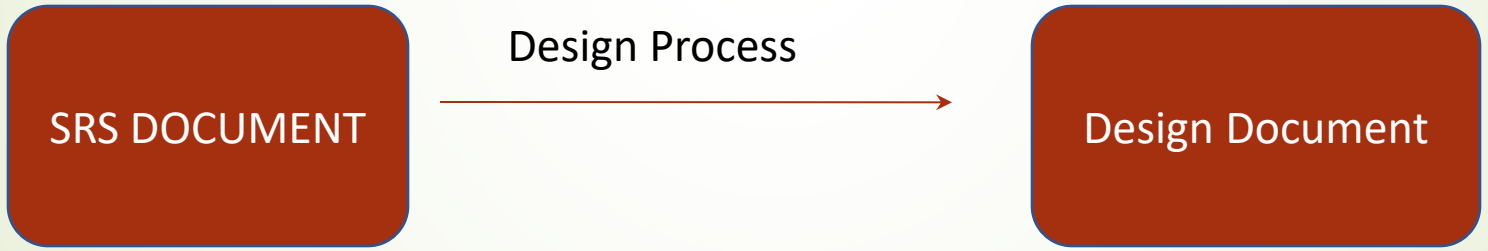
Software Design

- The activities carried out during the design phase transform the SRS document into the design Document



Software design and its activities

- Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language. A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps. Design activities can be broadly classified into two important parts:

- Preliminary (or high-level) design and

- Detailed design

Items to be Design…

Different modules required to implement the solution.

Control relationship among the identified modules

Interface among different module

Data structure of different modules

Algorithm required to implement individual module.

Characteristics of a good software design…

- **Correctness**: A good design should correctly implement all the functionalities identified in the SRS document.
- **Understandability:** A good design is easily understandable.
- **Efficiency**: It should be efficient.
- **Maintainability:** It should be easily enabled to change.

Features of a design document

In order to facilitate understandability, the design should have the following features:

- It should use consistent and meaningful names for various design components.
- The design should be modular. The term modularity means that it should use a cleanly decomposed set of modules.
- It should neatly arrange the modules in a hierarchy, e.g. in a tree-like diagram.

Classification of Design Phase

Through high level design, a problem is decomposed into a set of modules identified, and the interfaces among various modules identified.

During detailed design , each module is examined carefully to design its data structures and the algorithm.

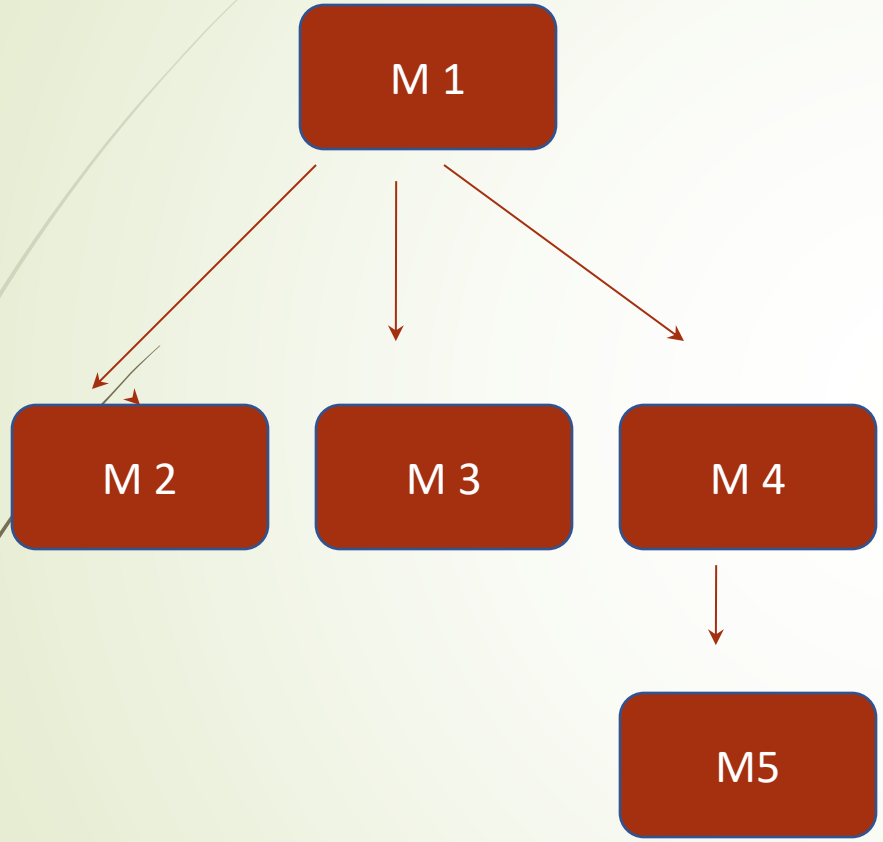
Modularity & Layered

A DESIGN SOLUTION IS UNDERSTANDABLE IF IT IS MODULAR & THE MODULES ARE ARRANGED IN LAYERS .

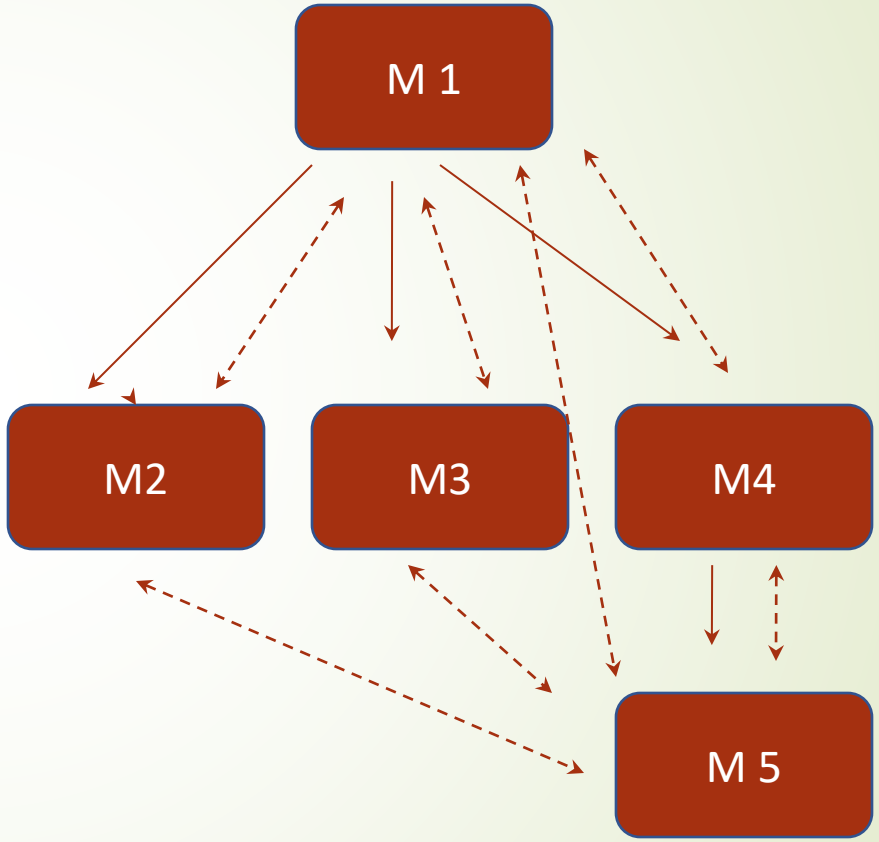
Modularity : it implies that the problem has been decomposed into a set of modules.

Decomposition of a problem into modules facilitates taking advantage of Divide & Conquer.

A design solution is considered to be highly modular , if the different modules in the solution have high cohesion and their inter-module coupling are low



A modular & hierarchical Design



A design solution exhibiting poor modularity & hierarchy

Layered Design (Neat Arrangement)

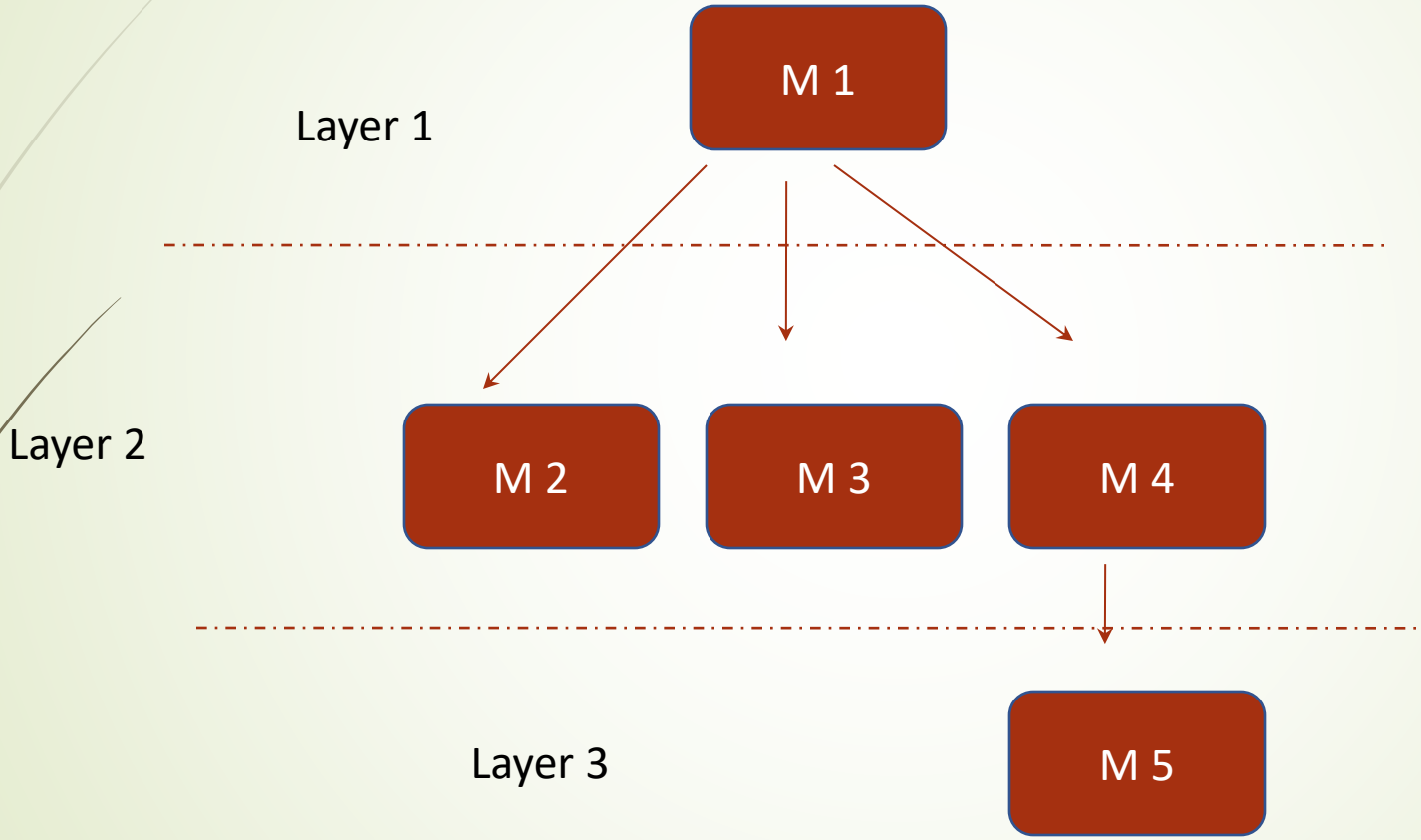
A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree like diagram with clear layering.

A module can only invoke functions of the modules in the layer immediately below it. (Visibility & Control Abstraction)

Higher layer modules can be considered to be similar to manager that invoke the lower layer modules to get certain task done.

