Semester: V	SOFTWARE ENGINEERING (Professional Elective - I) Common to CSE-AIML,CSE-DS	L	T	P	C
Code:		3	0	0	3

Course Objectives:

- Software life cycle models, Software requirements and SRS document.
- Project Planning, quality control and ensuring good quality software.
- Software Testing strategies, use of CASE tools, Implementation issues, validation &verification procedures.

Course Outcomes (COs):

At the end of the course students will be able to:

CO1	Understand software engineering evolution and apply life cycle models
CO2	Analyze and prepare SRS, project estimation & risk management techniques.
CO3	Design software using modularity, cohesion, coupling, agile methods, and UI design principles.
CO4	Implement and test software with coding standards, testing strategies, ISO 9000, CMM
CO5	Apply CASE tools, software maintenance, reuse, reverse engineering

Unit–III

Software Design: Overview of the design process, How to characterize a good software design? Layered arrangement of modules, Cohesion and Coupling. approaches to software design.

Agility: Agility and the Cost of Change, Agile Process, Extreme Programming (XP), Other Agile Process Models, Tool Set for the Agile Process

Function-Oriented Software Design: Overview of SA/SD methodology, Structured analysis, Developing the DFD model of a system, Structured design, Detailed design, and Design Review.

User Interface Design: Characteristics of a good user interface, Basic concepts, Types of user interfaces, Fundamentals of component-based GUI development, and user interface design methodology.

SOFTWARE DESIGN

OVERVIEW OF THE DESIGN PROCESS

The activities carried out during the design phase (called as *design process*) transform the SRS document into the design document.

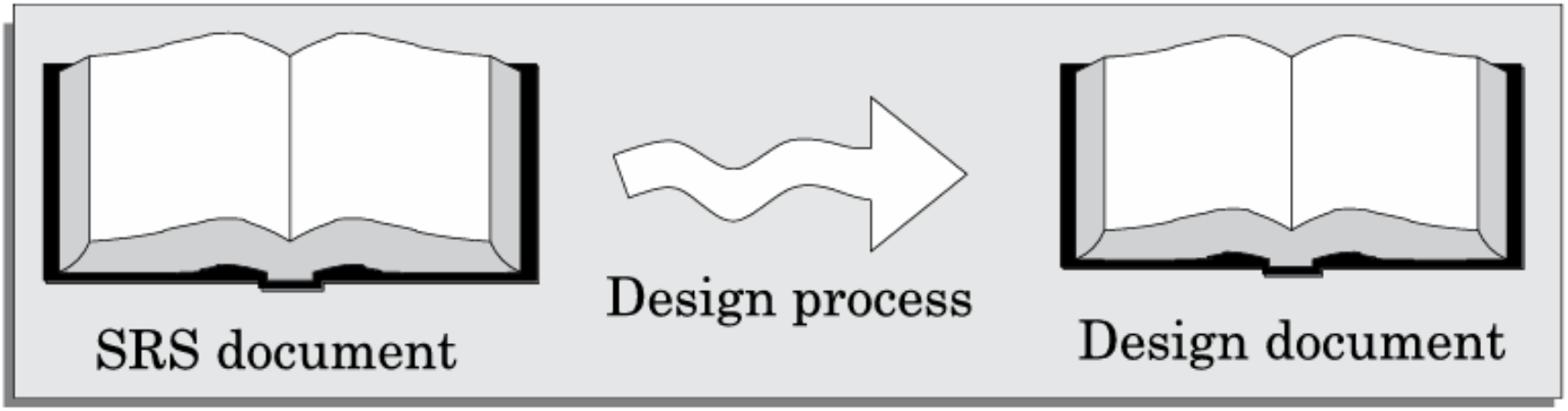


FIGURE 5.1 The design process.

Outcome of the Design Process

Different modules required:

- The different modules in the solution should be identified.
- Each module is a collection of functions and the data shared by these functions.
- Each module should accomplish some well-defined task out of the overall responsibility of the software.
- Each module should be named according to the task it performs.

Control relationships among modules:

- A control relationship between two modules essentially arises due to function calls across the two modules.
- The control relationships existing among various modules should be identified in the design document.

Interfaces among different modules:

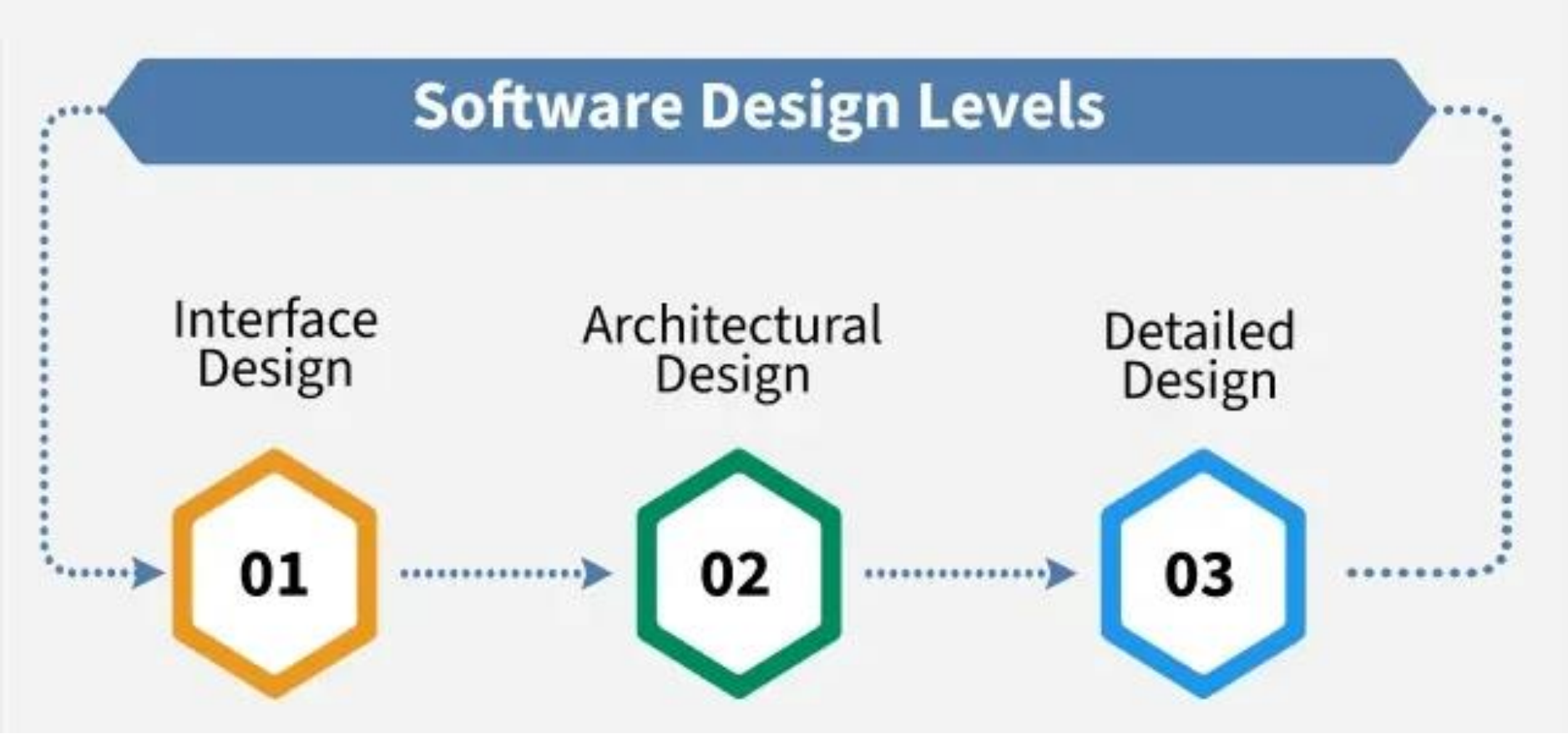
- The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.

Data structures of the individual modules:

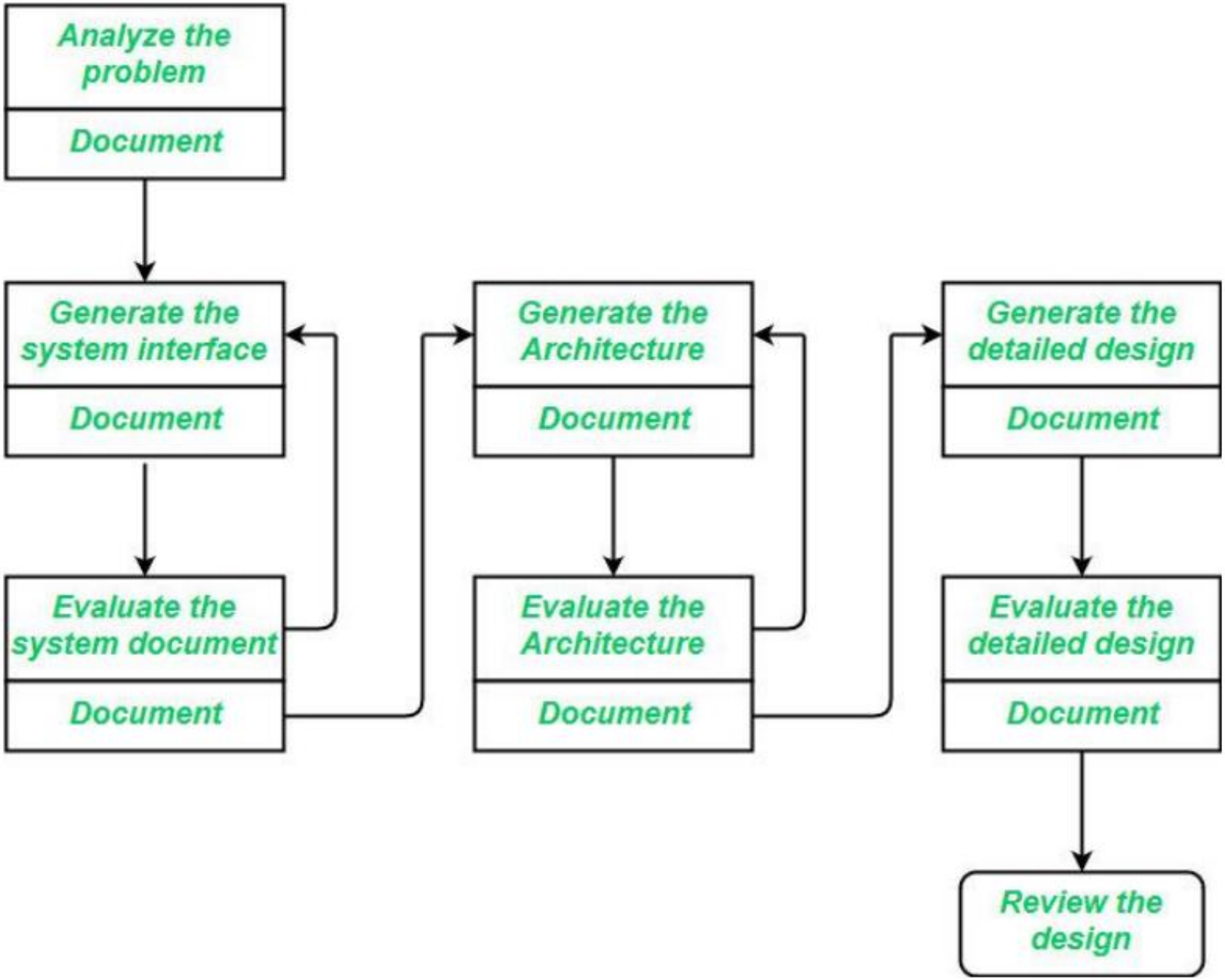
- Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module.
- Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

Algorithms required to implement the individual modules:

- Each function in a module usually performs some processing activity.
- The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.



Software Design Process is the phase where developers plan how to turn a set of requirements into a working system. Like a blueprint for the software. Instead of going straight into writing code, developers break down complex requirements into smaller, manageable pieces, design the system architecture, and decide how everything will fit together and work.


Analyze the problem
Document
Generate the system interface
Document
Evaluate the system document
Document
Generate the Architecture
Document
Evaluate the Architecture
Document
Generate the detailed design
Document
Evaluate the detailed design
Document
Review the design

Interface Design:

- Interface design is the specification of the interaction between a system and its environment.
- This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box.
- Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts.
- The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

Interface design should include the following details:

1.Precise description of events in the environment, or messages from agents to which the system must respond.

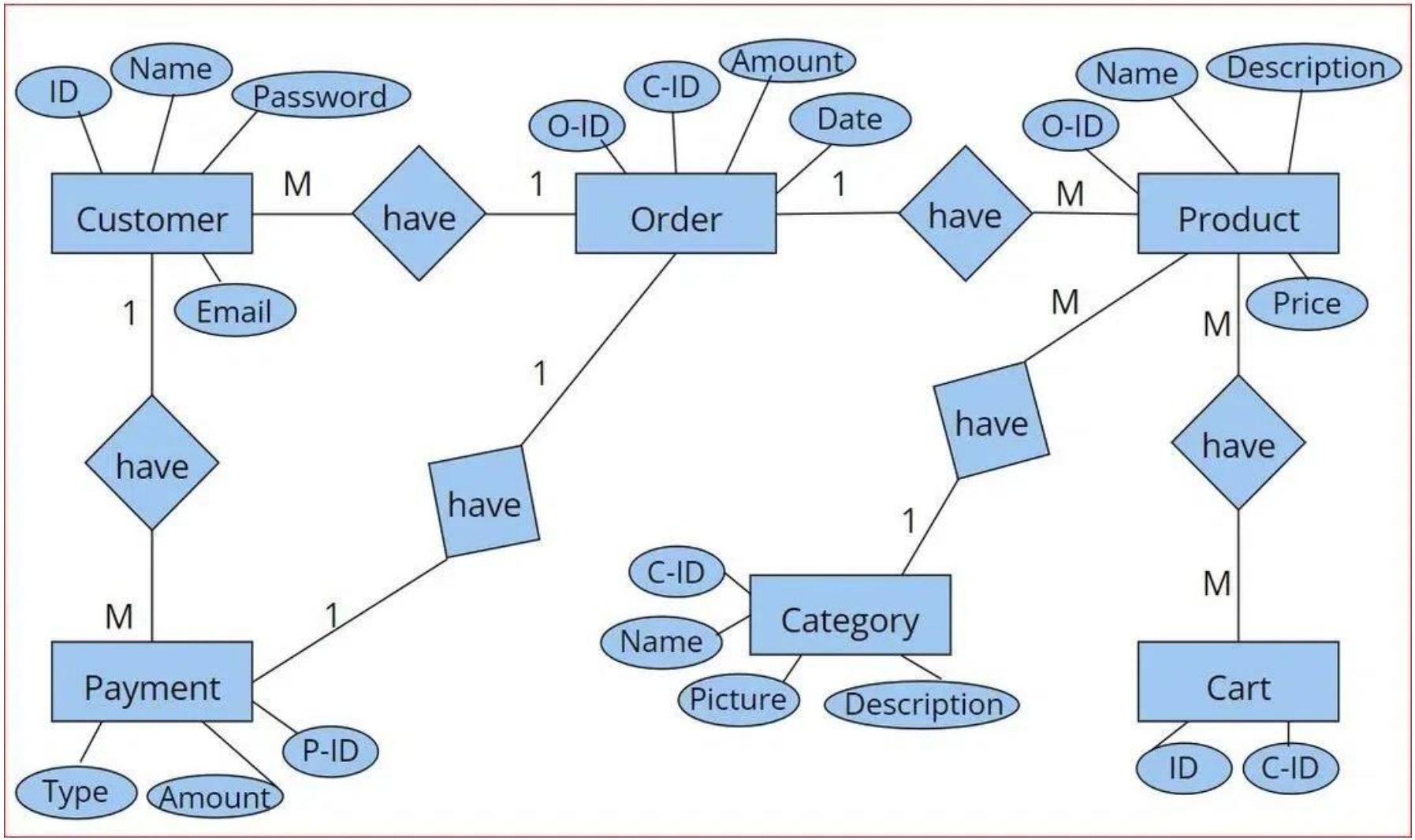
2.Precise description of the events or messages that the system must produce.

