Software Engineering (CSE3004)
High level and detailed Design

Puneet Kumar Jain

CSE Department

National Institute of Technology Rourkela

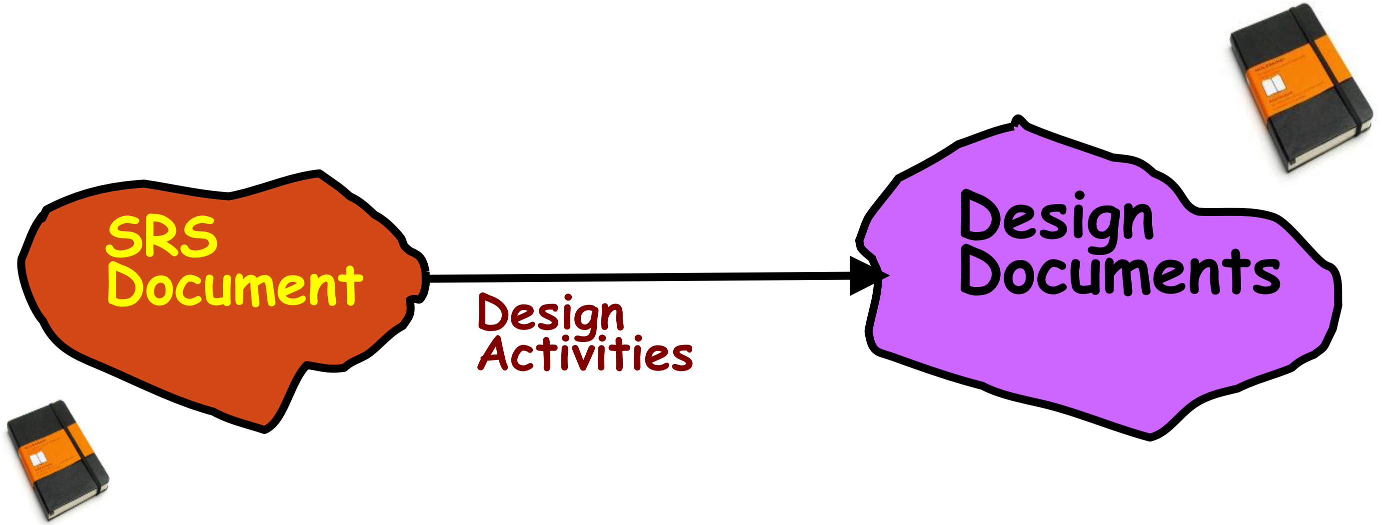
Reference

- Rajib Mall, Introduction to Software Engineering

- Reference to his video lecture: https://www.youtube.com/watch?v=l9XFipXoJb0&list=PLbRMhDVUMngf8oZR3DpKMvYhZKga90JVt&index=20

What is Achieved during design phase?

- Transformation of SRS document to Design document:
  - A form easily implementable in some programming language.



Items Designed During Design Phase

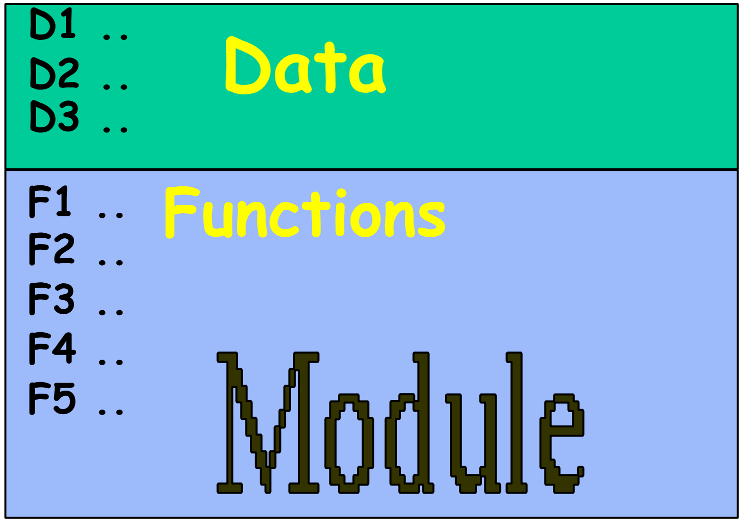
- Module structure,
- Control relationship among the modules
  - call relationship or invocation relationship
- Interface among different modules,
  - data items exchanged among different modules,
- Data structures of individual modules,
- Algorithms for individual modules.

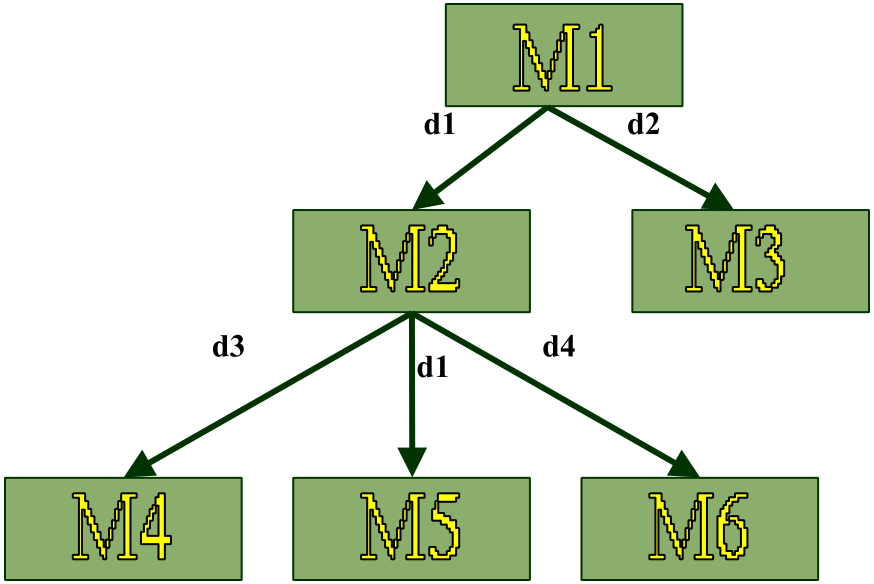
Stages in Design

- Design activities are usually classified into two stages:
  - **Preliminary (or high-level) design**
  - **Detailed design.**

High-level design

- Identify:
  1. modules
  2. control relationships among modules
  3. interfaces among modules.