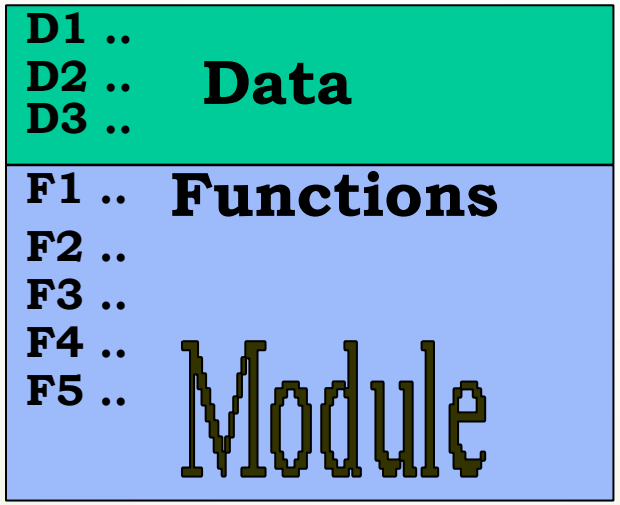
Fan-out : It is a measure of the number of modules that are directly controlled by a given module .A design having modules with high fan out numbers is not a good design as such modules modules would lack cohesion.

Fan in: It indicates the number of directly invoking a given module .High fan in represents code reuse.


Layer 1
M 1
Layer 2
M 2
M 3
M 4
Layer 3
M 5

Introduction

- A module consists of:
  - Several functions
  - Associated data structures.

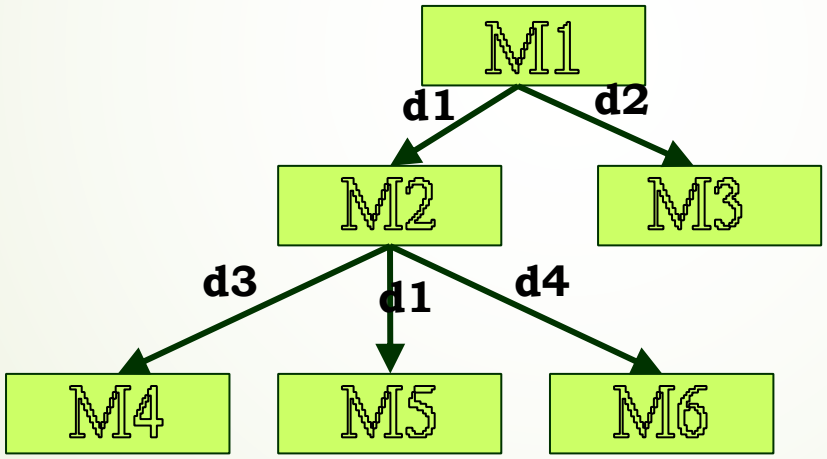


Introduction

- Design activities are usually classified into two stages:
  - High-level design.
  - Detailed design.

High-Level Design

- Identify:
  - Modules
  - Control relationships among modules
  - Interfaces among modules.



High-Level Design

- Several notations are available to represent high-level design:
  - Usually a tree-like diagram called structure chart is used.

Detailed Design

- For each module, design:
  - Data structure
  - Algorithms

Good and Bad Designs

- There is no unique way to design a system.
- Even using the same design methodology:
  - Different designers can arrive at different design solutions.
- We need to distinguish between good and bad designs.

Good SOFTWARE Design?

- Should implement all functionalities of the system correctly.
- Should be easily understandable.
- Should be efficient.
- Should be easily maintainable.
- Should use consistent and meaningful names:
  - for various design components.

Abstraction & Decomposition in Design?

- Two principal ways:
  - Modular Design
  - Layered Design

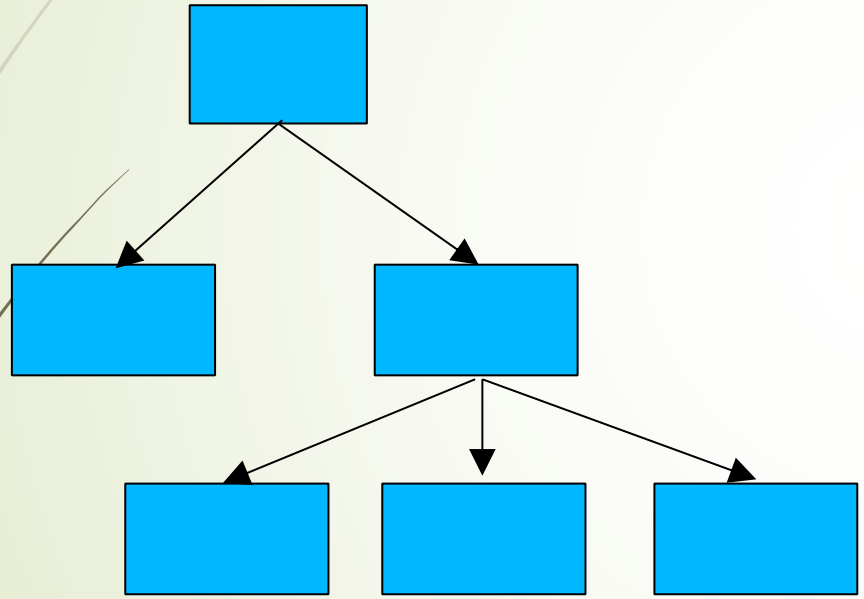
Modularity

- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem clearly into modules:
  - Modules are almost independent of each other
  - Divide and conquer principle.

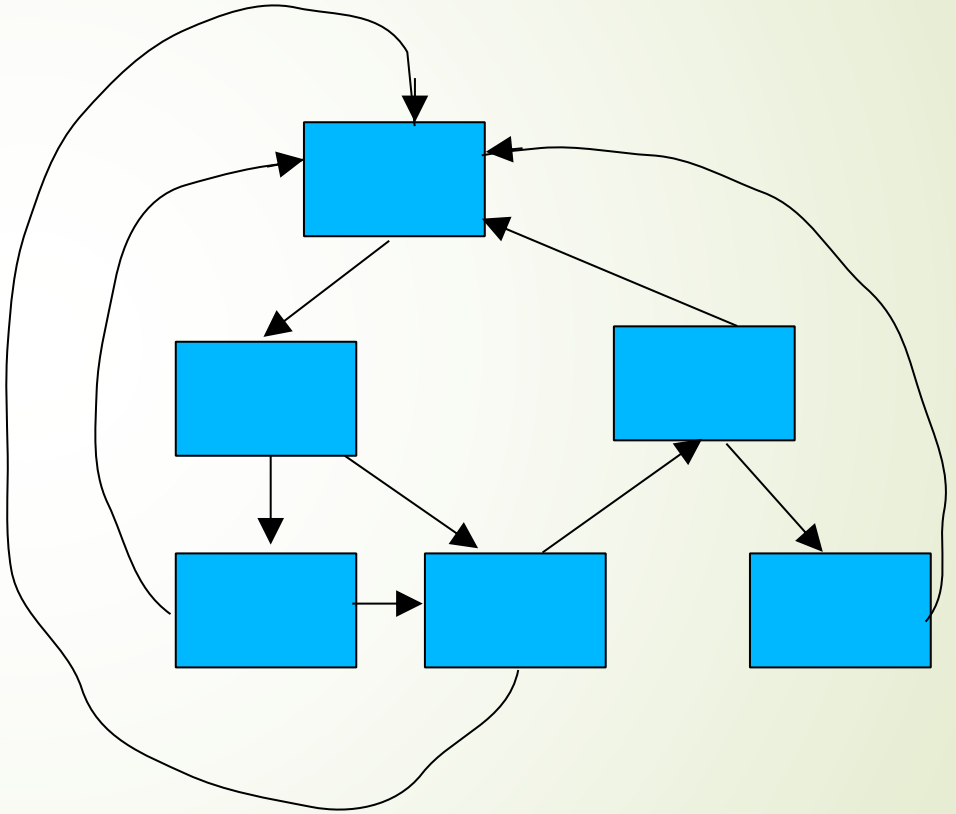
Modularity

- If modules are independent:
  - Modules can be understood separately,
    - Reduces the complexity greatly.

Layered Design



YES



NO

Layered Design

- Neat arrangement of modules in a hierarchy means:
  - Control abstraction among modules.

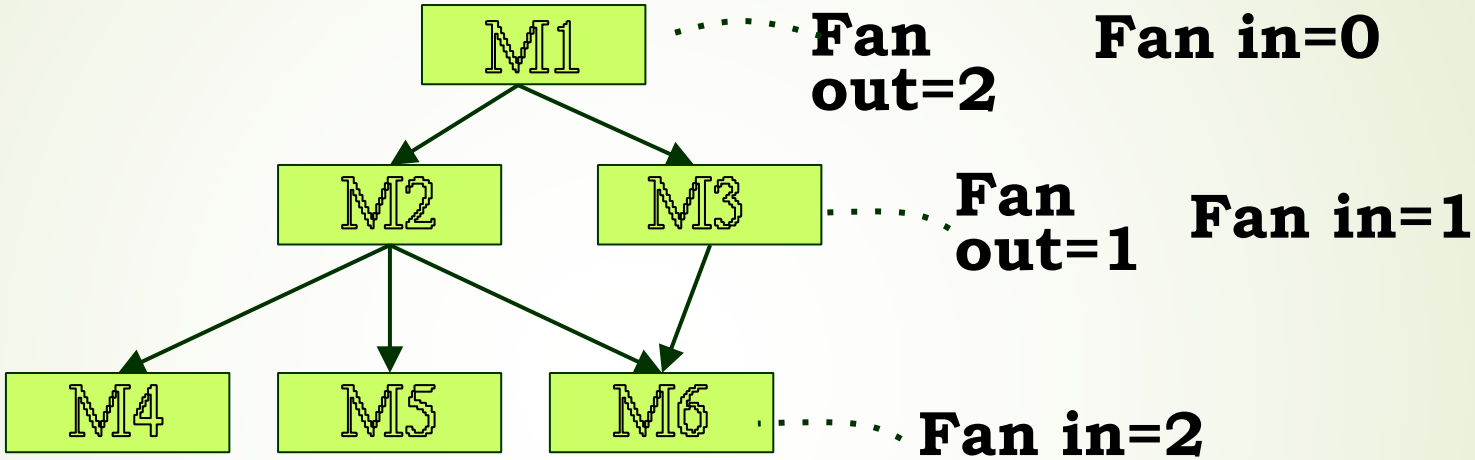
Characteristics of Module STRUCTURE

- Depth:
  - Number of levels of control
- Width:
  - Overall span of control.
- Fan-out:
  - A measure of the number of modules directly controlled by given module.

Characteristics of Module STRUCTURE

- Fan-in:
  - Indicates how many modules directly invoke a given module.

Module Structure



Layered Design

- A design having modules:
  - With high fan-out numbers is not a good design:
  - A module having high fan-out lacks cohesion.

Goodness of Design

- A module that invokes a large number of other modules:
  - Likely to implement several different functions:
  - Not to perform a single cohesive function.

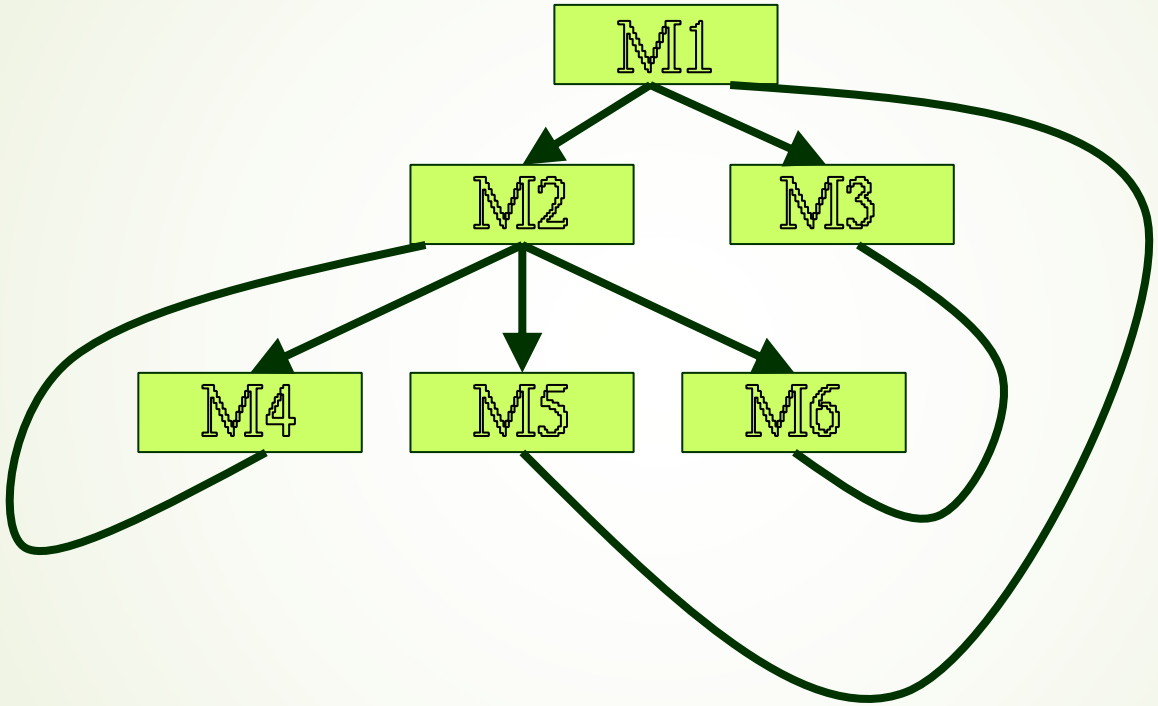
Control Relationships

- A module that controls another module:
  - said to be super-ordinate to it.
- Conversely, a module controlled by another module:
  - said to be subordinate to it.