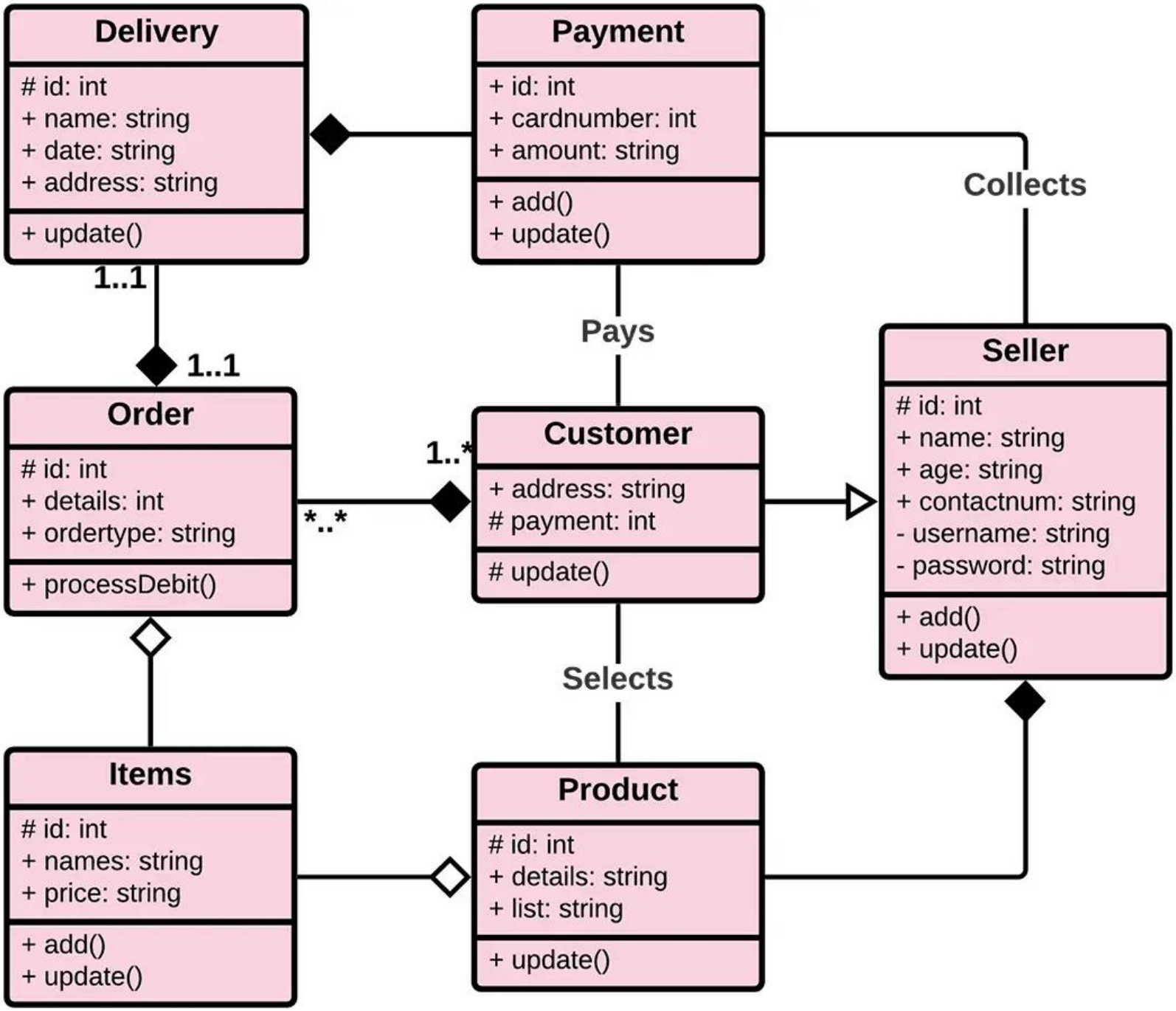
3.Specification of the data, and the formats of the data coming into and going out of the system.

4.Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

Architectural Design

- ✓ Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them.
- ✓ The outcome of high-level design is called the program structure or the *software architecture*.
- ✓ High-level design is a crucial step in the overall design of a software.
- ✓ When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy.
- ✓ Many different types of notations have been used to represent a high-level design.
- ✓ A notation that is widely being used for procedural development is a tree-like diagram called the structure chart.
- ✓ Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.


ID
Name
Password
Customer
M
have
1
Order
O-ID
C-ID
Amount
Date
1
have
M
Product
O-ID
Name
Description
Price
1
Email
1
have
M
have
M
have
1
have
1
M
Payment
1
P-ID
Type
Amount
C-ID
Name
Category
Picture
Description
M
Cart
ID
C-ID


Delivery
id: int
+ name: string
+ date: string
+ address: string
+ update()
Payment
+ id: int
+ cardnumber: int
+ amount: string
+ add()
+ update()
Collects
1..1
1..1
Pays
Seller
id: int
+ name: string
+ age: string
+ contactnum: string
- username: string
- password: string
+ add()
+ update()
Order
id: int
+ details: int
+ ordertype: string
+ processDebit()
1..*
..
Customer
+ address: string
# payment: int
# update()
Selects
Items
id: int
+ names: string
+ price: string
+ add()
+ update()
Product
id: int
+ details: string
+ list: string
+ update()

Detailed Design

The outcome of the detailed design stage is usually documented in the form of a *module specification* (MSPEC) document.

Detailed design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.

The detailed design may include:

1. Decomposition of major system components into program units.
2. Allocation of functional responsibilities to units.
3. User interfaces.
4. Unit states and state changes.
5. Data and control interaction between units.
6. Data packaging and implementation, including issues of scope and visibility of program elements.
7. Algorithms and data structures.

Analysis *versus* design

- Analysis and design activities differ in goal and scope.
- The analysis results are generic and does not consider implementation or the issues associated with specific platforms.
- The analysis model is usually documented using some graphical formalism.
- In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart.
- On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using unified modelling language (UML).
- The analysis model would normally be very difficult to implement using a programming language.
- The design model is obtained from the analysis model through transformations over a series of steps.
- In contrast to the analysis model, the design model reflects several decisions taken regarding the exact way system is to be implemented.
- The design model should be detailed enough to be easily implementable using a programming language.

HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

Most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess.

These characteristics are listed below:

Correctness: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

Efficiency: A good design solution should adequately address resource, time, and cost optimisation issues.

Maintainability: A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

COHESION AND COUPLING

- Effective problem decomposition is an important characteristic of a good design.
- Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other.

Coupling:

Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:

- If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
- If the interactions occur through some shared data, then also we say that they are highly coupled.

If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

Cohesion:

Cohesion is a measure of the *functional strength* of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

Classification of Cohesiveness

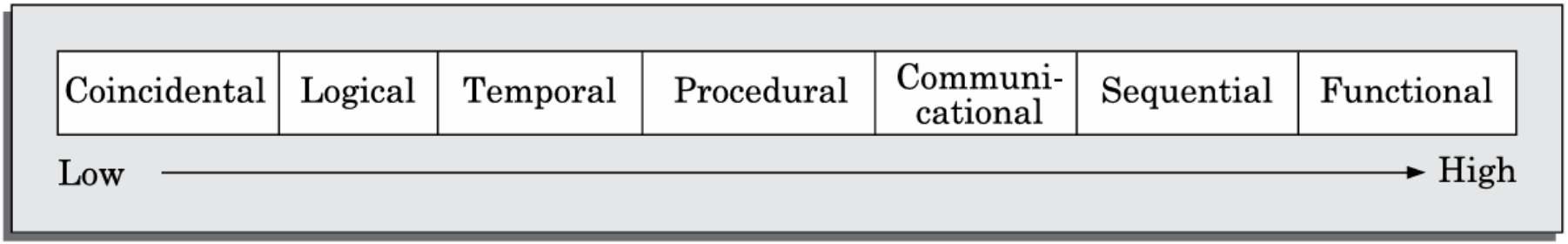


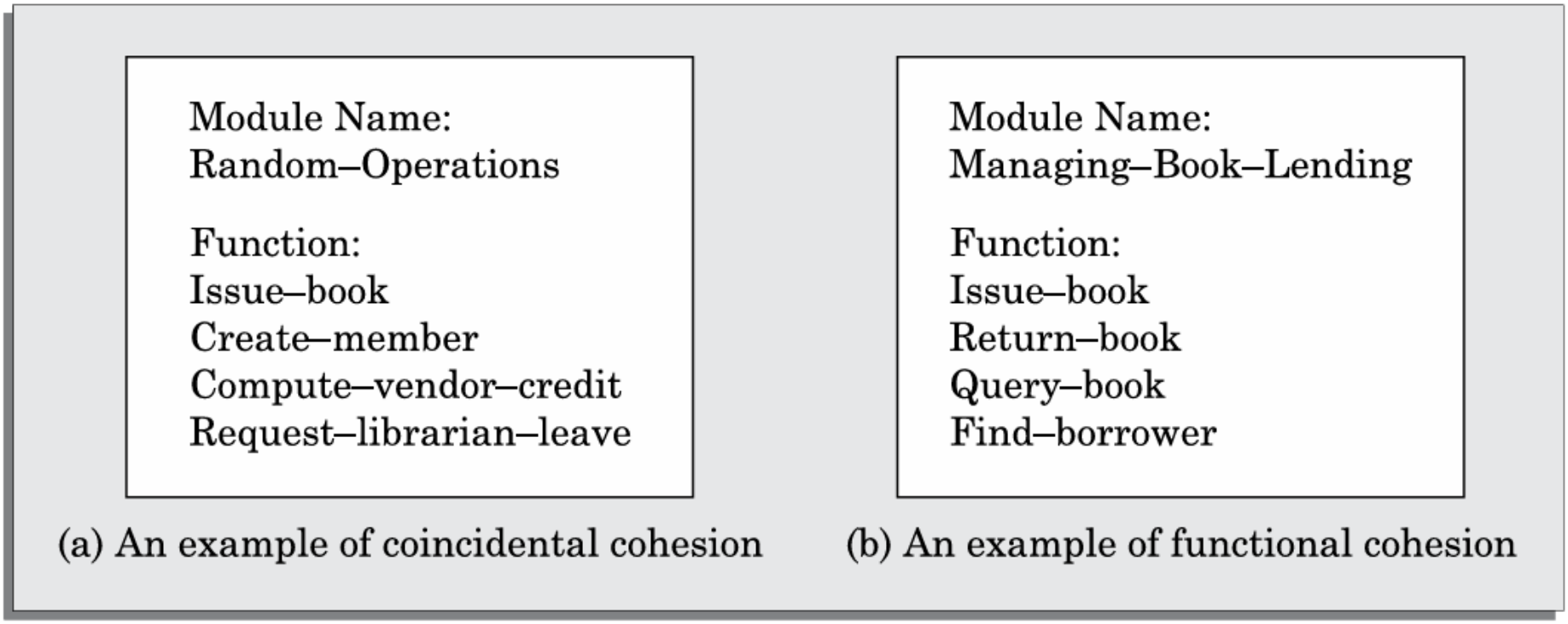
FIGURE 5.3 Classification of cohesion.

- ✓ Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective.
- ✓ The different modules of a design can possess different degrees of freedom.
- ✓ The cohesiveness increases from coincidental to functional cohesion.
- ✓ That is, coincidental is the worst type of cohesion and functional is the best cohesion possible.

Coincidental cohesion:

A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely.

It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design.



(a) An example of coincidental cohesion

(b) An example of functional cohesion

FIGURE 5.4 Examples of cohesion.

Logical cohesion:

A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc.

Example: a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

Temporal cohesion:

When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.

Example: When a computer is booted, several functions need to be performed. These include initialisation of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.

Procedural cohesion:

A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.

Example: The functions login(), place-order(), check-order(), print-bill(), place-order-on-vendor(), update inventory(), and logout() all do different thing and operate on different data.

Communicational cohesion:

A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.

Example: consider a module named *student* in which the different functions in the module such as ***admitStudent, enterMarks, printGradeSheet***, etc. access and manipulate data stored in an array named ***studentRecords*** defined within the module.

Sequential cohesion:

A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.

Example: In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions ***create-order(), check-item-availability(), place-order-on-vendor()*** are placed in a single module, then the module would exhibit sequential cohesion.

Functional cohesion:

A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task.

Example: a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (***e.g.,*** ***computeOvertime(), computeWorkHours(), computeDeductions()***, etc.) work together to generate the payslips of the employees.

Classification of Coupling

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled.

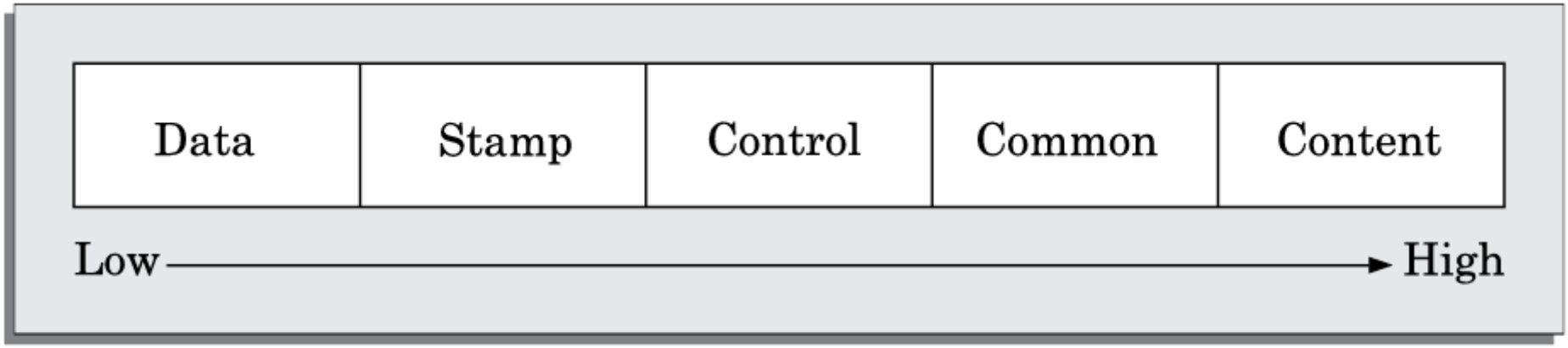


FIGURE 5.5 Classification of coupling.

Data coupling: Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share some global data items.

Content coupling: Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

Note:

High coupling among modules not only makes a design solution difficult to understand and maintain, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

LAYERED ARRANGEMENT OF MODULES

- ✓ The control hierarchy represents the organisation of program components in terms of their call relationship.
- ✓ The control hierarchy of a design is determined by the order in which different modules call each other.
- ✓ Many different types of notations have been used to represent the control hierarchy.
- ✓ The most common notation is a tree like diagram known as a structure chart.
- ✓ In a layered design solution, the modules are arranged into several layers based on their call relationships.
- ✓ A module is allowed to call only the modules that are at a lower layer.
- ✓ A module should not call a module that is either at a higher layer or even in the same layer.

