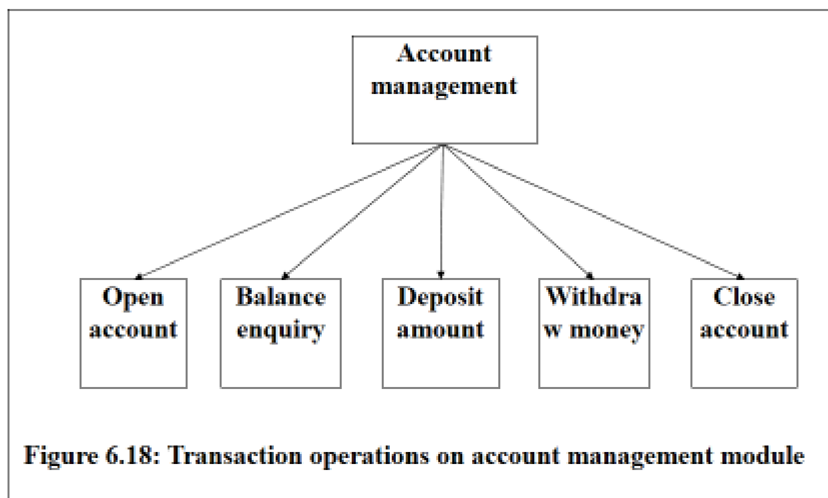


Figure 6.18: Transaction operations on account management module

Detailed Design:

- Detailed design concentrates on specifying the procedural descriptions, data structure representation, interfaces between data structures and functions, and packaging of the software product.
- Detailed design of a system is closer to its implementation.
- Detailed design helps to specify implementation decisions such as procedural details, data structure, interfaces, packaging information etc.
- Detailed design tools: Algorithm, pseudo code, PDL, HIPO diagram, structured English, data structure diagrams, structured flow charts, and so on.

Program Design Language (PDL):

- Program Design Language (PDL) is a software design tool which is used to produce structured design in a top-down manner.
- PDL generates pidgin language that is similar to structured English.

```
SUM (ARRAY A, N)
    Initialize total to 0
    DO FOR N input
            read a number into A
    ENDDO
    DO WHILE there are numbers in A
            add number to total
    print (total)
END
```

Figure: PDL program for finding the sum of n numbers

- PDL covers all important requirements required during implementation. These are the procedures, procedure calls, global data, control blocks, interface definitions, error situations, processing procedures of PDL programs, etc.
- The most common constructs used in PDL are IF-THEN-ELSE, DO, DO-WHILE, DO-UNTIL, DO-FOR-EXCEPT, and CASE-OF.
- PDL programs can be used to produce source codes of programming languages through PDL procedures.

Algorithmic Design:

- An algorithm is a step-by-step process of problem solving.
- n algorithm has two parts, namely, problem definition and design of algorithm.
- Problem definition states the problem to be solved by the algorithm.

- The design of algorithm is the sequence of steps performed to solve the problem.
- An algorithm is written in a formal statements or pseudo code.
- The design of an algorithm is done through step-wise refinement, in which algorithm is broken down into smaller parts so that it can be solved in a convenient and manageable manner.

Figure 6.20: An algorithm for binary search

Algorithm Binary_Search (A, N, X). This algorithm searches the element X from in an array A of N elements arranged in an ascending order. The variables Low, Mid, and High represent the lower, middle, and upper limits of the search interval, respectively. This algorithm prints the middle element in case of successful search, otherwise it prints unsuccessful search.

```
1. [Initialize]
     Low = 1, High = N
2. [Perform search]
     Repeat thru step 4 while Low<=High
3. [Find the middle element]
     Mid = [(Low + High)/2]
4. [Compare]
     If X < A[Mid]
     then High= Mid-1
     else If X > A[Mid]
     then Low = Mid + 1
     else Write ('Successful search')
     Write (Mid)
5. [Unsuccessful search]
     Write ('Unsuccessful search')
     Write ('Element not found')
```

Design Verification:

- Design verification ensures that all the requirements have been incorporated in the design, modules and their interfaces are accurately specified, and there is a systematic flow of information among modules.
- A good design can easily be modified and maintained in the future. Also a systematic design can be easily converted into the programming languages.
- Two most common approaches used for design verification are *design reviews* and *design inspections*.

Design reviews:

- Design reviews are conducted at the end of software design.
- They are conducted by a group of people involving a project leader and software engineers.
- A review meeting is called to discuss whether all the requirements have been included in the design and whether it satisfies all the conditions and constraints laid down in the SRS document.
- It tries to recover design error.

Design inspection

- Design inspection is performed by expert team members who are experienced in the functional area.
- A checklist of items is provided to the team members to inspect the design.

- The team generally consists of a project coordinator, a designer, a software engineer, and a tester.
- The checklist covers different components of requirements, quality factors and design standards, design principles, and modularity and design package issues.
- Design is verified with the help of a checklist and a detailed report is prepared and given to the project coordinator.

Conclusion:

- Software design is an important phase of software development life cycle that focuses on the solution domain.
- A good software design has several traits, such as correctness, efficiency, understandability, maintainability, simplicity, completeness, verifiability, portability, modularity, reliability, and reusability.
- Deciding the right architecture is crucial to the success of most of the software system.
- A design methodology provides the techniques and guidelines for the design process of a system. The important design methodologies are function-oriented design and object-oriented design.
- The structured design methodology (SDM) is a systematic approach used for decomposing and organizing different modules in a structure chart.
- Detailed design concentrates on specifying the procedural descriptions, data structure representation, interfaces between data structures and functions, and packaging of the software product.
- Design verification ensures that all the requirements have been incorporated in the design, modules and their interfaces are accurately specified, and there is a systematic flow of information among the modules.