Visibility and Layering

- Module B is said to be visible by another module A,
  - If A directly calls B.
- The layering principle requires
  - Modules at a layer can call only the modules immediately below it.

Bad Design



Abstraction

- A module is unaware of the higher level modules.
- The modules at the higher layers should not be visible to the modules at the lower layers.

Abstraction

- The principle of abstraction requires:
  - Lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.

Modularity

- In technical terms, modules should display:
  - High cohesion
  - Low coupling.
- What are
  - cohesion and coupling???

Cohesion and Coupling

- Cohesion is a measure of:
  - functional strength of a module.
  - A cohesive module performs a single task or function.
- Coupling between two modules:
  - A measure of the degree of the interdependence or interaction between the two modules.

Cohesion and Coupling

- A module having high cohesion and low coupling:
  - functionally independent of other modules:
    - A functionally independent module has minimal interaction with other modules.

Advantages of Functional Independence

- Better understandability,
- Complexity of design is reduced,
- Different modules easily understood in isolation.

Advantages of Functional Independence

- Functional independence reduces error propagation.
  - Degree of interaction between modules is low.
  - An error existing in one module does not directly affect other modules.

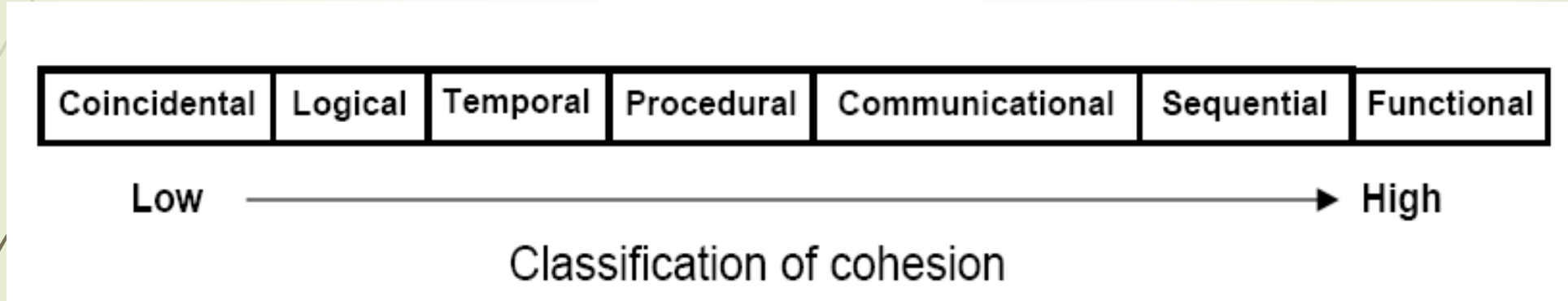
Advantages of Functional Independence

- A functionally independent module:
  - Can be easily taken out and reused in a different program.
    - Each module does some well-defined and precise function.

Cohesion

- The primary characteristics of neat module decomposition are high cohesion and low coupling.
- Cohesion is a measure of functional strength of a module.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Classification of cohesion



Coincidental cohesion:

- The module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.
  - In (TPS), the get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem

Logical cohesion:

- A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

Temporal cohesion:

- When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Procedural cohesion:

- A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, **e.g. the algorithm for decoding a message.**

Communicational cohesion:

- A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

Sequential cohesion:

- A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

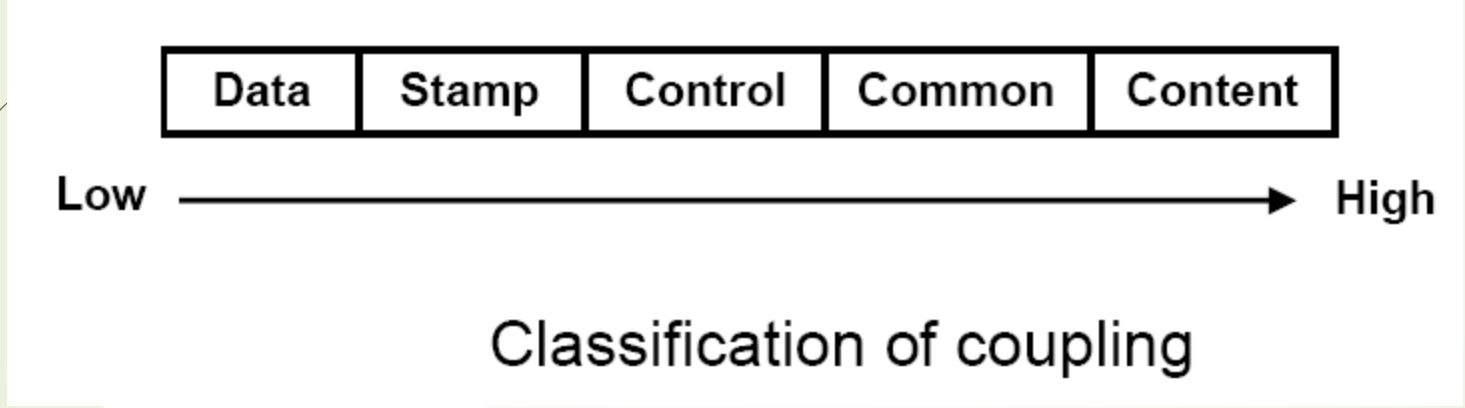
Functional cohesion:

- Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

Coupling

- Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Classification of Coupling



Classification of coupling

Data coupling:

- Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

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 instructions 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 data through some global data items.

Content coupling:

- Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module

Functional independence

- A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

Need for functional independence

Functional independence is a key to any good design due to the following reasons:

- Error isolation: Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.

Need for functional independence

- Scope of reuse: Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.

Need for functional independence

- Understandability: Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

Function-oriented design

The following are the salient features of a typical function-oriented design approach:

- **A system is viewed as something that performs a set of functions. Starting at this high-level view of the system, each function is successively refined into more detailed functions.** For example, consider a function create-new-library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This function may consist of the following sub-functions:

  - assign-membership-number
  - create-member-record
  - print-bill

- **2. The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updating to several functions such as:**
- create-new-member
- delete-member
- update-member-record

Object-oriented design

- **In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information.** For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

Function-oriented vs. object-oriented design approach

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc.
- Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.
- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system.

Functional Oriented SD

Structured Analysis[SA] :

- Structured analysis is used to carry out the top-down decomposition of a set of high-level functions depicted in the problem description and to represent them graphically. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analyzed and hierarchically decomposed into more detailed functions

Structured analysis technique is based on the following essential underlying principles:

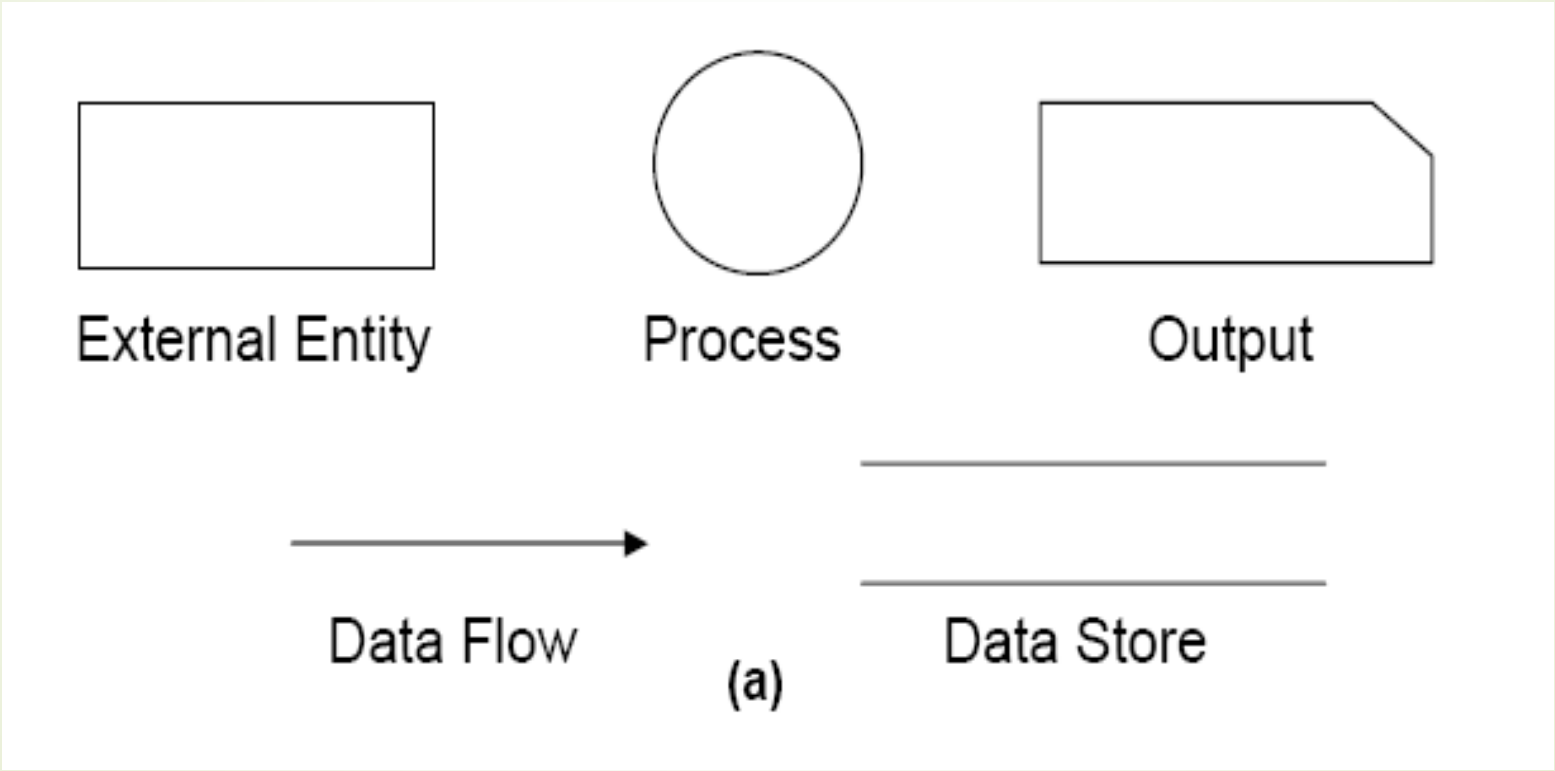
- Top-down decomposition approach.
- Divide and conquer principle. Each function is decomposed independently.
- Graphical representation of the analysis results using Data Flow Diagrams (DFDs).