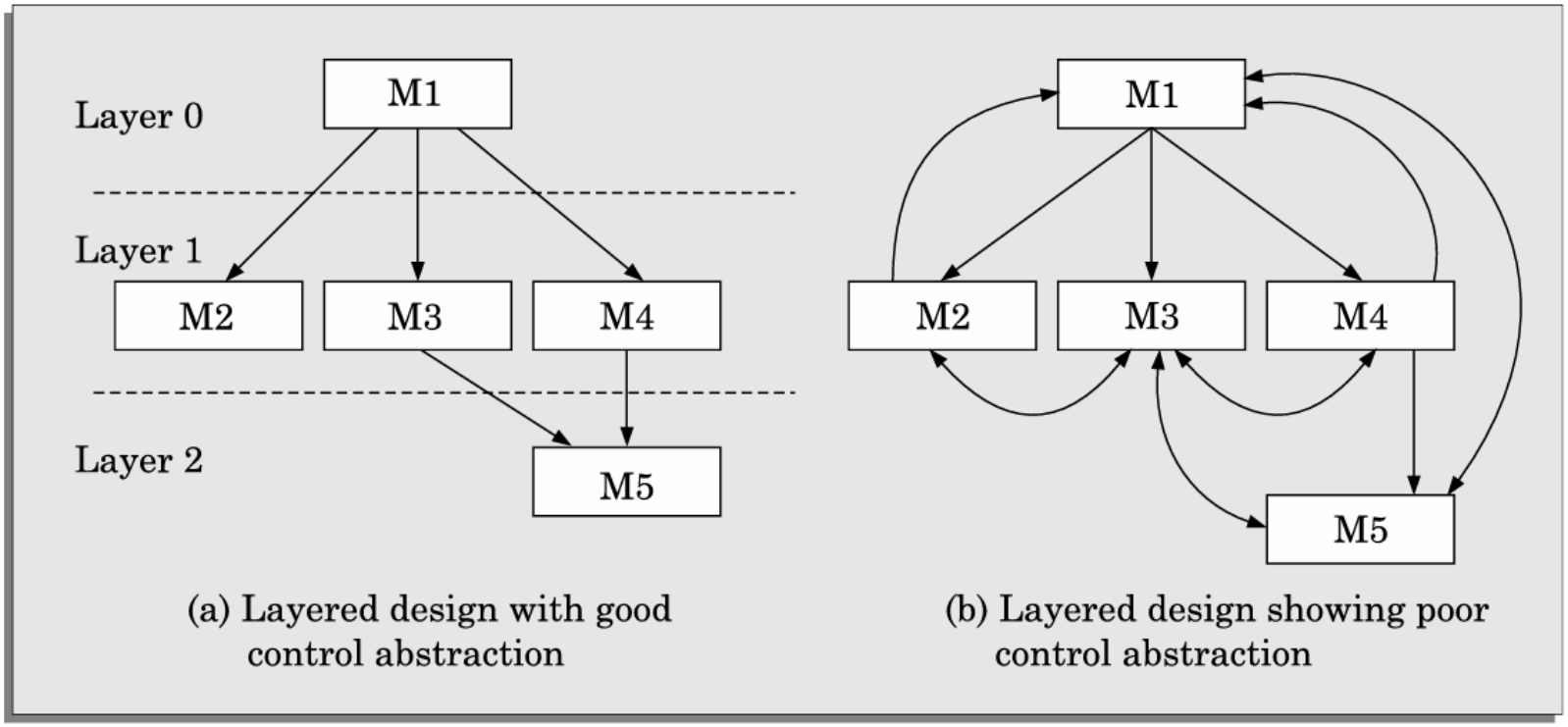


FIGURE 5.6 Examples of good and poor control abstraction.

Some important concepts and terminologies associated with a layered design:

Superordinate and subordinate modules: In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.

Visibility: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.

Control abstraction: In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

Depth and width: Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.

Fan-in: Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

APPROACHES TO SOFTWARE DESIGN

- There are two fundamentally different approaches to software design that are in use today—function-oriented design, and object-oriented design.
- Though these two design approaches are radically different, they are complementary rather than competing techniques.

Function-oriented Design

- Function-oriented design is the result of focusing attention on the function of the program.
- This is based on the **stepwise refinement**.
- Stepwise refinement is based on the **iterative procedural decomposition**.
- Stepwise refinement is a **top-down strategy** where a program is refined as a hierarchy of increasing levels of detail.
- In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

Object-oriented Design

- In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e., entities).
- Each object is associated with a set of functions that are called its methods.
- Each object contains its own data and is responsible for managing it.
- The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object.
- The system state is decentralised since there is no globally shared data in the system and data is stored in each object.

Example: In a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world functions.	The basic abstractions are not the real-world functions but are the data abstraction where the real-world entities are represented.
Function	Functions are grouped together by which a higher-level function is obtained.	Function are grouped together on the basis of the data they operate since the classes are associated with their methods.
Execute	carried out using structured analysis and structured design i.e, data flow diagram	Carried out using UML
State information	In this approach the state information is often represented in a centralized shared memory.	In this approach the state information is not represented in a centralized memory but is implemented or distributed among the objects of the system.
Approach	It is a top-down approach.	It is a bottom-up approach.
Begins basis	Begins by considering the use case diagrams and the scenarios.	Begins by identifying objects and classes.
Decompose	In function-oriented design we decompose in function/procedure level.	We decompose in class level.
Use	This approach is mainly used for computation sensitive application.	This approach is mainly used for evolving system which mimics a business or business case.

FUNCTION-ORIENTED SOFTWARE DESIGN

- Function-oriented design techniques were proposed nearly four decades ago.
- These techniques are at the present time still very popular and are currently being used in many software development projects.
- These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software.

- We shall call the design technique as structured analysis/ structured design (SA/SD) methodology.
- This technique draws heavily from the design methodologies proposed by the following authors:
  - DeMarco and Yourdon [1978]
  - Constantine and Yourdon [1979]
  - Gane and Sarson [1979]
  - Hatley and Pirbhai [1987]
- The SA/SD technique can be used to perform the high-level design of a software.

OVERVIEW OF SA/SD METHODOLOGY

SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1. Observe the following from the figure:

- During structured analysis, the SRS document is transformed into a *data flow diagram* (DFD) model.
- During structured design, the DFD model is transformed into a structure chart.

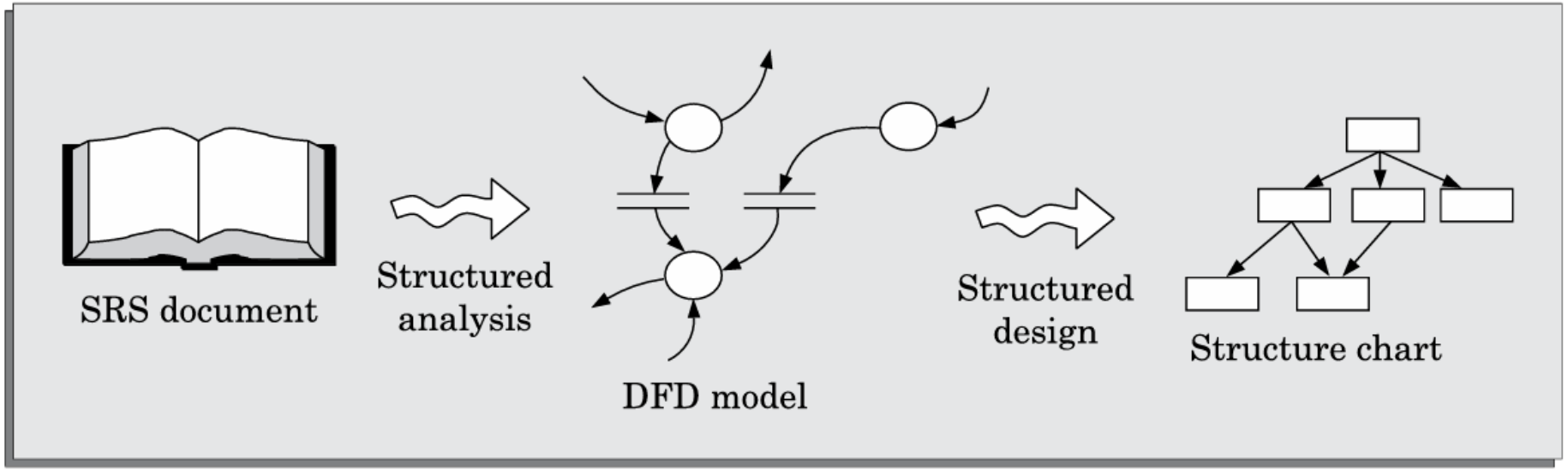


FIGURE 6.1 Structured analysis and structured design methodology.

- The structured analysis activity transforms the SRS document into a graphic model called the DFD model.
- During structured analysis, functional decomposition of the system is achieved.
- That is, each function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions.
- On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure.
- This module structure is also called the high-level design or the software architecture for the given problem. This is represented using a structure chart.
- The high-level design stage is normally followed by a detailed design stage.
- During the detailed design stage, the algorithms and data structures for the individual modules are designed.
- The detailed design can directly be implemented as a working system using a conventional programming language.

Data Flow Diagrams (DFDs)

- The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system.
- The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism—it is simple to understand and use.
- A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the data flow among these functions.

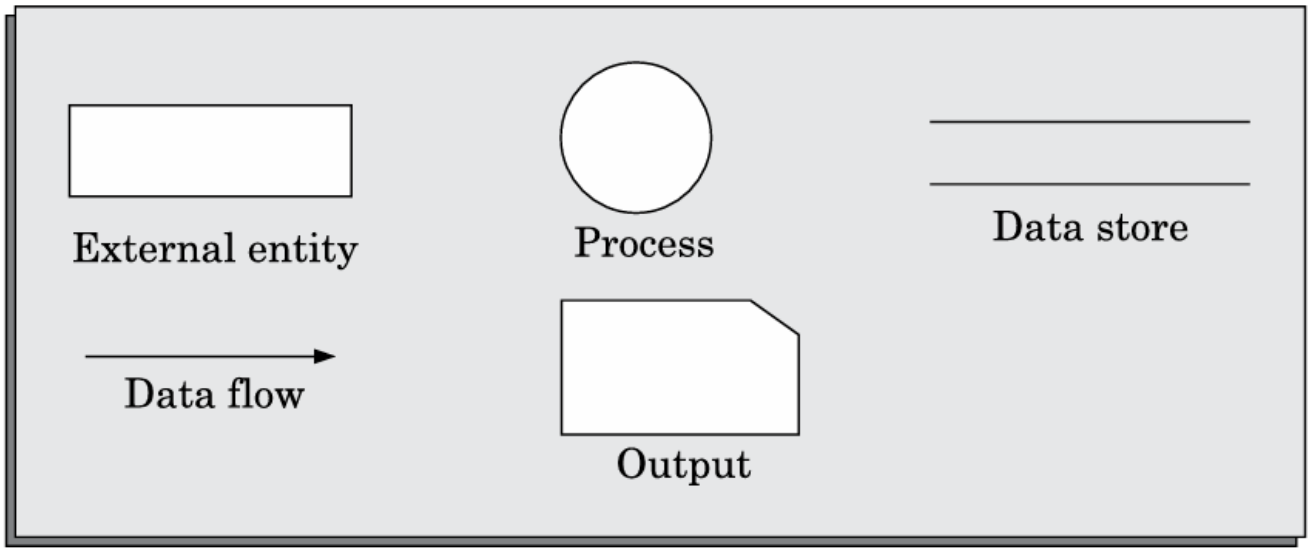


FIGURE 6.2 Symbols used for designing DFDs.

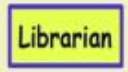
Function Symbol

- A function such as "search-book" is represented using a circle:
  - This symbol is called a **process** or **bubble** or **transform.**

  - Bubbles are annotated with corresponding function names.
  - A function represents some activity:
    - **Function names should be verbs.**

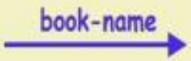
External Entity Symbol

- Represented by a rectangle
- External entities are either users or external systems:
  - input data to the system or
  - consume data produced by the system.
  - Sometimes external entities are called **terminator, source, or sink.**



Data Flow Symbol

- A directed arc or line.



  - Represents data flow in the direction of the arrow.
  - Data flow symbols are annotated with names of data they carry.

Data Store Symbol

- Represents a logical file:
  - A logical file can be:
    - a data structure
    - a physical file on disk.
  - Each data store is connected to a process:
    - By means of a data flow symbol.

book-details

Data Store Symbol

- Direction of data flow arrow:
  - Shows whether data is being read from or written into it.
- An arrow into or out of a data store:
  - Implicitly represents the entire data of the data store
  - Arrows connecting to a data store need not be annotated with any data name.

find-book

Books

A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.

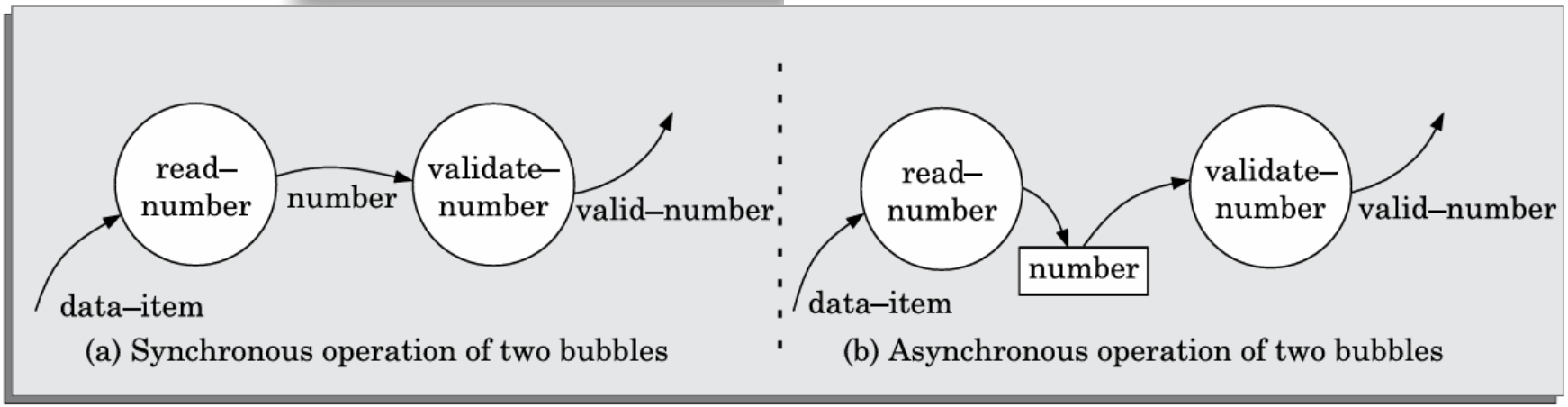


FIGURE 6.3 Synchronous and asynchronous data flow.

DEVELOPING THE DFD MODEL OF A SYSTEM

- ✓ A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
- ✓ The DFD model of a system is constructed by using a hierarchy of DFDs.
- ✓ The top level DFD is called the level 0 DFD or the context diagram.
- ✓ This is the most abstract (simplest) representation of the system (highest level).
- ✓ It is the easiest to draw and understand.
- ✓ At each successive lower level DFDs, more and more details are gradually introduced.
- ✓ To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.
- ✓ However, there is only a single data dictionary for the entire DFD model.
- ✓ All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