SRS
SA
DFD
SD
STRUCTURE CHART

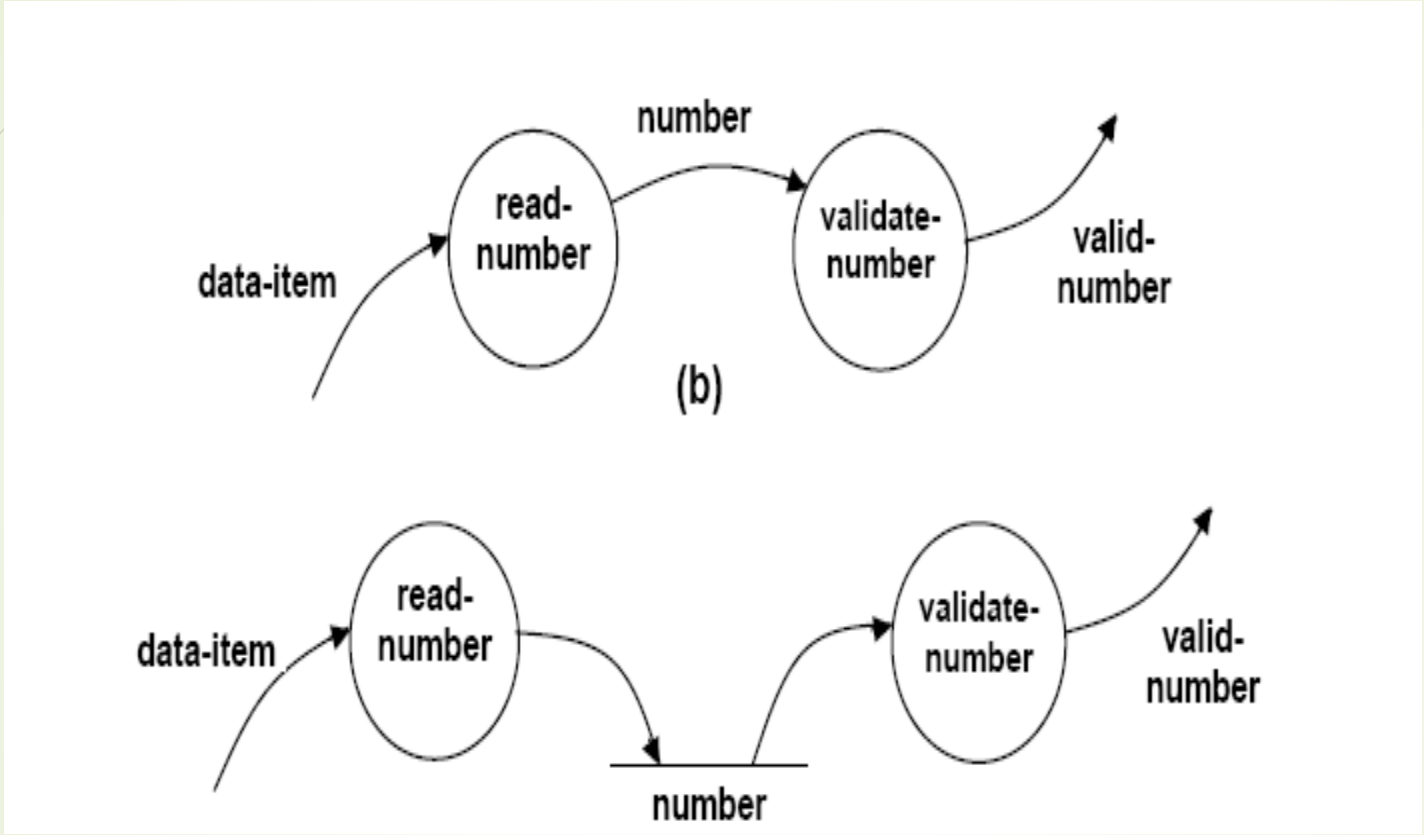
DFD

- The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions.

Symbols used in DFD



Synchronous & Asynchronous DFD



(a)

(a) Asynchronous Chart (b) Synchronous Chart

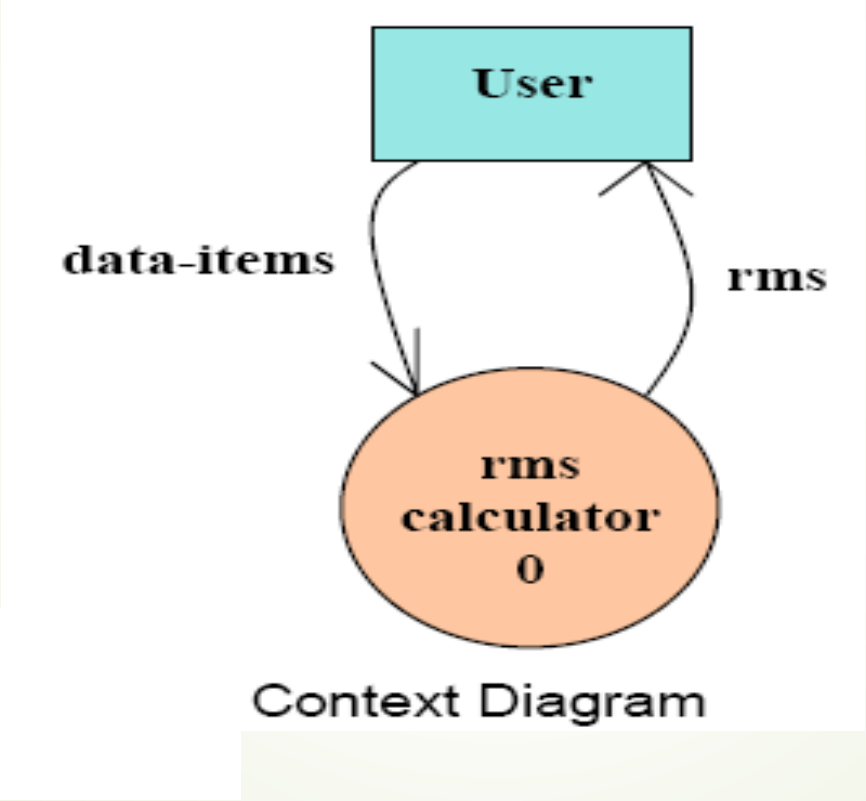
Context diagram /Level 0 DFD

- The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble.
- The context diagram is also called as the level O DFD.

RMS Calculating Software.

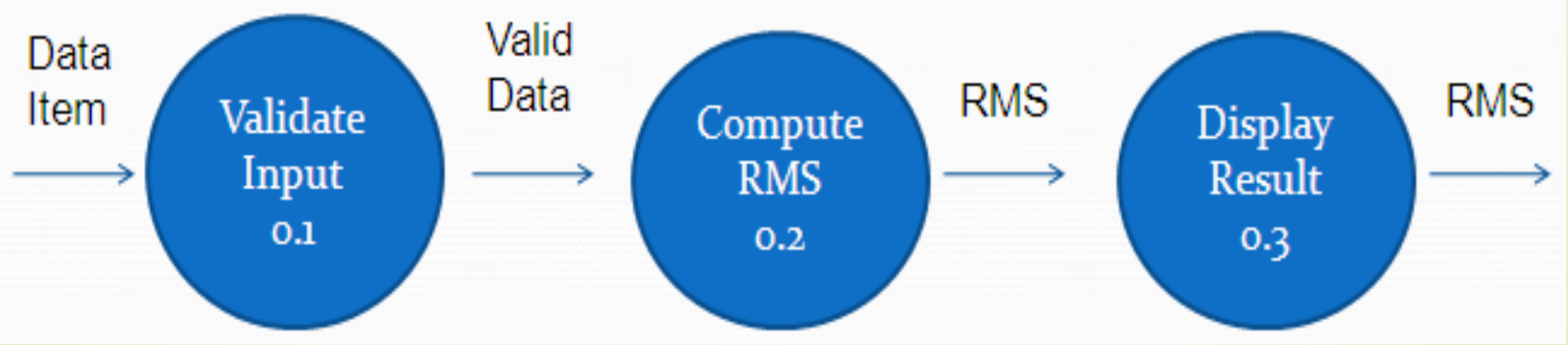
- A software system called RMS calculating software would read three integral numbers from the user in the range of -1000 and +1000 and then determine the root mean square (rms) of the three input numbers and display it.

Level 0 DFD/ Context Diagram

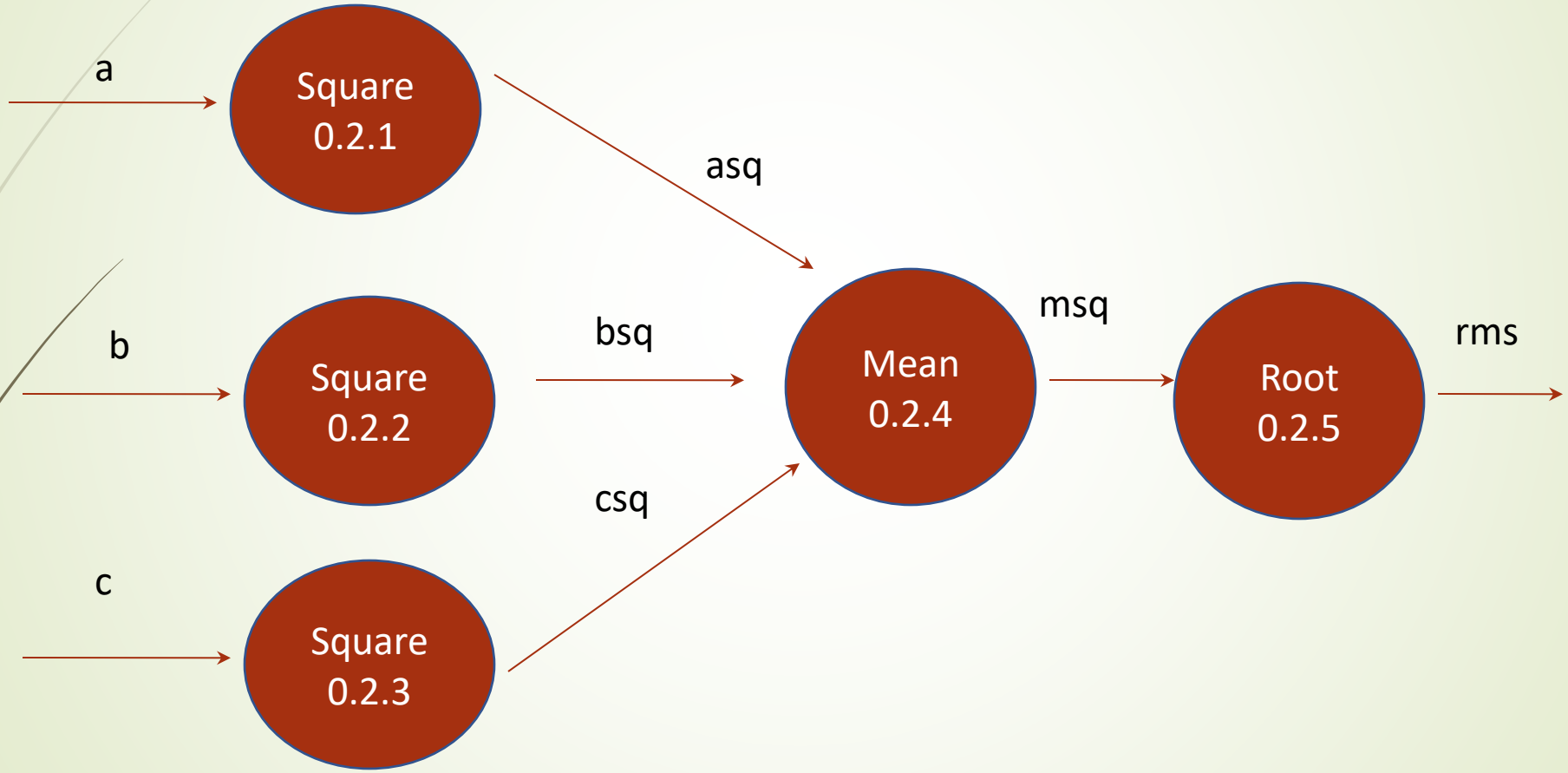


Level 1 DFD

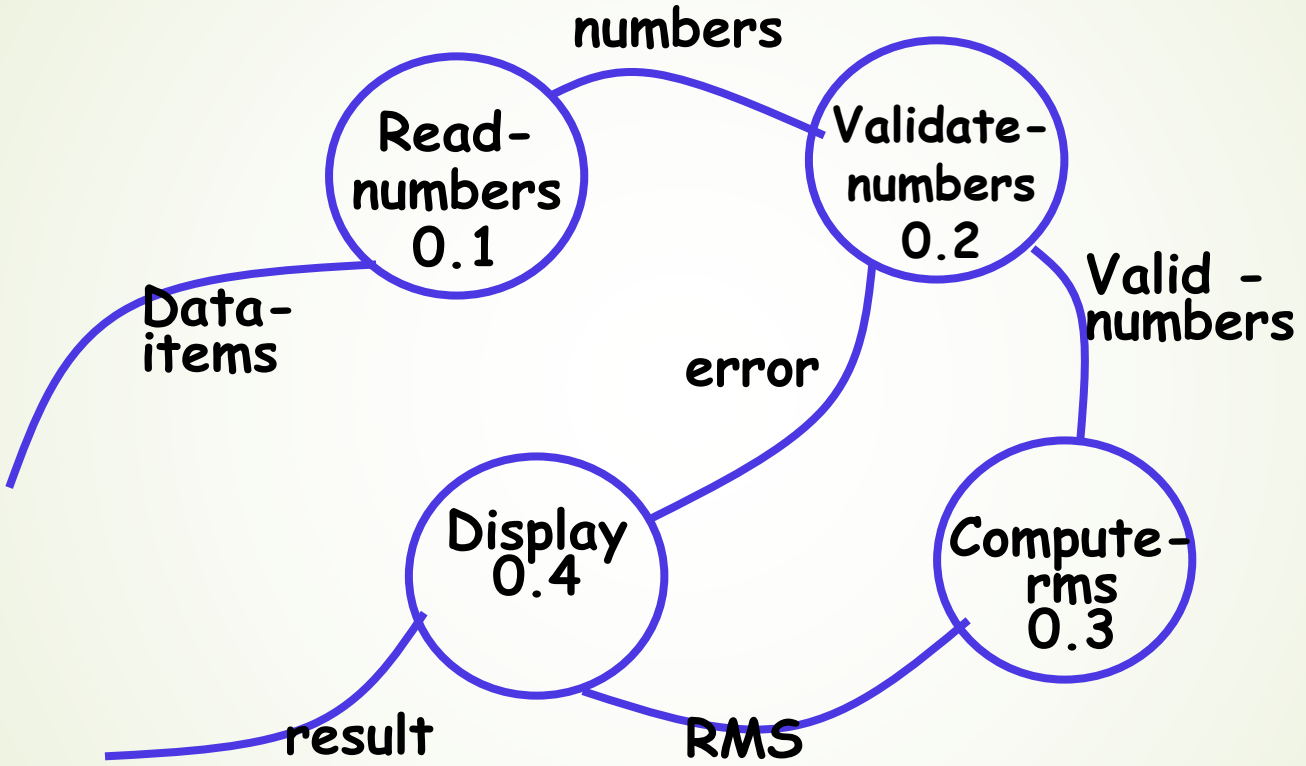
- To develop the level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the level 1 DFD.



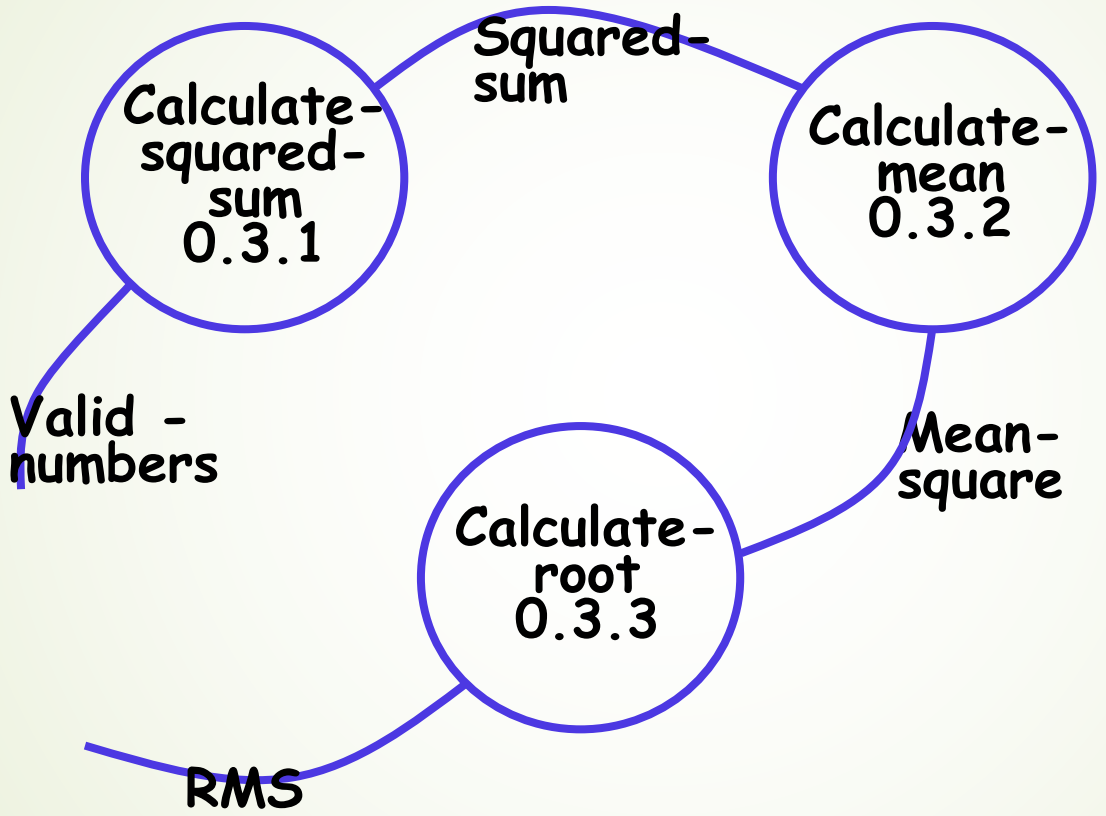
Level 2 DFD


a
Square
0.2.1
asq
b
Square
0.2.2
bsq
Mean
0.2.4
msq
Root
0.2.5
rms
c
Square
0.2.3
csq