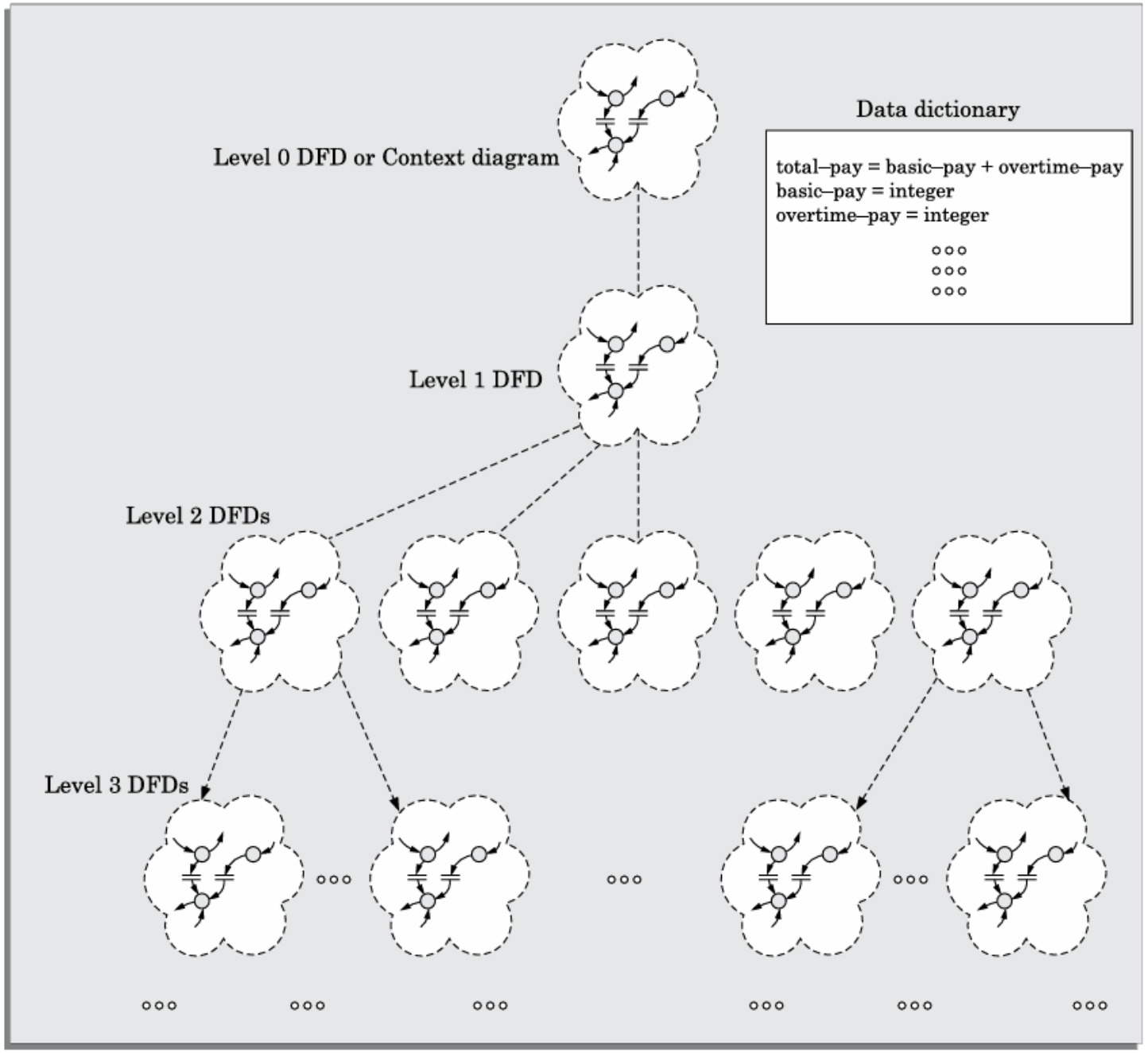


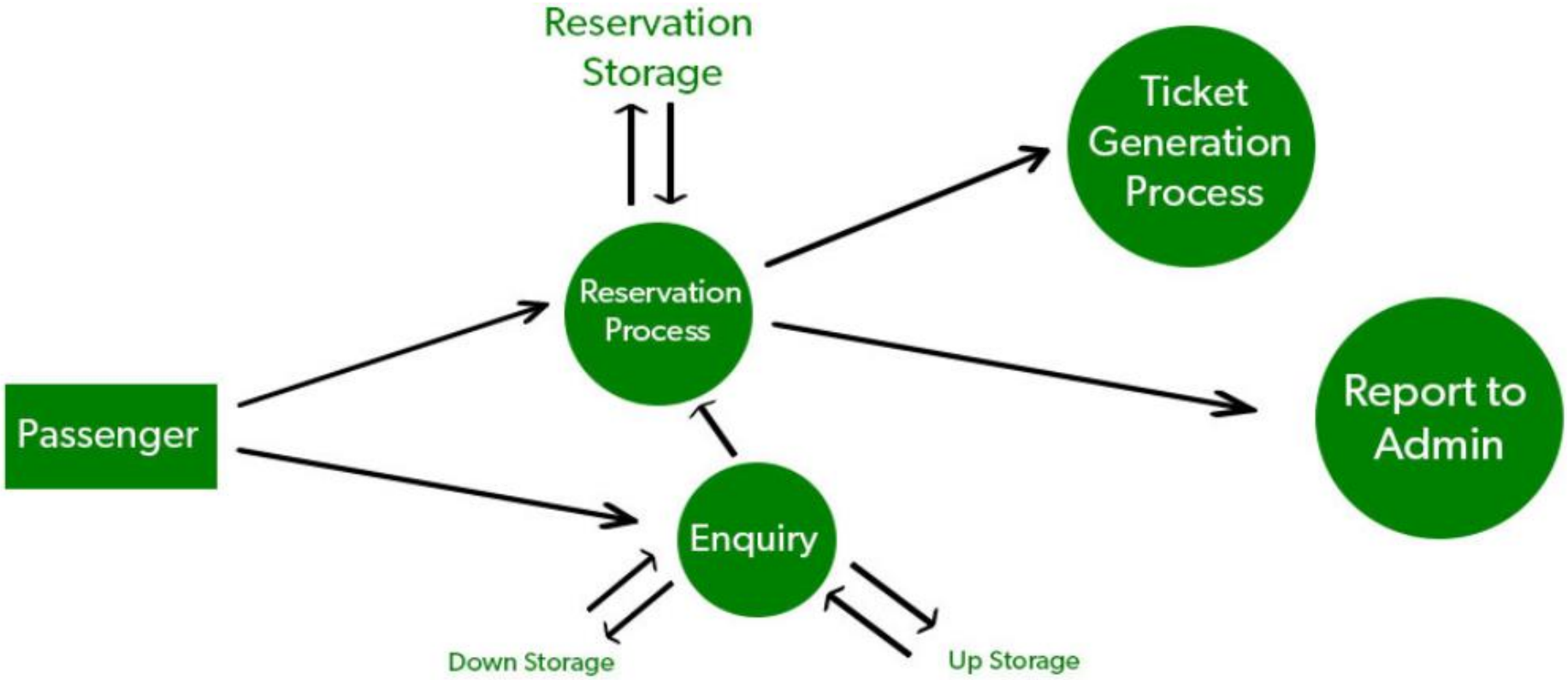
FIGURE 6.4 DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

Construction of context diagram: Examine the SRS document to determine:

- Different high-level functions that the system needs to perform.
- Data input to every high-level function.
- Data output from every high-level function.
- Interactions (data flow) among the identified high-level functions.

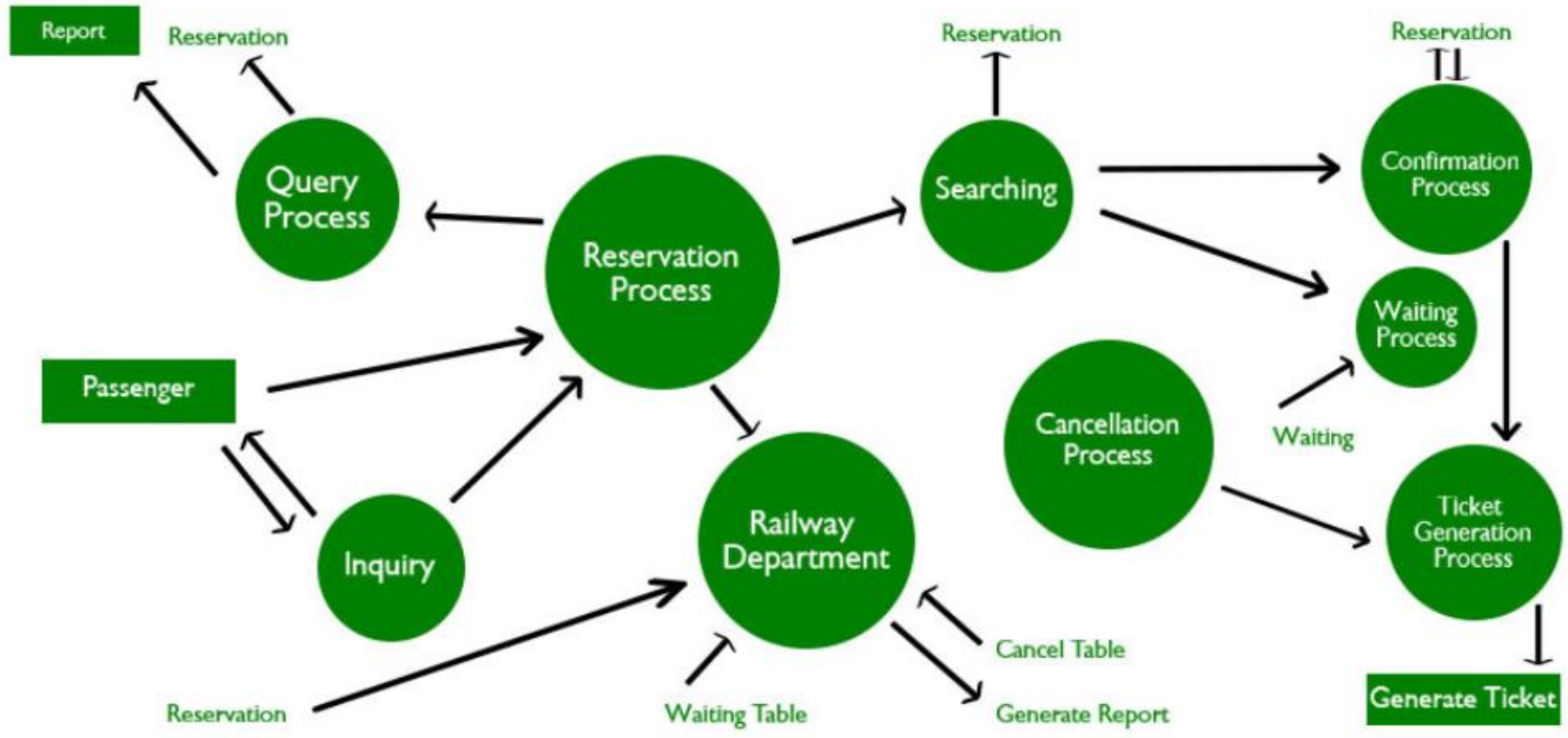
This would form the top-level data flow diagram (DFD), usually called the DFD 0.

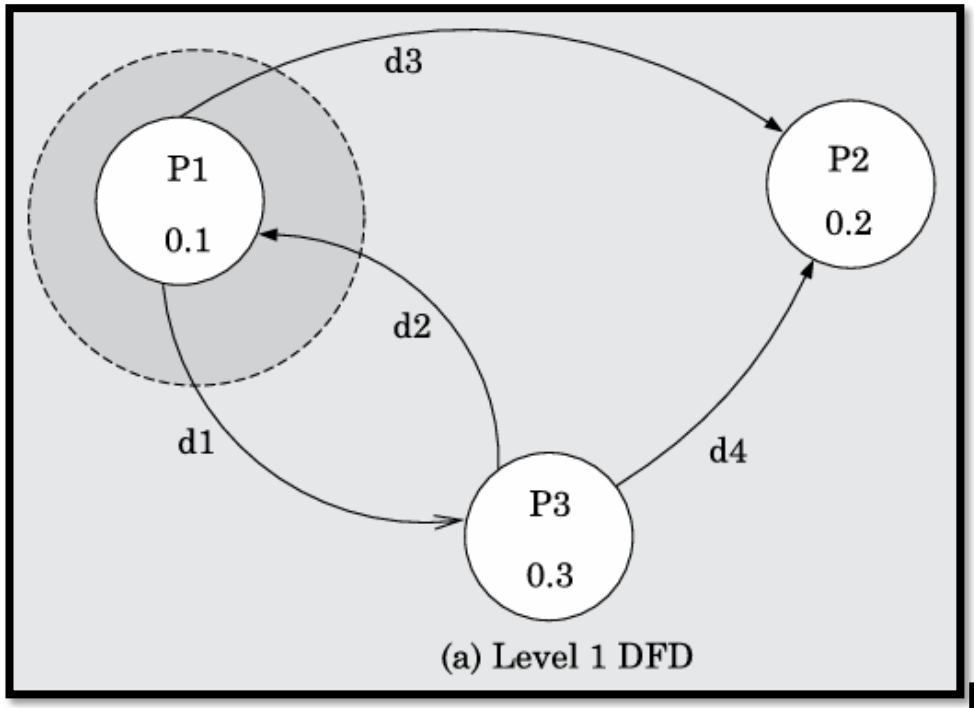
Construction of level 1 diagram: Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.



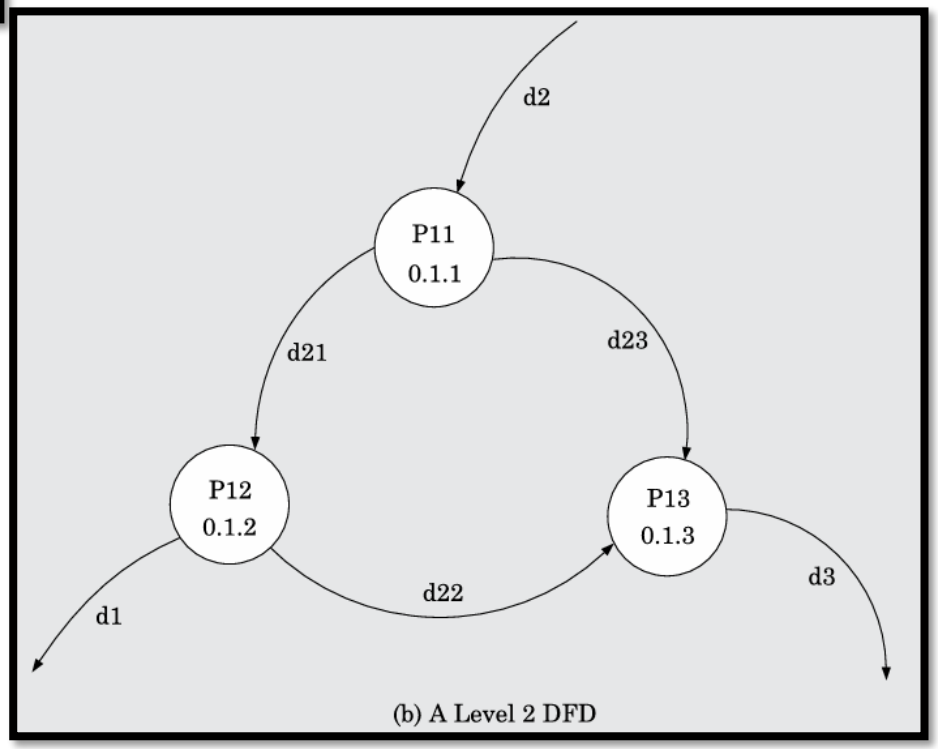
Construction of lower-level diagrams: Decompose each high-level function into its constituent subfunctions through the following set of activities:

- Identify the different subfunctions of the high-level function.
- Identify the data input to each of these subfunctions.
- Identify the data output from each of these subfunctions.
- Identify the interactions (data flow) among these subfunctions