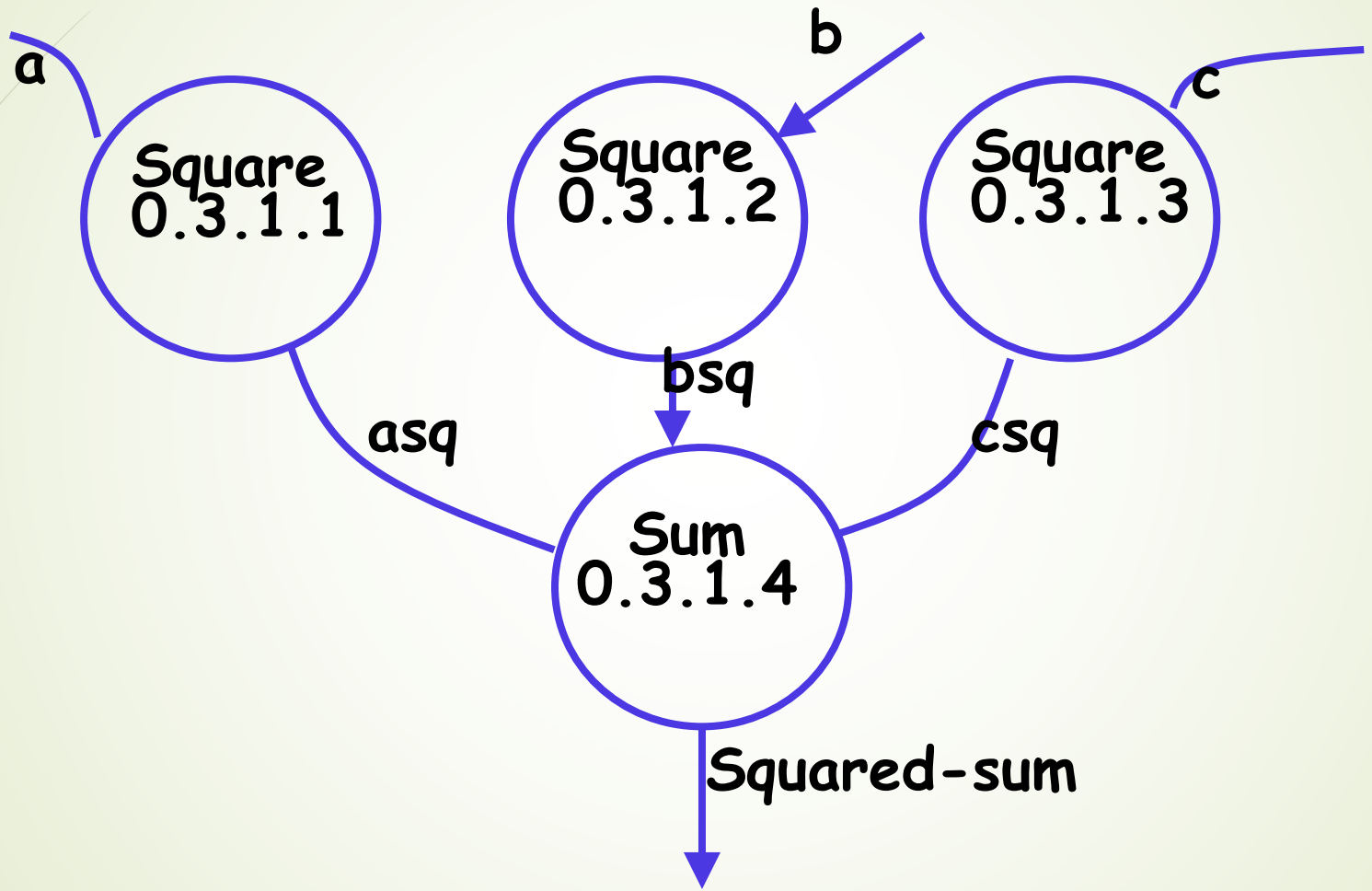
RMS Calculating Software
Level 1



RMS Calculating Software
Level 2

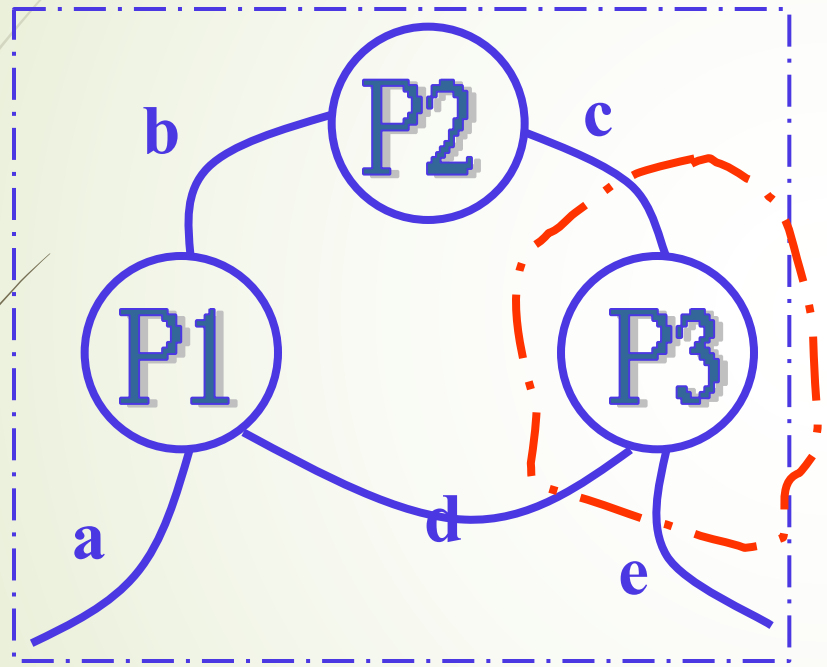


RMS Calculating Software
Level 3

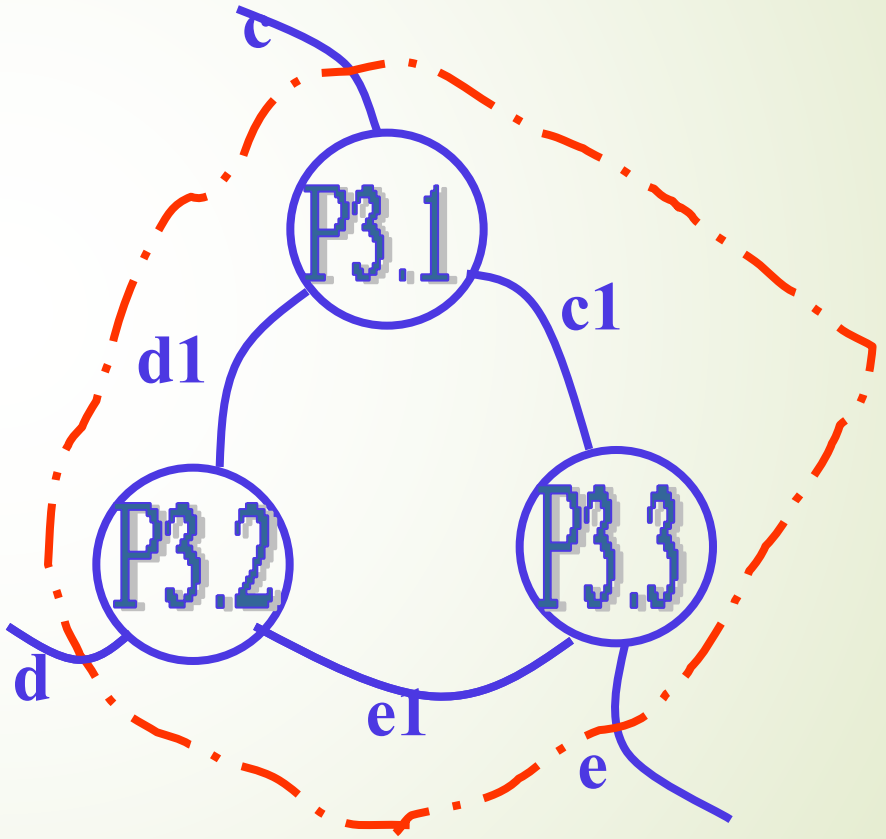


Balancing DFDs

The Data that flow into or out of a bubble must match the data flow at the nest level of DFD. This is known as Balancing a DFD



Level 1



Level 2

Data Dictionary

- Every DFD model of a system must be accompanied by data dictionary. A data dictionary lists all data items appearing in the DFD model of the system.

Why Data Dictionary

- A data dictionary provides a standard terminology for all relevant data for use by all engineers working in the same project.
- It provides the analyses with means to determine the definition of different in terms of their component element.

DD for RMS Software

data_ items : {integer}3

rms : float

valid_data : data_items

a= integer

b= integer

c = integer

asq=integer

bsq= integer

csq= integer

msq= integer

Importance of Data Dictionary

- Provides all engineers in a project with standard terminology for all data:
  - A consistent vocabulary for data is very important
  - Different engineers tend to use different terms to refer to the same data,
    - Causes unnecessary confusion.

Importance of Data Dictionary

- Data dictionary provides the definition of different data:
  - In terms of their component elements.
- For large systems,
  - The data dictionary grows rapidly in size and complexity.
  - Typical projects can have thousands of data dictionary entries.
  - It is extremely difficult to maintain such a dictionary manually.

Data Definition

- Composite data are defined in terms of primitive data items using following operators:
- +: denotes composition of data items, e.g
  - a+b represents data a and b.
- [,,,]: represents selection,
  - i.e. any one of the data items listed inside the square bracket can occur.
  - For example, [a,b] represents either a occurs or b occurs.

Data Definition

- ( ): contents inside the bracket represent optional data
  - which may or may not appear.
  - a+(b) represents either a or a+b occurs.
- {}: represents iterative data definition,
  - e.g. {name}5 represents five name data.

Data Definition

- {name}* represents
  - zero or more instances of name data.
- = represents equivalence,
  - e.g. a=b+c means that a represents b and c.
- * *: Anything appearing within * * is considered as comment.

For DFD Refer Example

- Design the different level of DFD For SuperMarket Prize Scheme. Also design the Data Dictionary For the DFD model.
- Design L0 and L1 DFD for Tic-Tac-Toe Game. Also design the Data Dictionary.
- Design L0 L1 and L2 DFD for Personal Library System. Also design the Data Dictionary.

Advantage of DFD

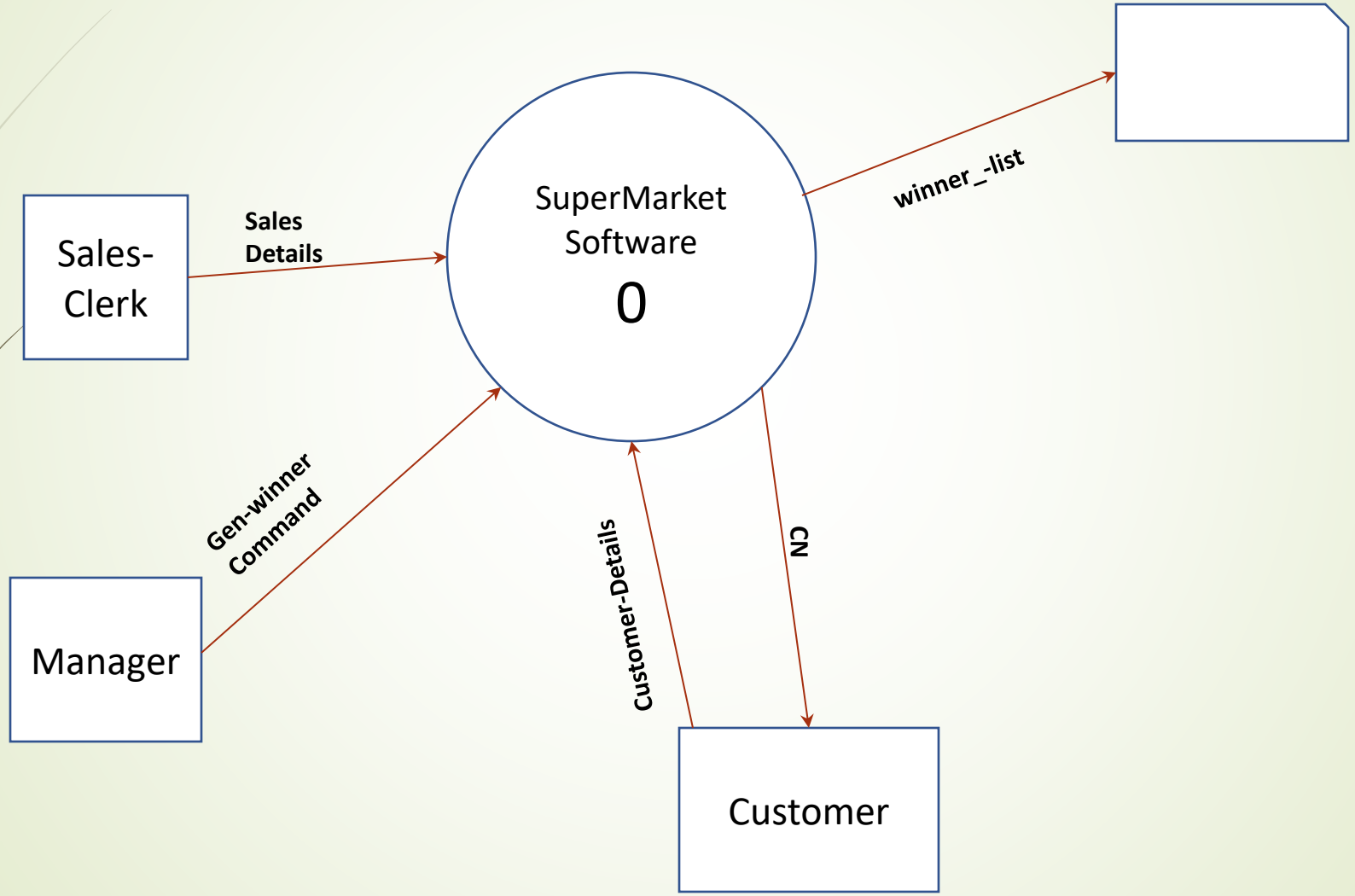
- Simple graphical notation to represent the whole system with primitive symbols.
- It is very simple formalism , simple to understand & rescue
- It supports modular programming approach starting with a set of high level functions that a system performs, a DFD model hierarchically represent various sub functions.
- Different level of abstraction are possible.

Shortcomings of a DFD model

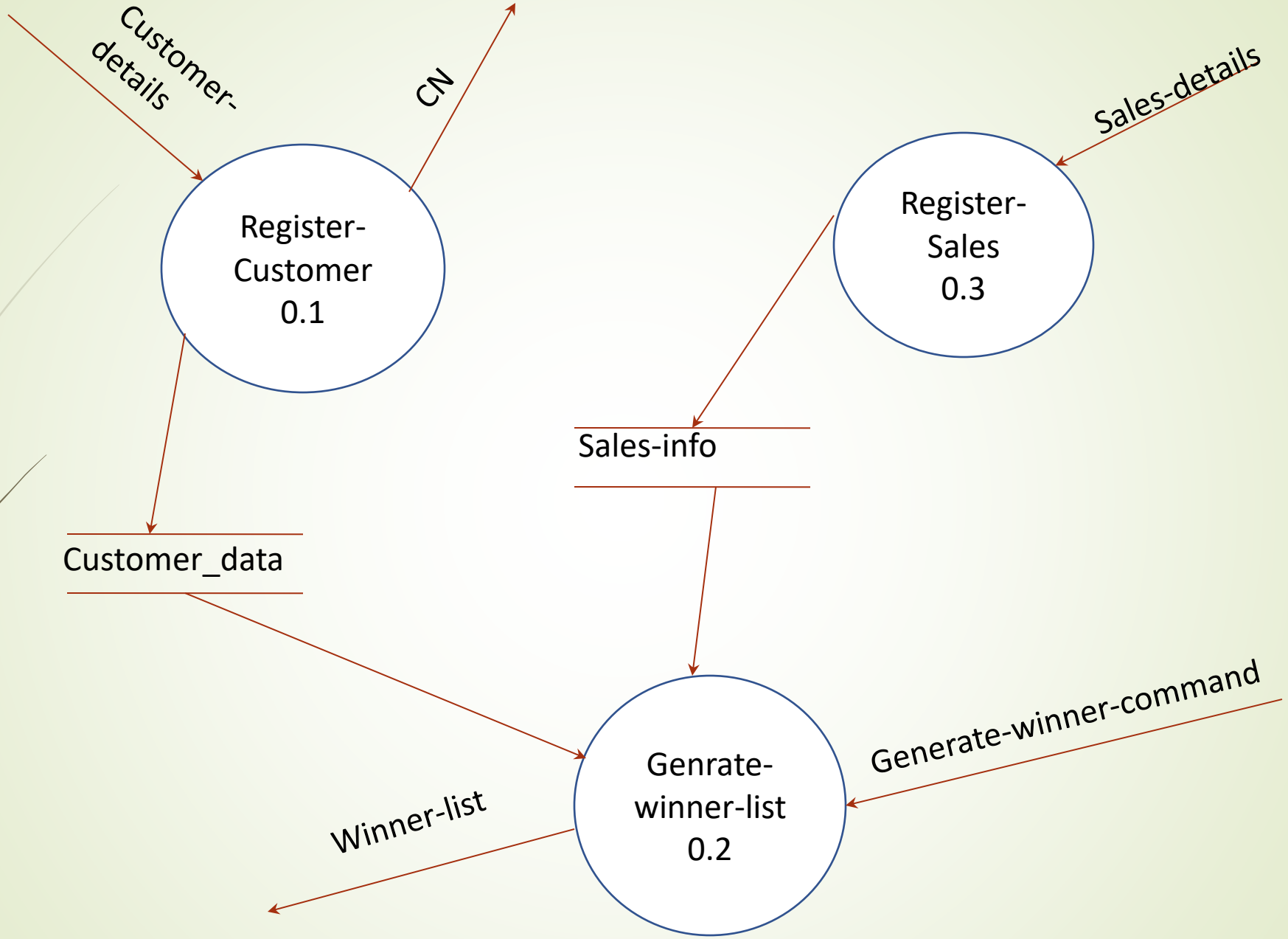
- Scalability: DFDs for large applications are difficult to handle.
- Lack of ordering: A DFD model does not specify the order in which the different bubbles are executed.
- No specific rule for decomposition: The level of decomposition is highly subjective and depends on the choice of the analyst.