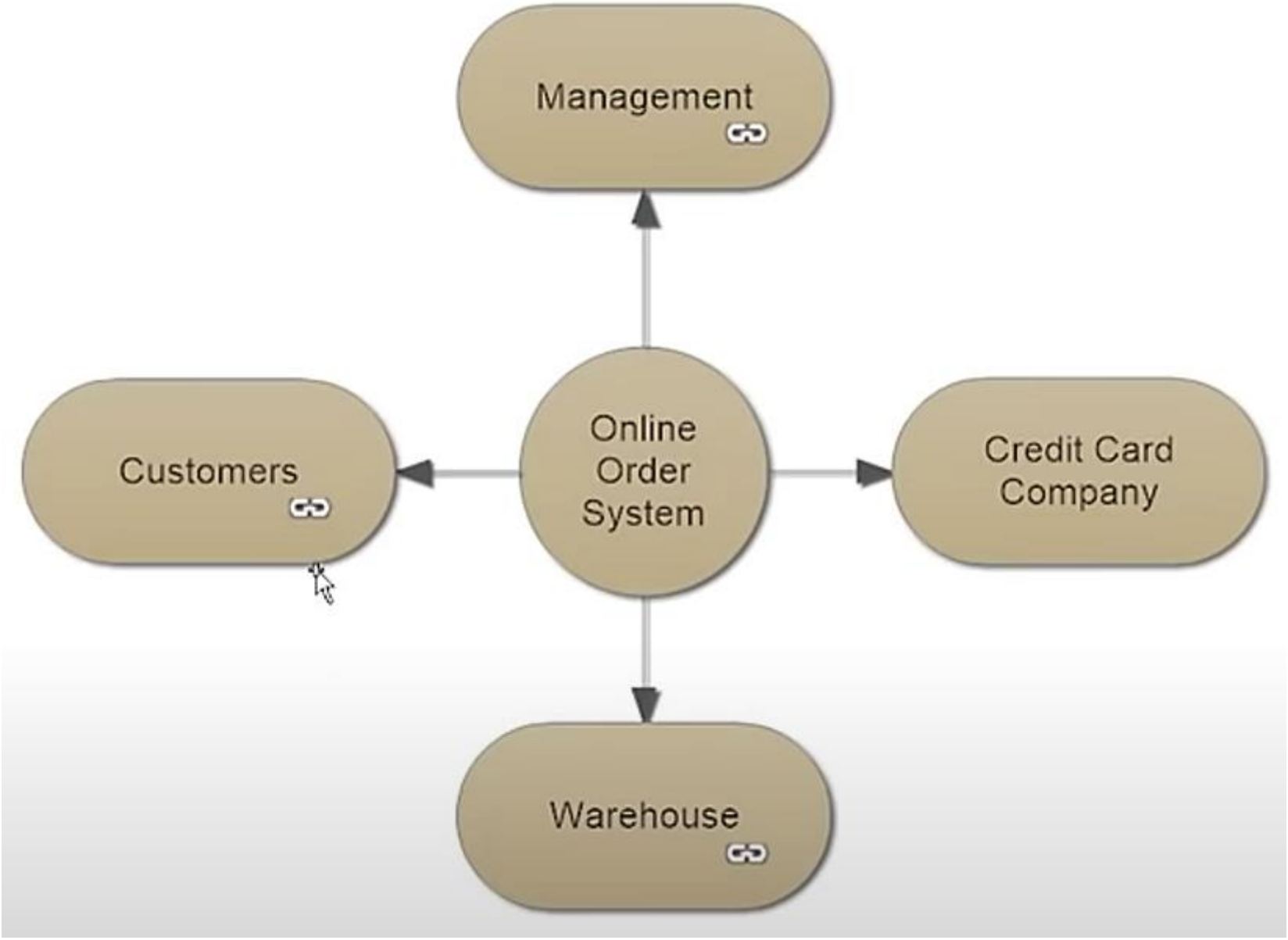


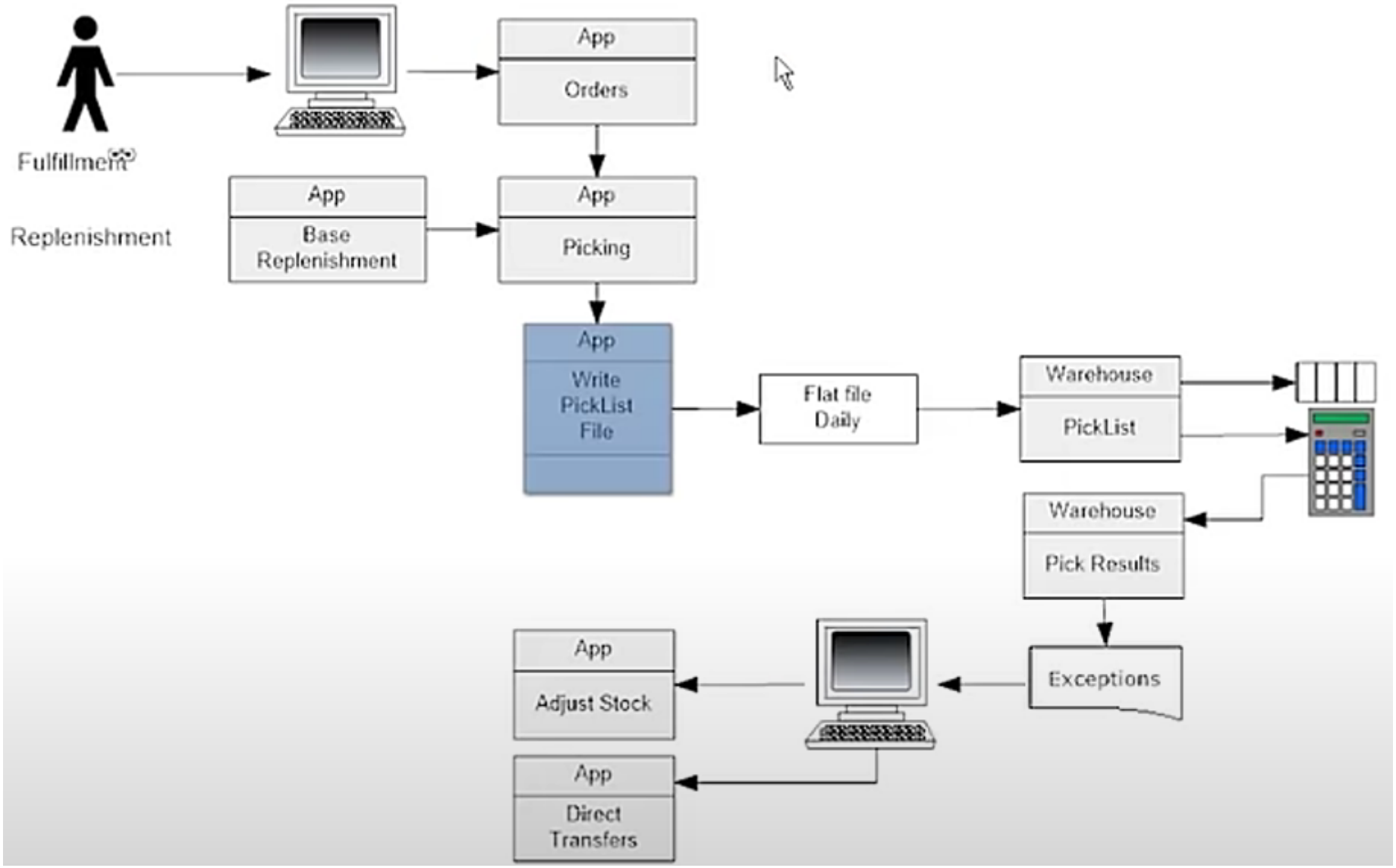


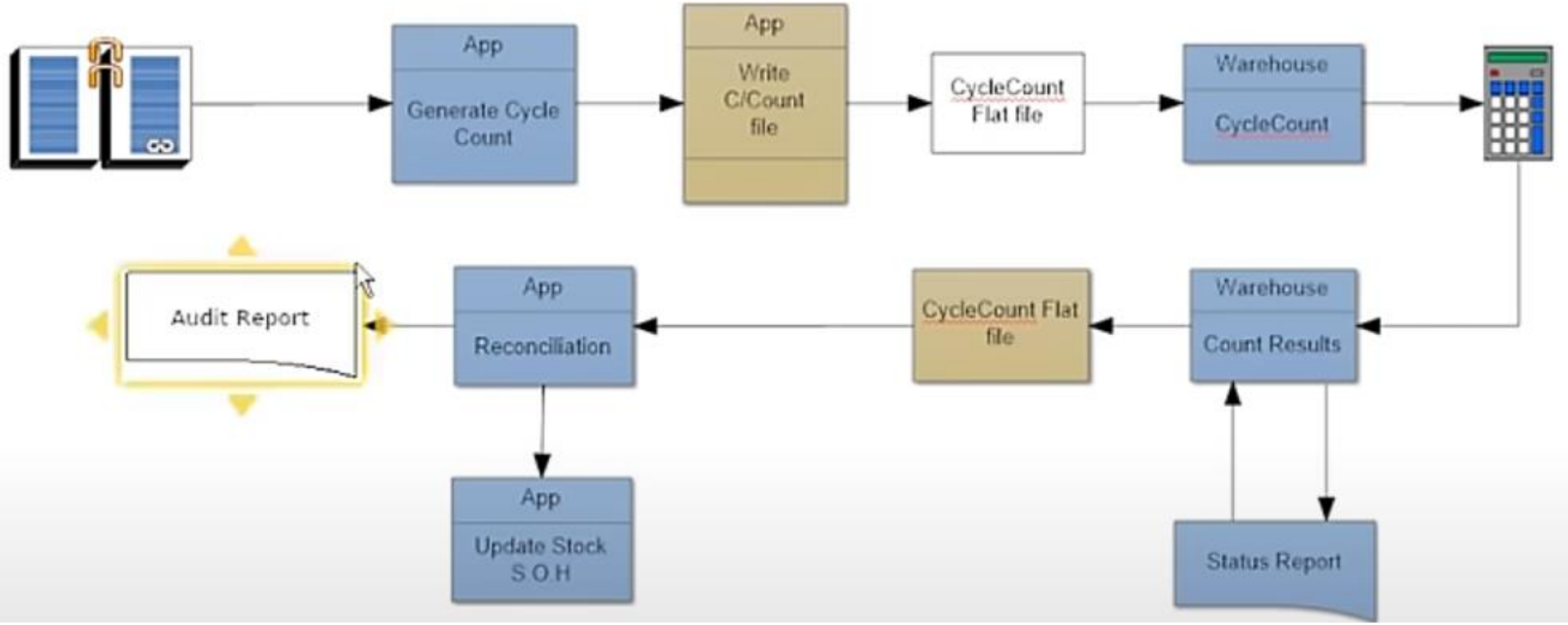
(a) Level 1 DFD



(b) A Level 2 DFD


Management
Customers
Online Order System
Credit Card Company
Warehouse


Fulfillment
Replenishment
App
Orders
App
Base Replenishment
App
Picking
App
Write PickList File
Flat file Daily
Warehouse
PickList
Warehouse
Pick Results
Exceptions
App
Adjust Stock
App
Direct Transfers


App
Generate Cycle Count
App
Write C/Count file
CycleCount Flat file
Warehouse
CycleCount
Audit Report
App
Reconciliation
CycleCount Flat file
Warehouse
Count Results
App
Update Stock S O H
Status Report

Data Flow Diagrams (DFDs):

•Show the flow of data through a system and how processes transform data. They represent a system's functional and data domains.

Structure Charts:

•Show the modular components of a system, along with the data and control flow between them, representing a program's structure.

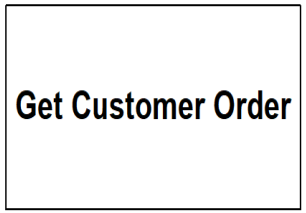
Structure Chart

- The major tool used in the structured design to depict the structure of a system is the **structure chart (SC)**

- There are three main components in the structure chart:
  - Modules
  - Connection between modules, and
  - Communication between modules

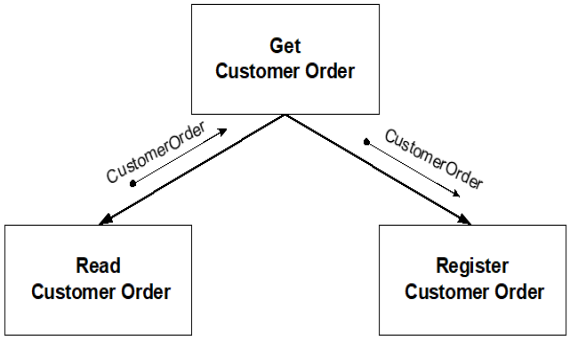
- **Rectangular box**
  - A rectangular box represents a module.
  - It is always annotated with the name of the module it represents.

Example



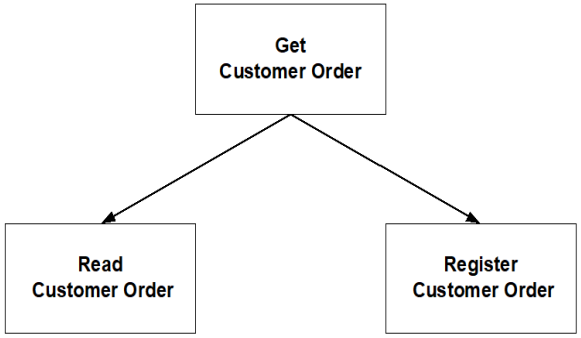
- **Data flow arrow**
  - To represent the data passes from one modulo to the other in the direction of arrow.
  - Small arrow appears alongside the modulo invocation arrow.
  - It is annotated with the corresponding data name.

Example:



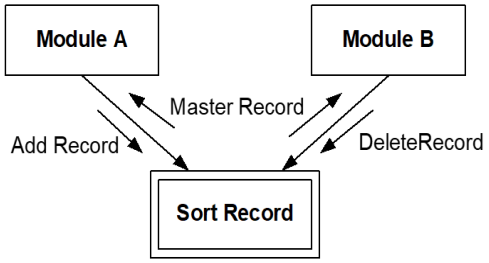
- **Module invocation arrow**
  - An arrow connecting two modules (rectangular boxes).
  - It represents the control passing from one module to another.

Example:



- **Library Modules**
  - To represent frequently called modules.
  - When a module is invoked by many other modules it is treated as a library module.
  - It is denoted by a rectangle with double edges.

Example



Transformation of a DFD Model into Structure Chart

- A systematic technique should be followed to transform the DFD representation of a problem into a structure chart.
- From structure analysis obtain the final version of DFD.
- Two strategies are know for transforming DFD into a structure chart:
  - Transform analysis
    - For a large classes of data when the same process is applied.
    - Usually at a higher level of DFD.
  - Transaction analysis
    - For a given data several possible paths are there in the DFD.
    - Usually at a lower level of DFD.

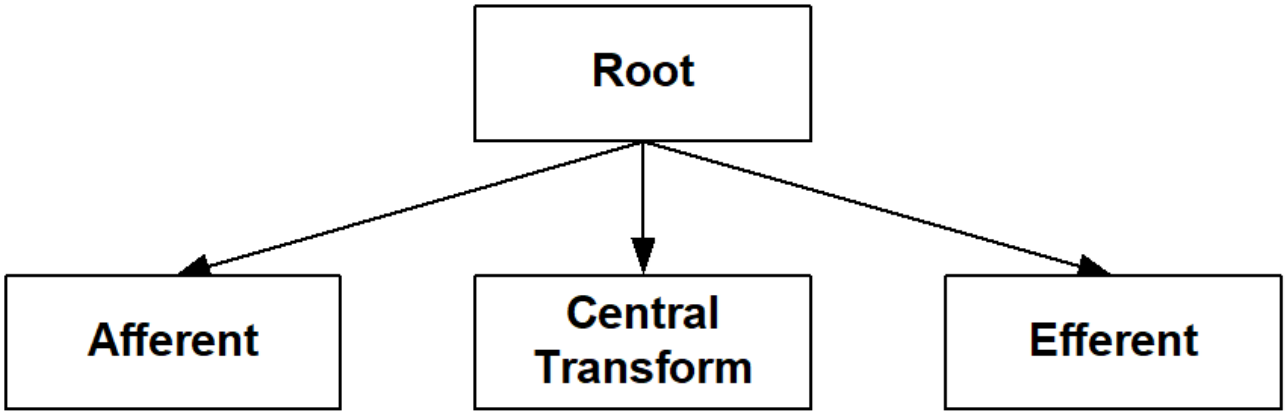
- Whether the transform analysis or transaction analysis is applicable it can be decided from the data input to the DFD
- Transform analysis is applicable when all the data flows into the diagram are processed in similar ways.
  - For example, a number of data are incident on the same bubble.
- Transaction analysis is applicable when the same data but process in different ways.
  - For example, same data goes to a number of bubbles

Transformation Analysis

- Transform analysis identifies the primary functional components and high level input and outputs for these components.
- There are three steps in the transformation analysis:
- **Step 1:**
  - Divide the DFD into three parts
    - Afferent branch
      - The input portion that transform input data from physical (e.g. reading from a source) to logical (storing into a table)
    - Efferent branch
      - The output portion that transform output data from logical form (e.g. output from a process) to physical form (e.g. display the result)
    - Central transform
      - The remaining portion of the DFD

- **Step 2:**
  - Derive the structure chart by drawing a module for each of the afferent branch, central transform, and the efferent branch
  - They are drawn below a module called the root module which invokes these modules.

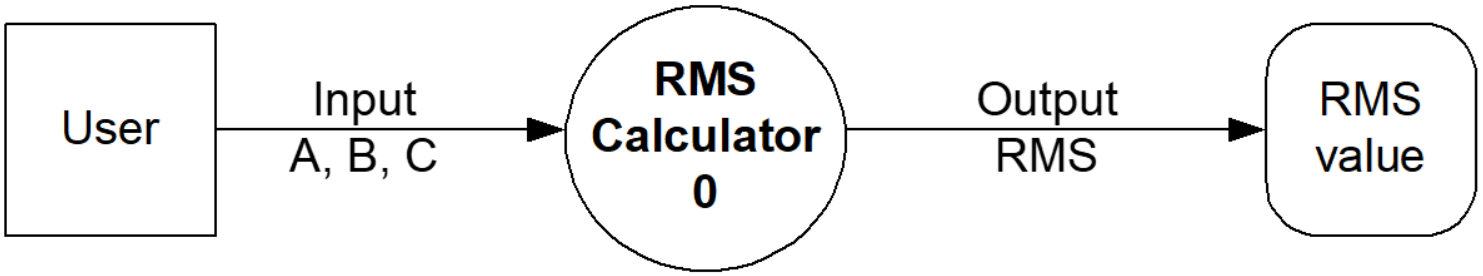
Example:

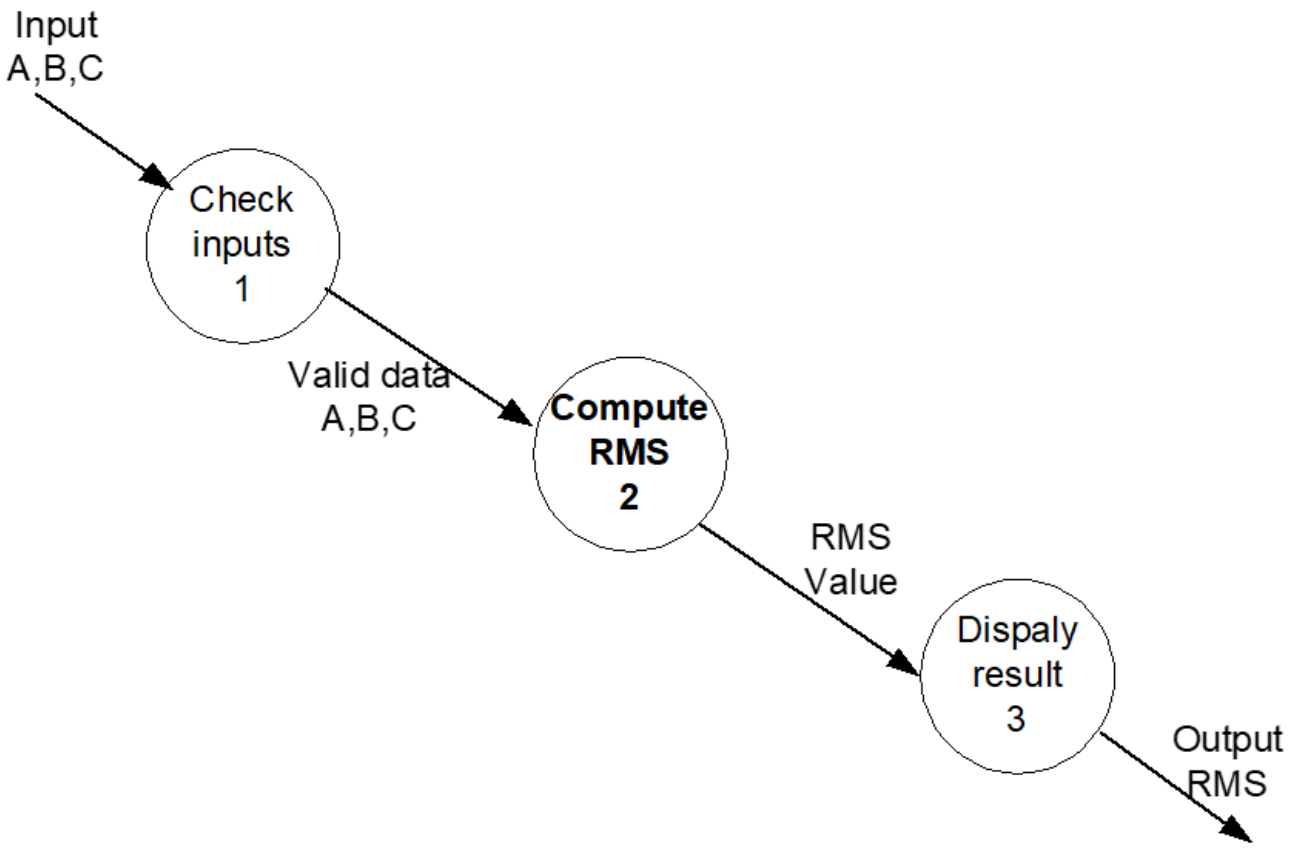


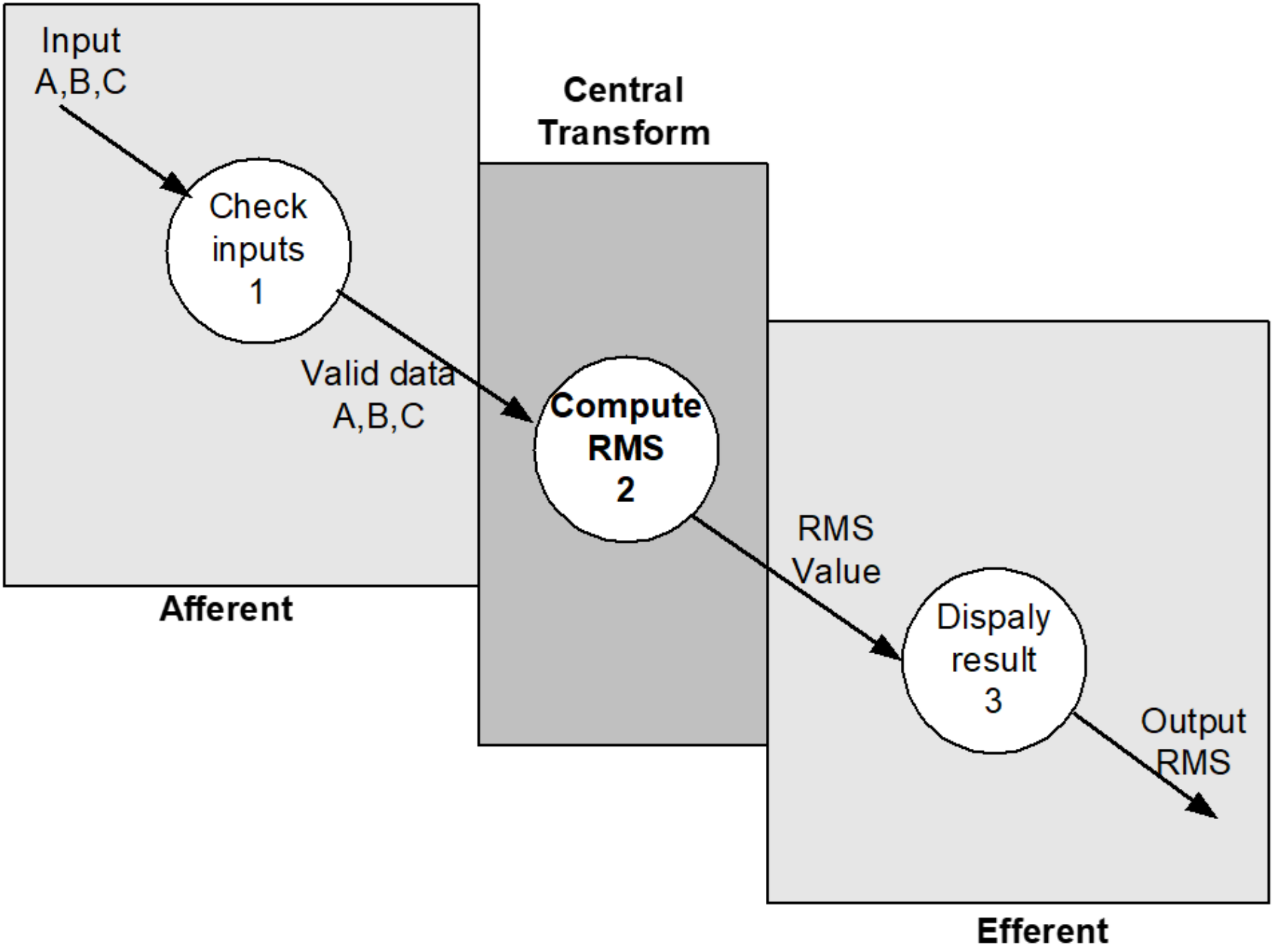
Transformation Analysis

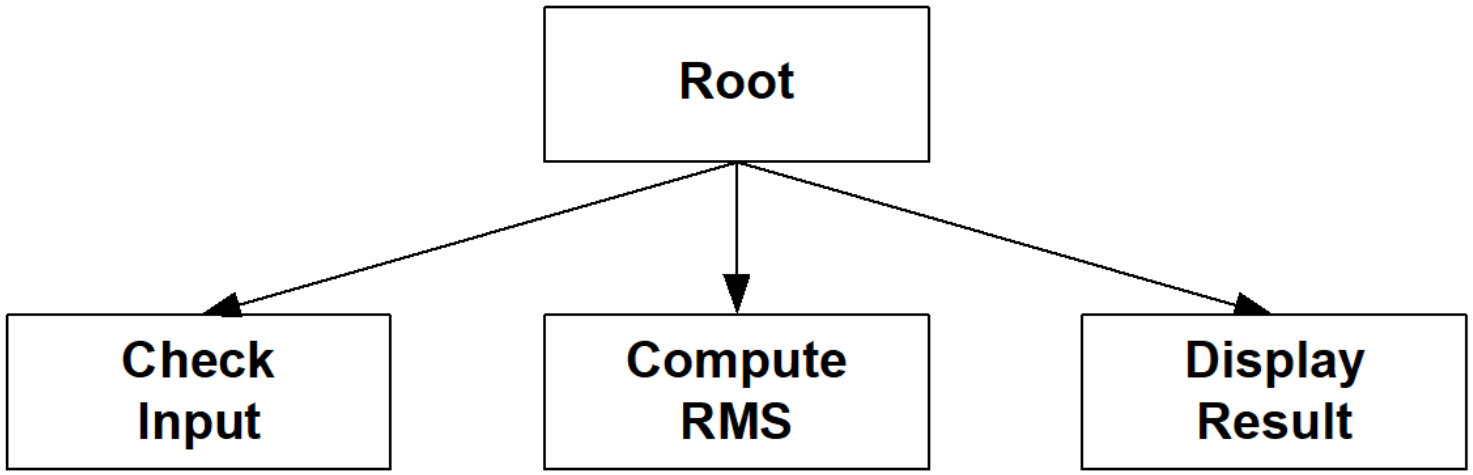
- Step 3:
  - Refine the structure chart by adding sub modules required by each of the high-level functional components.
  - Factoring
    - Process of breaking functional components into sub-components is called **factoring.**
    - The factoring process should be repeated until all bubbles in the DFD are represented in the structure chart.

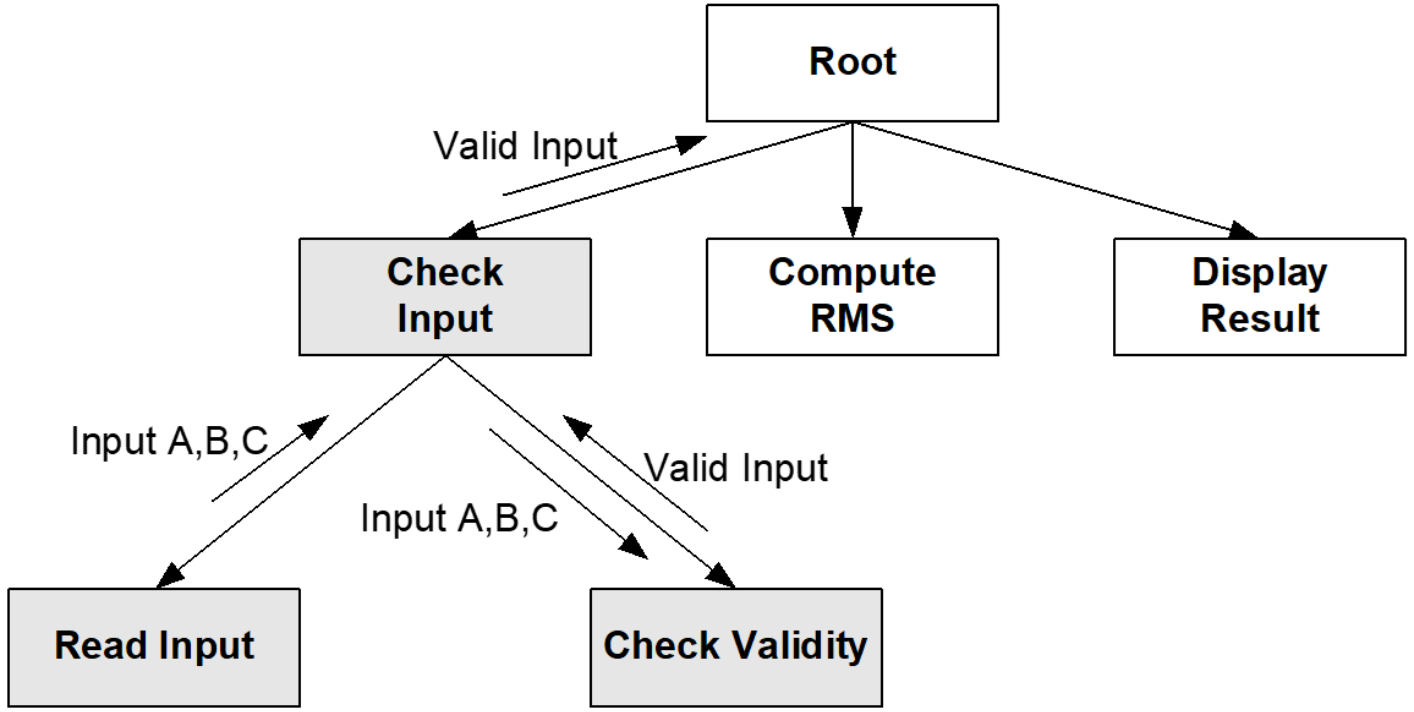
Transformation Analysis: An Example

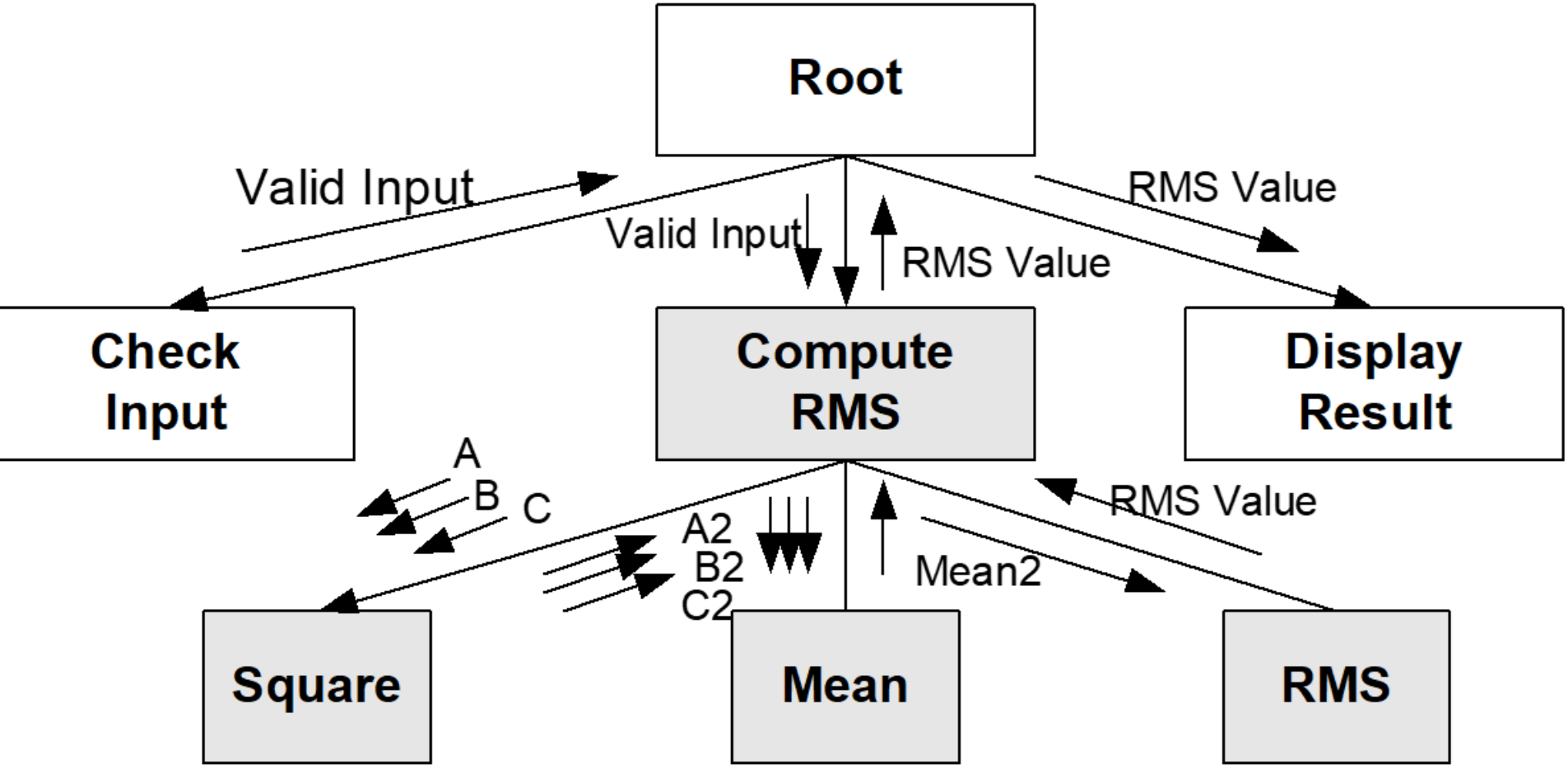





Input
A,B,C
Check
inputs
1
Valid data
A,B,C
Central
Transform
Compute
RMS
2
RMS
Value
Dispaly
result
3
Output
RMS
Afferent
Efferent