For the same problem, several DFD representations are possible.

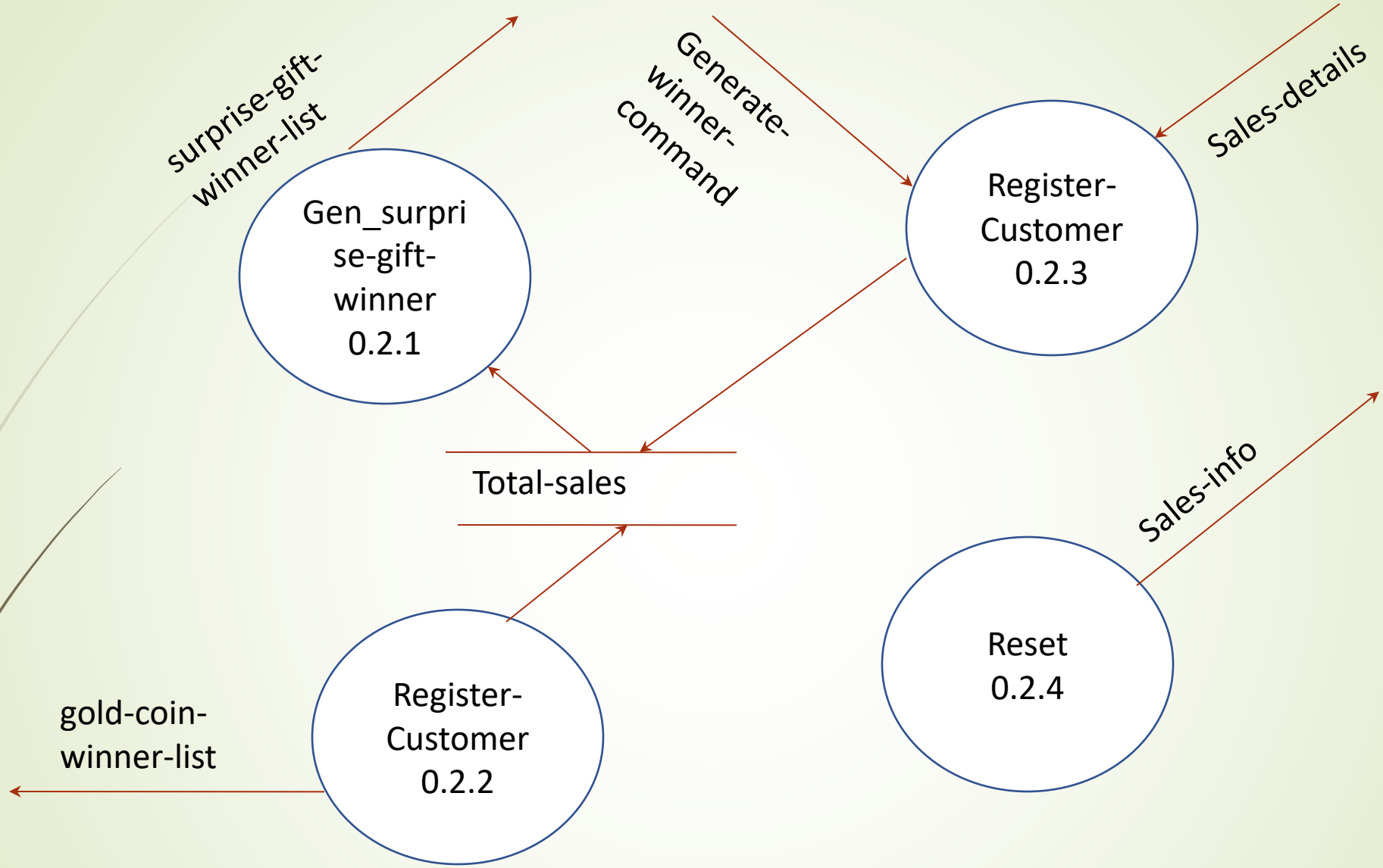
Supermarket Prize Distribution



Context Diagram



Level 1 Diagram



Level 2 Diagram

DD for Supermarket Prize Software

- address:name+house#+street#+city+pin
- Sales-details:{item+amount}* + CN
- CN:integer
- Customer-data:{address+CN}*
- Sales-info:{sales-details}*
- Winner-list:surpize-gift-winner-list+gold-coin-winner-list
- Surprise-gift-winner-list:{address+CN}*
- Gold-coint-winner-list:{address+CN}*
- Gen-winner-command:command
- Total-sales:{CN+integer}*

Structured Design

- The aim of structured design
  - Transform the results of structured analysis (i.e., a DFD representation) into a structure chart.
- A structure chart represents the software architecture:
  - Various modules making up the system,
  - Module dependency (i.e. which module calls which other modules),
  - Parameters passed among different modules.

Structure Chart

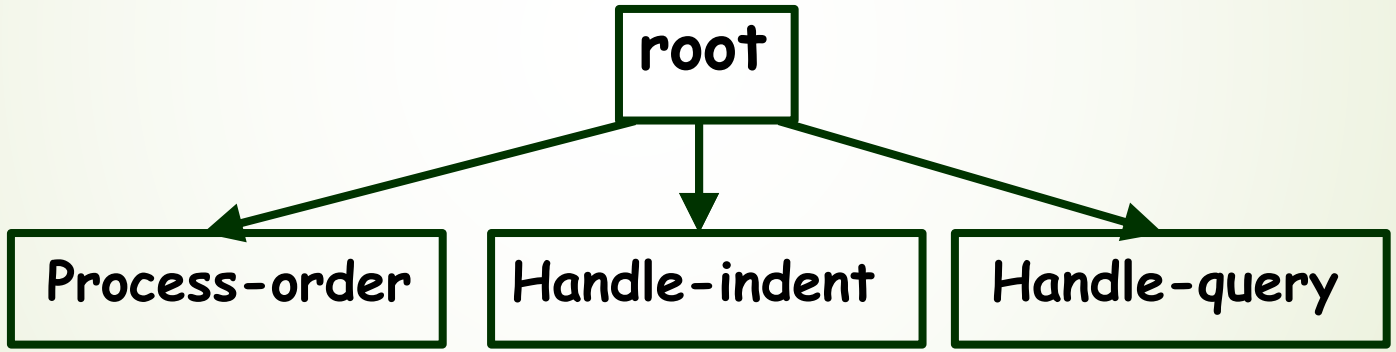
- Structure chart representation
  - Easily implementable using programming languages.
- Main focus of a structure chart:
  - Define the module structure of a software.
  - Interaction among different modules.

Basic Building Blocks of Structure Chart

- Rectangular box:
  - A rectangular box represents a module.
  - A name is assigned to the module it represents.

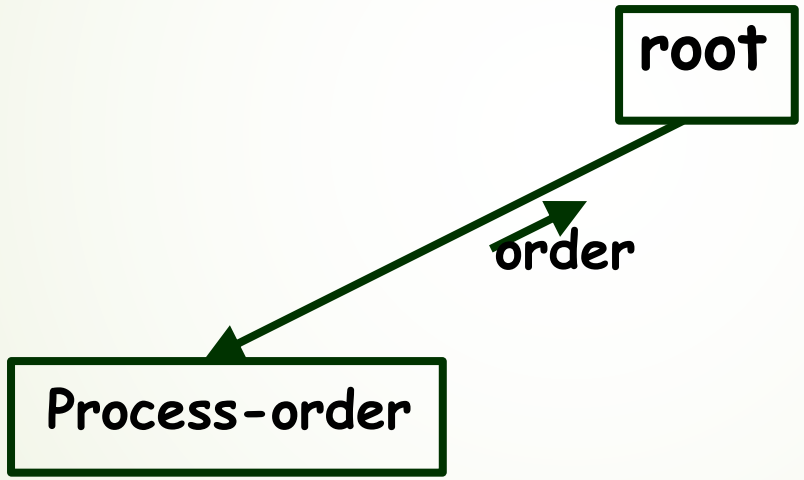
Basic Building Blocks of Structure Chart

- An arrow between two modules implies:
  - During execution control is passed from one module to the other in the direction of the arrow.



Basic Building Blocks of Structure Chart

- Data flow arrows represent:
  - Data passing from one module to another in the direction of the arrow.

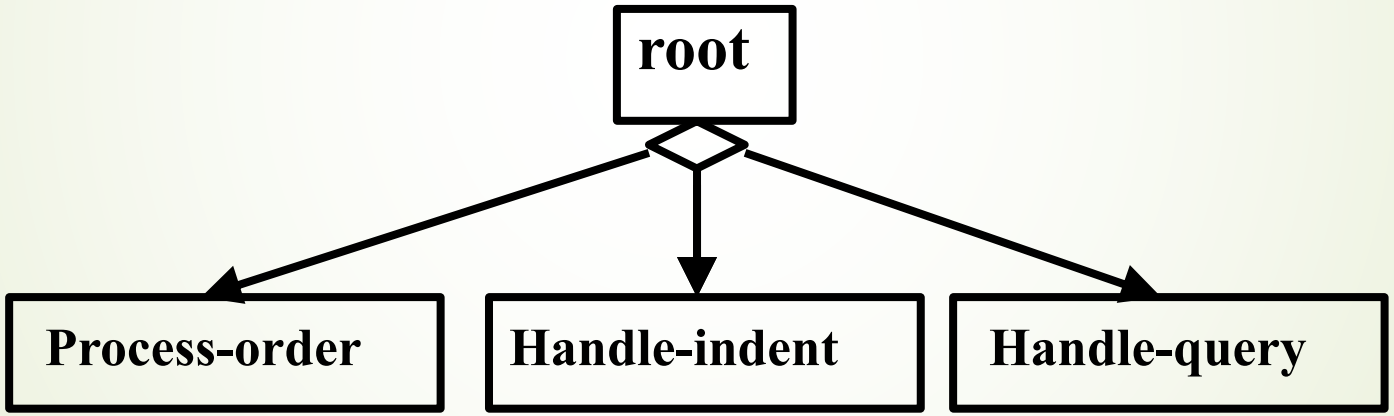


Basic Building Blocks of Structure Chart

- Library modules represent frequently called modules:
  - A rectangle with double side edges.
  - Simplifies drawing when a module is called by several modules.

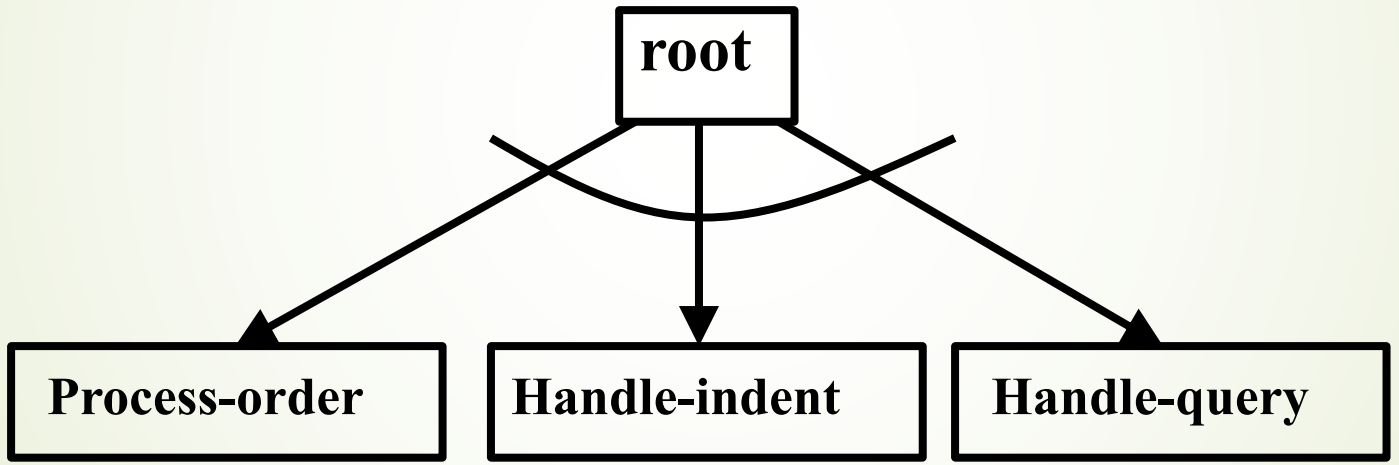
Basic Building Blocks of Structure Chart

- The diamond symbol represents selection:
  - One module of several modules connected to the diamond symbol is invoked depending on some condition.