Root
Check Input
Compute RMS
Display Result


Root
Valid Input
Check Input
Compute RMS
Display Result
Input A,B,C
Input A,B,C
Valid Input
Read Input
Check Validity


Root
Valid Input
Valid Input
RMS Value
RMS Value
Check Input
Compute RMS
Display Result
A
B
C
A2
B2
C2
Mean2
RMS Value
Square
Mean
RMS

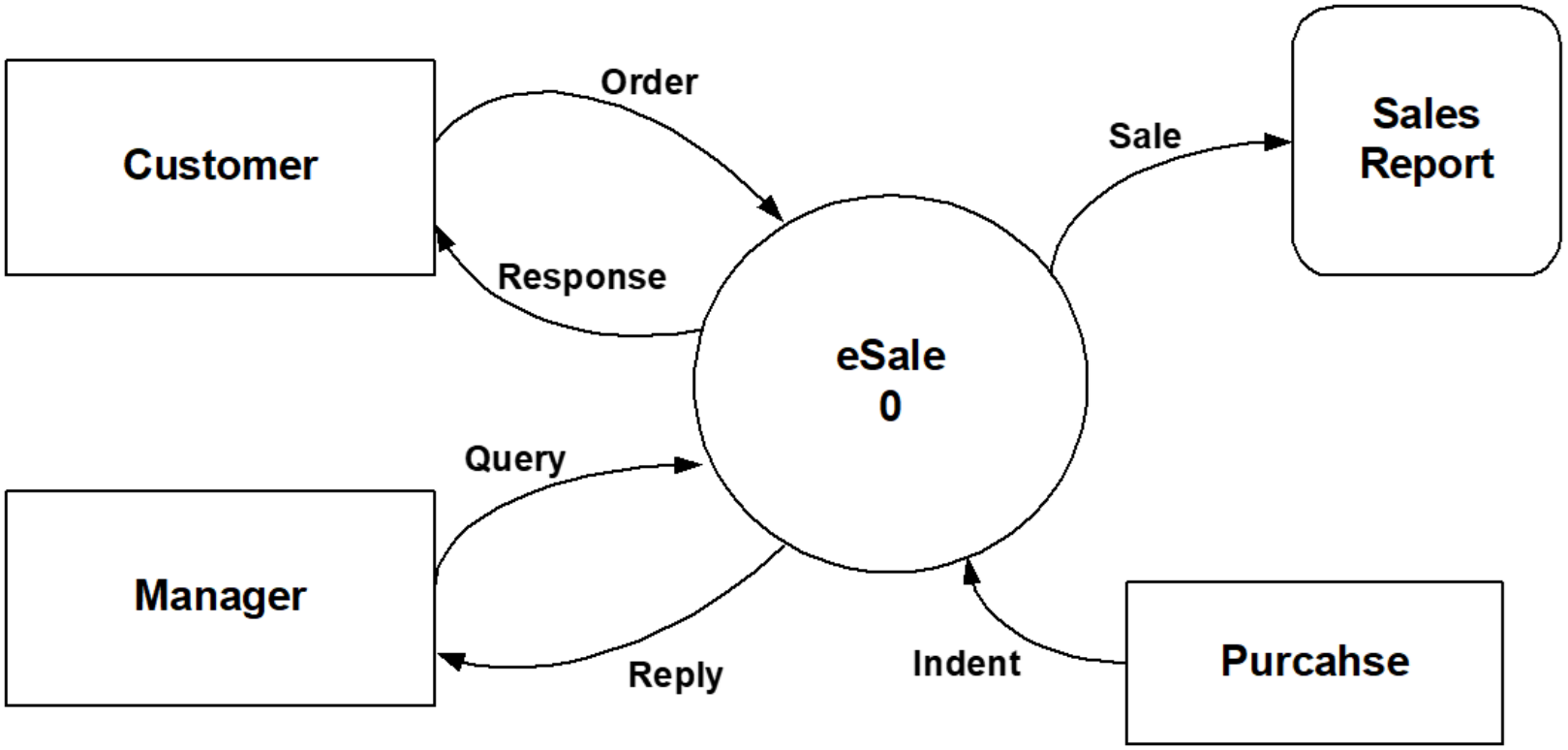
Transaction Analysis

- Transformation analysis is suitable for transforming central system, which is characterized by identical processing steps for each data items.
- Transaction analysis is suitable for transaction-driven system which is characterized by several possible paths are to be traversed through the DFD depending on the input data item.
- The number of bubbles on which the input data to the DFD are incident defines the number of transactions.
- However, some transactions may not require any input data.

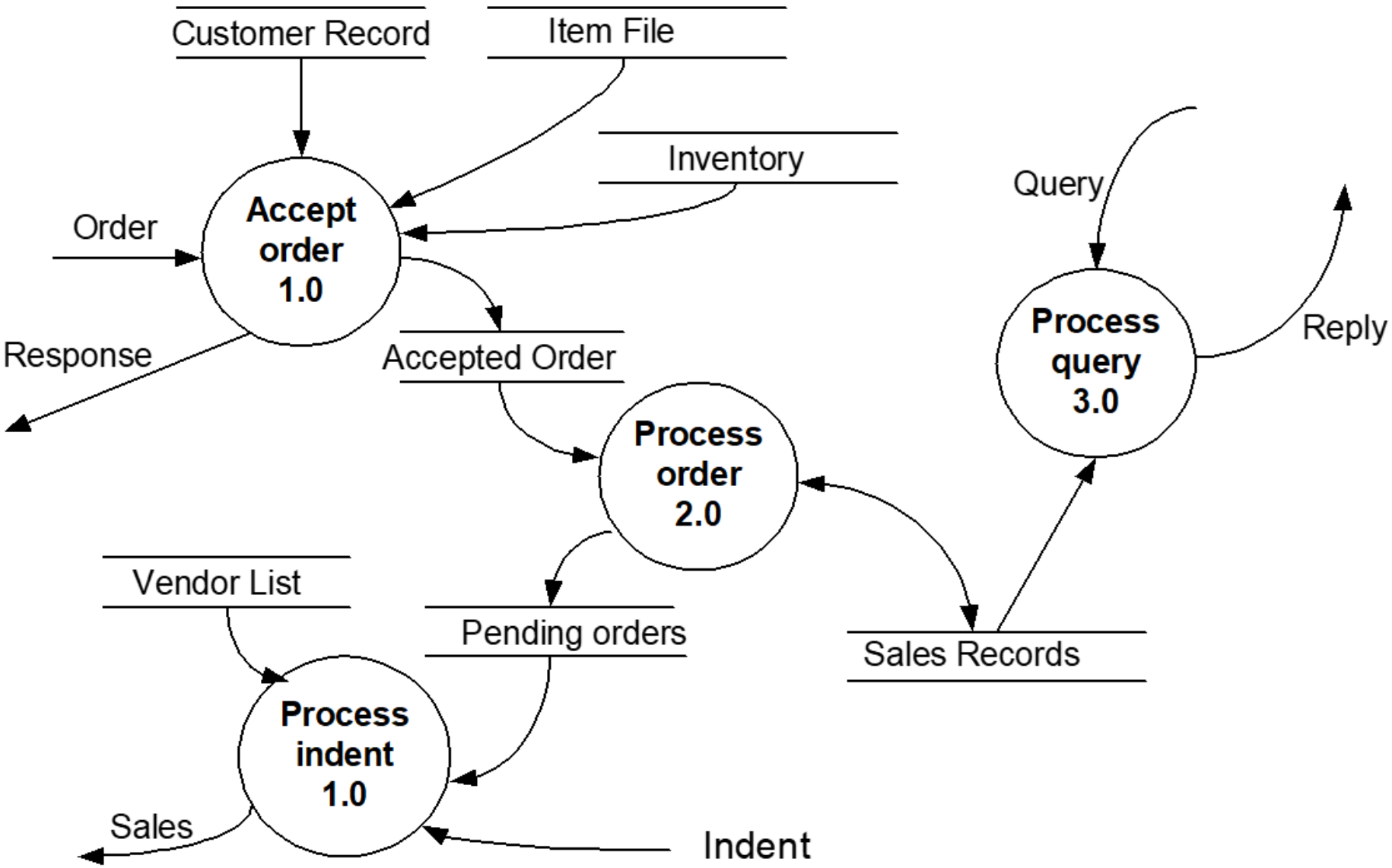
- Identify a transaction by tracing the path from an input data to output data.
- All traversed bubbles belong to the transaction.
- These bubbles is to be mapped to a module in the structure chart..
- There will be root module under which all modules identifying transactions will be drawn.

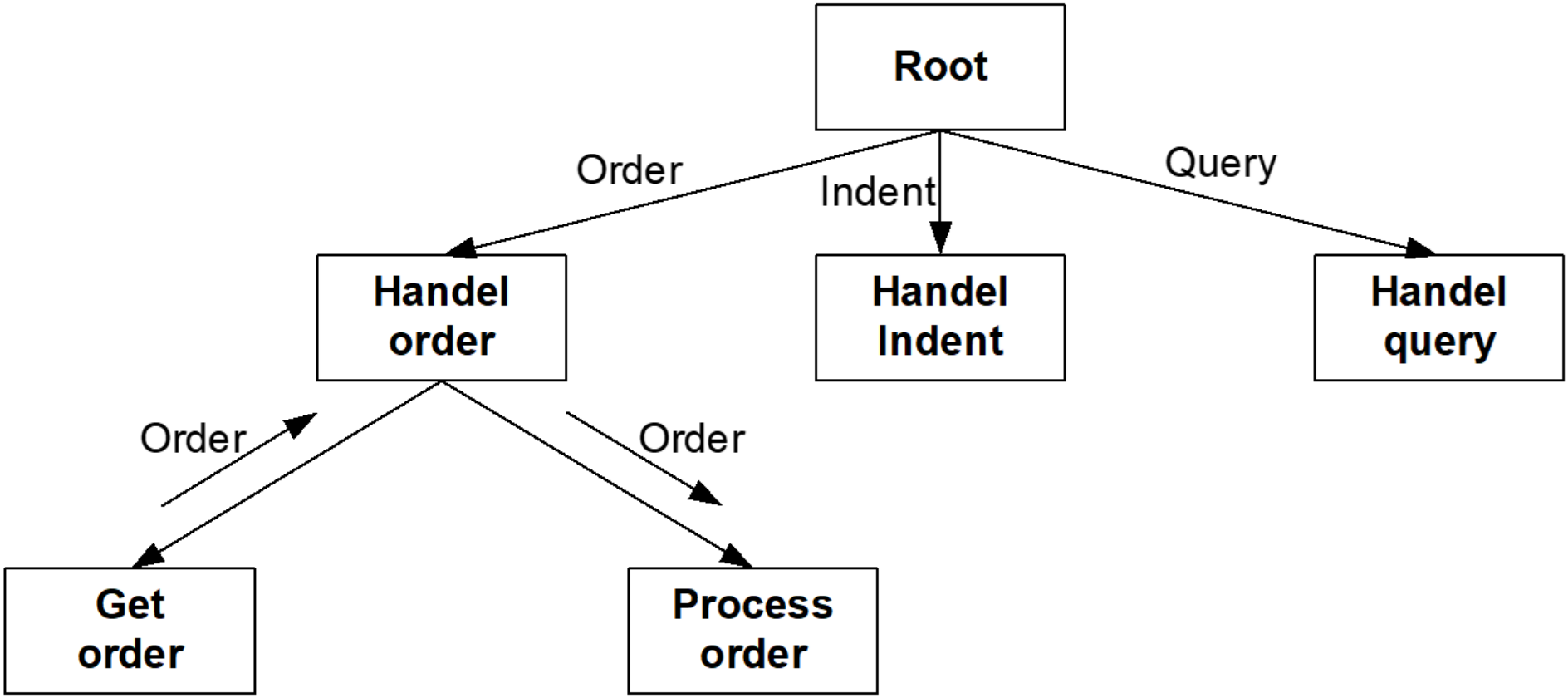
- Every transaction carries a label identifying its functionality.
- A transaction can be refined to its more detailed level (sub-modules).

An Example of Transaction Analysis: eSale



Context Diagram


Customer Record
Item File
Inventory
Query
Order
Accept order 1.0
Process query 3.0
Reply
Response
Accepted Order
Process order 2.0
Vendor List
Pending orders
Sales Records
Process indent 1.0
Sales
Indent


Root
Order
Indent
Query
Handel order
Handel Indent
Handel query
Order
Order
Get order
Process order

Structure Chart vs. Flow Chart

- SC – Identify different modules of the software.
- FC - Identify the control of execution of a program.

- SC - Represent data interchange among different modules.
- FC – Technique to represent the flow of control.

- SC – Not able to depict ordering of tasks.
- FC – Depicts the ordering of tasks.

Detailed Design:

- During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart.
- These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English.
- The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out.
- To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

USER INTERFACE DESIGN

User Interface Design

The user interface is the **front-end application** view to which the **user interacts** to use the software.

CHARACTERISTICS OF A GOOD USER INTERFACE

- Speed of learning
- Speed of use
- Speed of recall
- Error prevention
- Aesthetic and attractive
- Consistency
- Feedback
- Support for multiple skill levels

Speed of learning:

- A good user interface should be easy to learn.
- Speed of learning is hampered by complex syntax and semantics of the command issue procedures.
- A good user interface should not require its users to memorise commands.
- Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface.
- The following three issues are crucial to enhance the speed of learning:

Use of metaphors and intuitive command names:

- Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar with.
- The abstractions of real-life objects or concepts used in user interface design are called ***metaphors.***

Example:

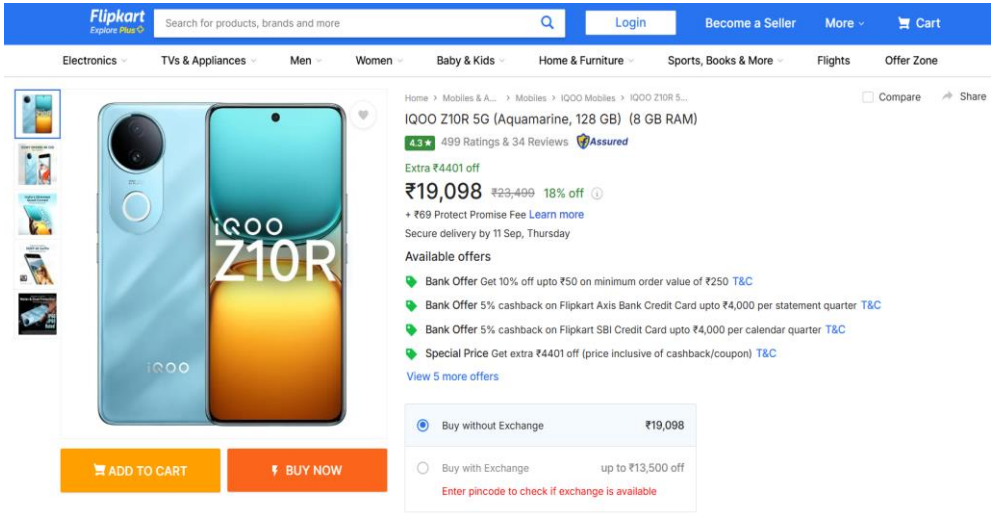
If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it.