Basic Building Blocks of Structure Chart

- A loop around control flow arrows denotes that the concerned modules are invoked repeatedly.



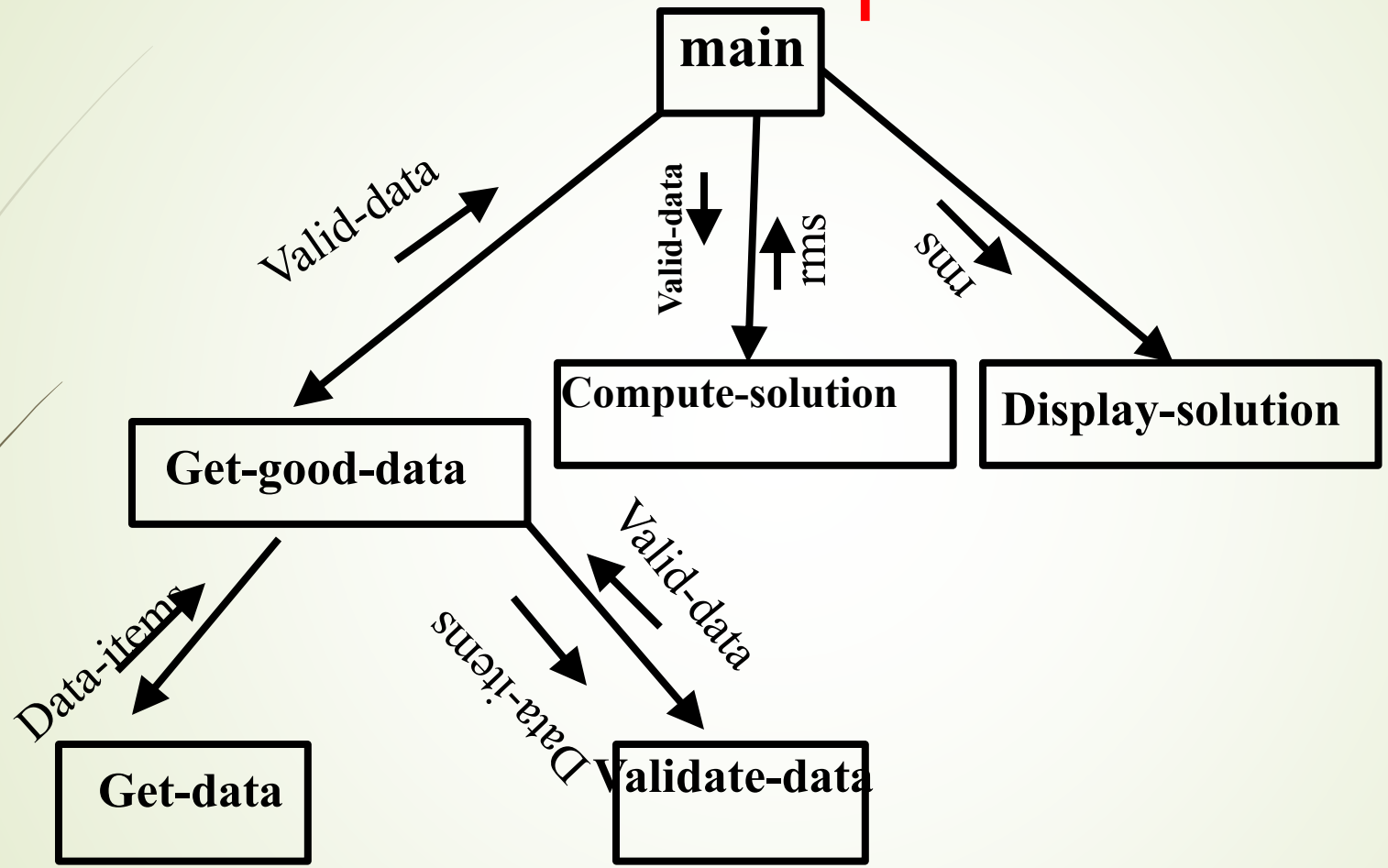
Structure Chart

- There is only one module at the top:
  - root module.
- There is at most one control relationship between any two modules:
  - if module A invokes module B,
  - Module B cannot invoke module A.
- The main reason behind this restriction:
  - consider modules in a structure chart to be arranged in layers or levels.

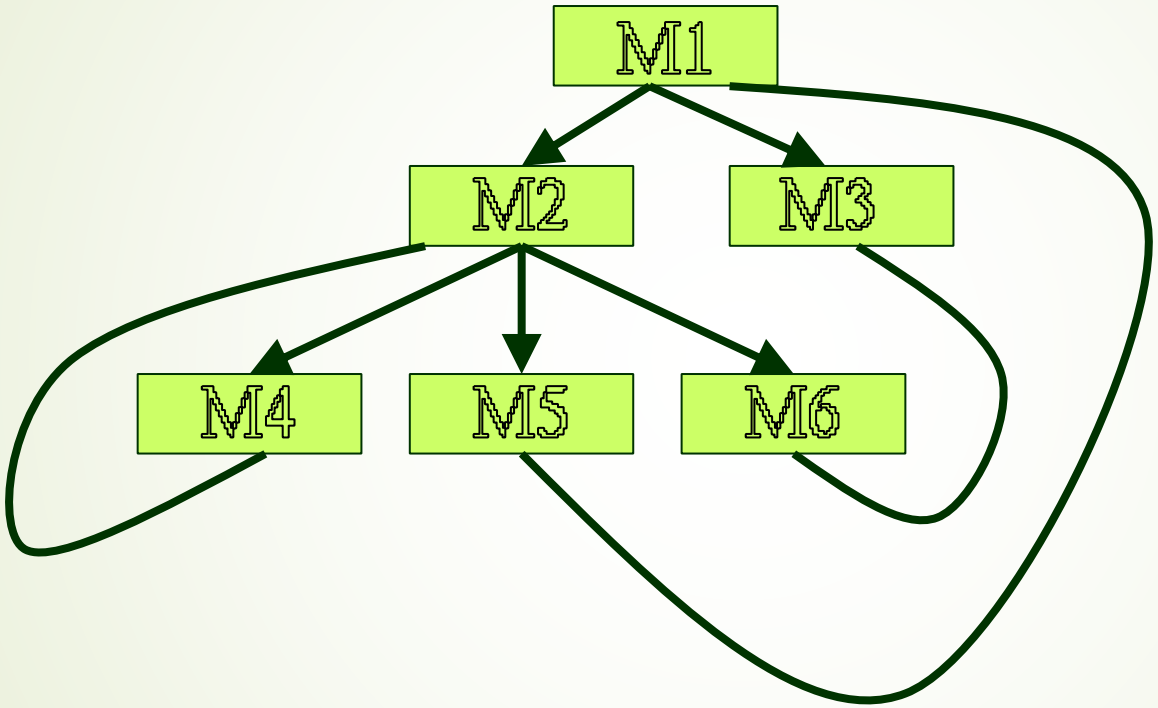
Structure Chart

- The principle of abstraction:
  - does not allow lower-level modules to invoke higher-level modules:
  - But, two higher-level modules can invoke the same lower-level module.

Example


main
Valid-data
Valid-data
rms
sum
Compute-solution
Display-solution
Get-good-data
Data-items
Data-items
Valid-data
Get-data
Validate-data

Bad Design

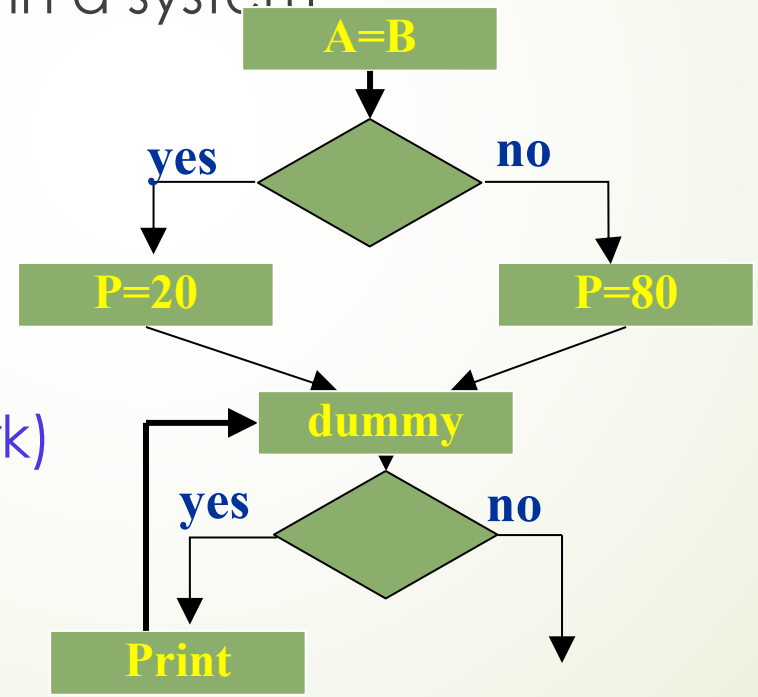


Shortcomings of Structure Chart

- By looking at a structure chart:
  - we can not say whether a module calls another module just once or many times.
- Also, by looking at a structure chart:
  - we can not tell the order in which the different modules are invoked.

Flow Chart

- We are all familiar with the flow chart representations:
  - Flow chart is a convenient technique to represent the flow of control in a system
- A=B
- if(c == 100)
- P=20
- else p= 80
- while(p>20)
- print(student mark)



Flow Chart vs. Structure Chart

- A structure chart differs from a flow chart in three principal ways:
  - It is difficult to identify modules of a software from its flow chart representation.
  - Data interchange among the modules is not represented in a flow chart.
  - Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

Questions

- Questions on Software Design

1. What is the main objective of software design in software engineering?
2. What are the two main stages of the design process in software development?
3. What are the characteristics of a good software design?
4. How does modularity contribute to a good design in software engineering?

- Questions on Design Documentation

5. What are the features of a well-structured design document?
6. Why is it important for design documents to use consistent and meaningful names for various design components?

- Questions on Modularity and Layering

7. Explain the concept of modularity in software design.
8. What is the difference between high cohesion and low coupling in the context of software design?
9. How does layered design improve the understandability of a software system?

- Questions on Design Phases

10. What activities are carried out during the high-level design phase?
11. What is detailed design, and how does it differ from high-level design?

- Questions on Cohesion and Coupling

12. Define cohesion in the context of software design.
13. What is coupling, and why is low coupling desirable in software design?
14. Describe the different types of cohesion that can exist within a module.
15. What are the different types of coupling between software modules?

- Questions on Function-Oriented and Object-Oriented Design

16. How does function-oriented design differ from object-oriented design?
17. What are the key principles of the function-oriented design approach?
18. Describe the concept of data flow diagrams (DFDs) in function-oriented design.

- Questions on Abstraction and Decomposition

19. What are abstraction and decomposition, and how are they applied in software design?
20. Why is abstraction important in designing software systems?

- Questions on Functional Independence

21. What is meant by functional independence in software design?
22. How does functional independence contribute to error isolation and code reuse?