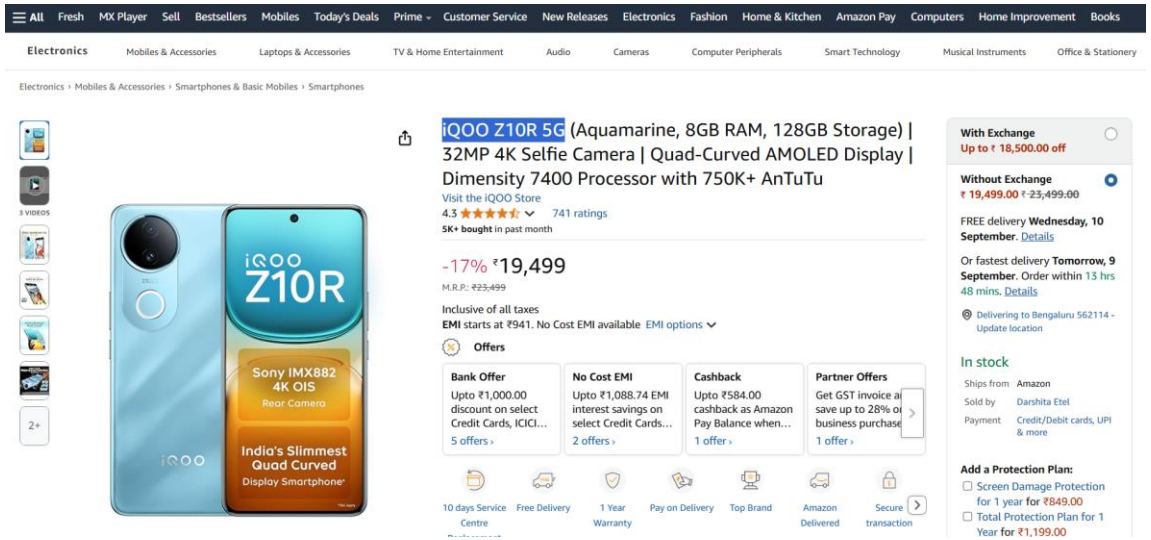
Consistency:

- Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions.
- This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts.
- Thus, the different commands supported by an interface should be consistent.

Component-based interface:

- Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with.
- This can be achieved if the interfaces of different applications are developed using some standard user interface components.





Speed of use:

- Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands.
- This characteristic of the interface is sometimes referred to as *productivity support* of the interface.
- It indicates how fast the users can perform their intended tasks.
- The time and user effort necessary to initiate and execute different commands should be minimal.

Speed of recall:

- Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximised.
- This characteristic is very important for intermittent users.
- Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

Error prevention: A good user interface should minimise the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by an average users while using the interface.

Aesthetic and attractive: A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

Consistency: The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalise the knowledge about aspects of the interface from one part to another.

Feedback: A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request.

Support for multiple skill levels: A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces.

Error recovery (undo facility): While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface.

User guidance and on-line help: Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software.

TYPES OF USER INTERFACES

Broadly speaking, user interfaces can be classified into the following three categories:

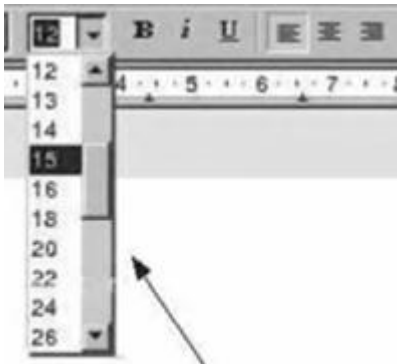
- Command language-based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

Command Language-based Interface

- A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands.
- The user is expected to frame the appropriate commands in the language and type them appropriately whenever required.
- A simple command language-based interface might simply assign unique names to the different commands.
- One can systematically develop a command language interface by using the standard compiler writing tools Lex and Yacc.

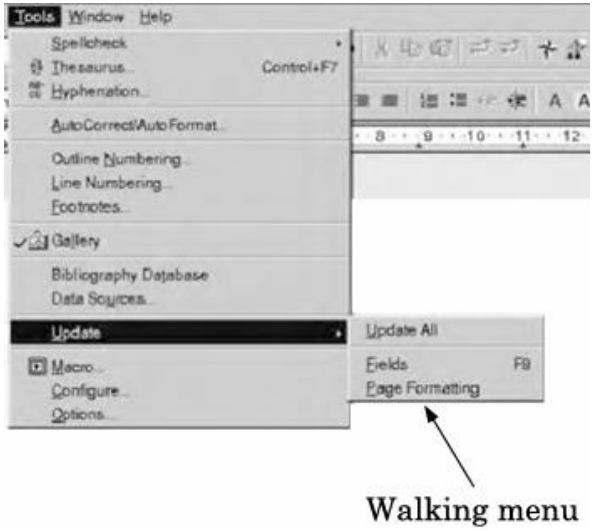
Menu-based Interface

- An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands.
- A menu-based interface is based on recognition of the command names, rather than recollection.
- Humans are much better in recognising something than recollecting it.
- Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device.



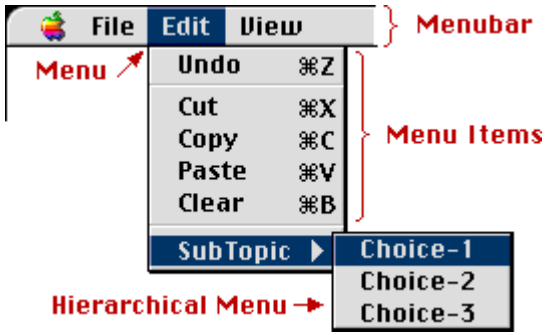
Scrolling menu

Scrolling menu:



Walking menu

Walking menu:



Hierarchical menu:

Direct Manipulation Interfaces

- Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons2 or objects).
- Direct manipulation interfaces are sometimes called as iconic interfaces.
- In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon representing a file into an icon representing a trash box, for deleting the file.
- Important advantages of iconic interfaces include the fact that the icons can be recognised by the users very easily, and that icons are language-independent.
- However, experienced users find direct manipulation interfaces very far too.
- It is difficult to give complex commands using a direct manipulation interface.

FUNDAMENTALS OF COMPONENT-BASED GUI DEVELOPMENT

- The current style of user interface development is component-based.
- It recognises that every user interface can easily be built from a handful of predefined components such as menus, dialog boxes, forms, etc.
- Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the window system.
- The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

Window System

Window: A window is a rectangular area on the screen.

A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g., one window can be used for editing a program and another for drawing pictures, etc.

A window can be divided into two parts—client part, and non-client part.

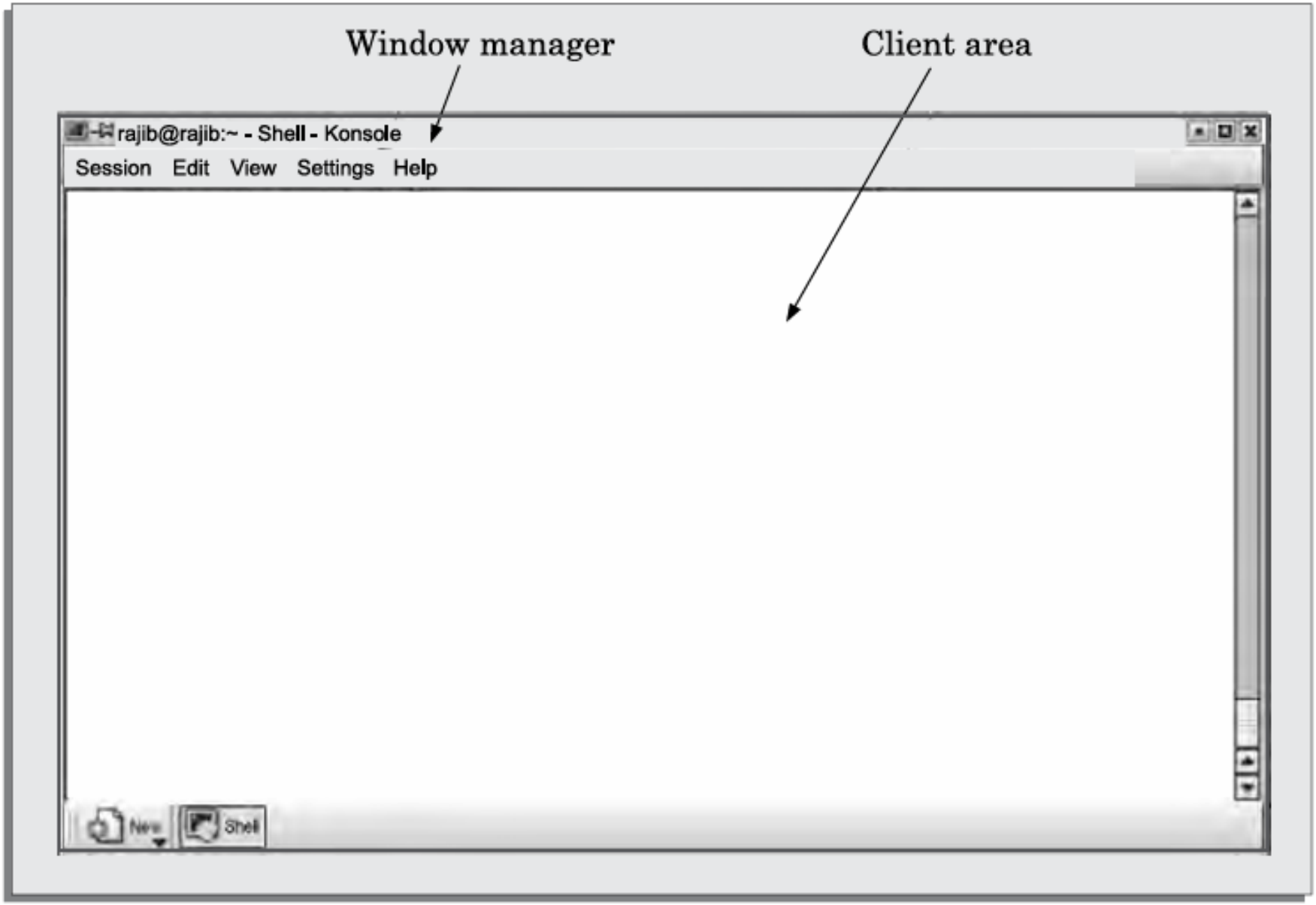
The client area makes up the whole of the window, except for the borders and scroll bars.

The client area is the area available to a client application for display.

The non-client-part of the window determines the look and feel of the window.

The look and feel defines a basic behaviour for all windows, such as creating, moving, resizing, iconifying of the windows.

The window manager is responsible for managing and maintaining the non-client area of a window.


Window manager
Client area
rajib@rajib:~ - Shell - Konsole
Session Edit View Settings Help
New
Shell

Window management system (WMS)

Window manager is the component of WMS with which the end-user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.

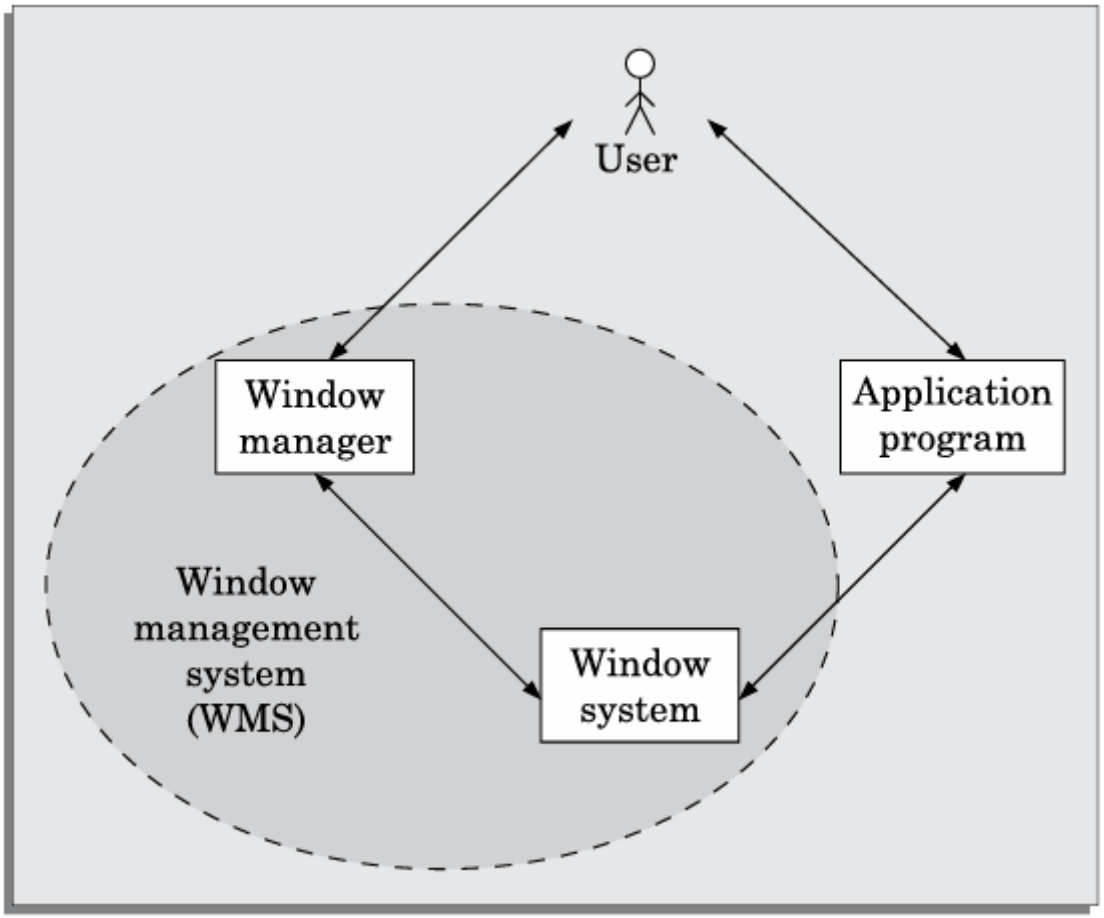


FIGURE 9.4 Window management system.

- Widgets are the standard user interface components.
- A user interface is usually made up by integrating several widgets.
- A development style based on widgets is called component-based (or widget-based) GUI development style.
- There are several important advantages of using a widget-based design style.
- One of the most important reasons to use widgets as building blocks is because they help users learn an interface fast.
- In this style of development, the user interfaces for different applications are built from the same basic components.
- Therefore, the user can extend his knowledge of the behaviour of the standard components from one application to the other.
- Also, the component-based user interface development style reduces the application programmer's work significantly as he is more of a user interface component integrator than a programmer in the traditional sense.
- A few important types of widgets normally provided with a user interface development system are described below:

Label widget: This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e., it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

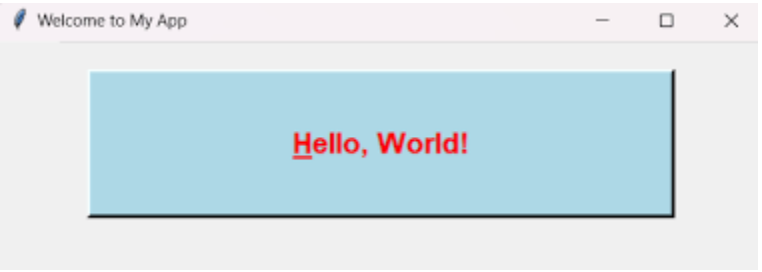
Container widget: These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.

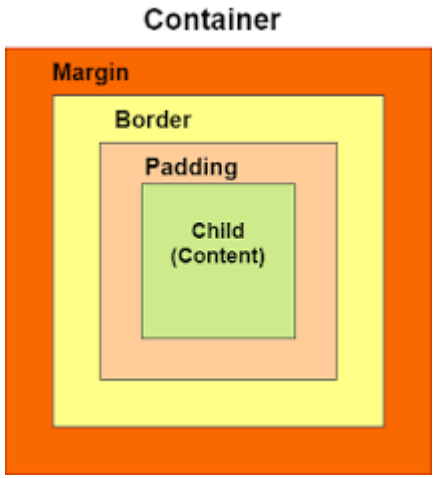
Pop-up menu: These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.

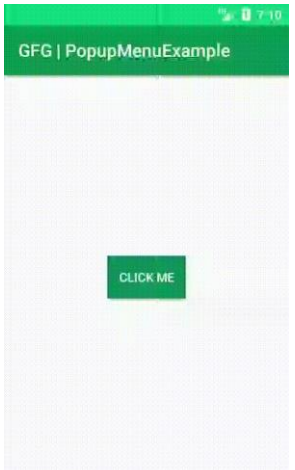
Pull-down menu: These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.

Dialog boxes: We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an *apply* command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Though most dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made. Generally, these boxes ask you to read the information presented and then click *OK* to dismiss the box.

Push button: A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (. . .). A push button with an ellipsis generally indicates that another dialog box will appear.







Radio buttons: A set of radio buttons are used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that were available in old radios.

Combo boxes: A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.