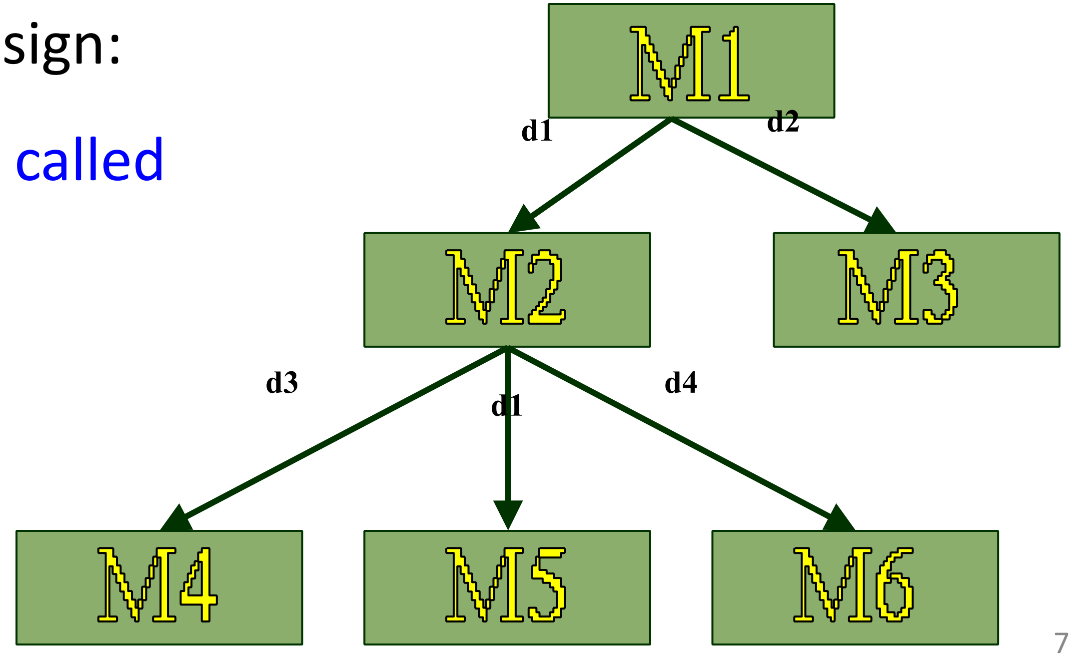


High-level Design

- Several notations are available to represent high-level design:
  - Usually a tree-like diagram called **structure chart** is used.
  - Other notations:
    - Jackson diagram or Warnier-Orr diagram can also be used.

The outcome of high-level design:

program structure, also called software architecture.



Detailed design

- For each module, design for it:
  1. data structure
  2. algorithms

- Outcome of detailed design:
  - module specification.

A fundamental question

- There is no unique way to design a software.

- How to distinguish between good and bad designs?
  - Unless we know what a good software design is:
    - we can not possibly design one.

Good and bad designs

- Different engineers can arrive at very different designs.

Need to determine which is a better design.

What Is a Good Software Design?

- Should implement all functionalities of the system correctly.
- Should be efficient.
- Should be easily amenable to change,
  - i.e. easily maintainable.
- **Should be easily understandable.**
  - Understandability of a design is a major issue:
    - a design that is easy to understand also easy to maintain and change.
  - If the software is not easy to understand:
    - maintenance effort would increase many times.

How to Improve Understandability?

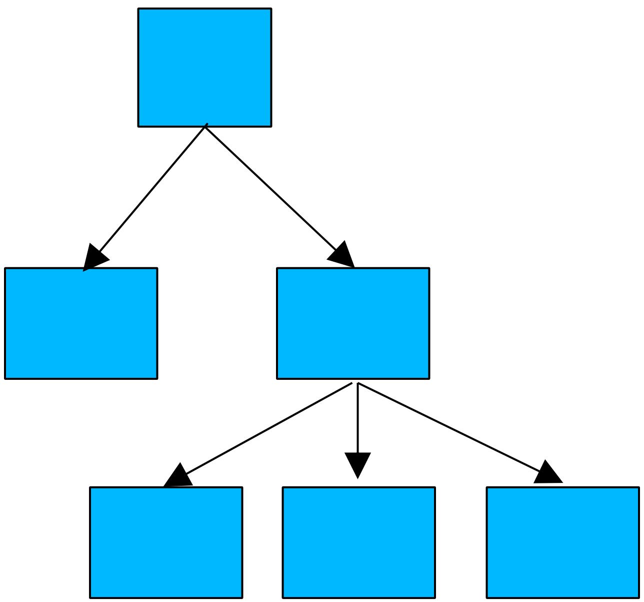
- Use consistent and meaningful names for various design components,

- Design solution should consist of:
  - A set of well decomposed modules (**modularity**),

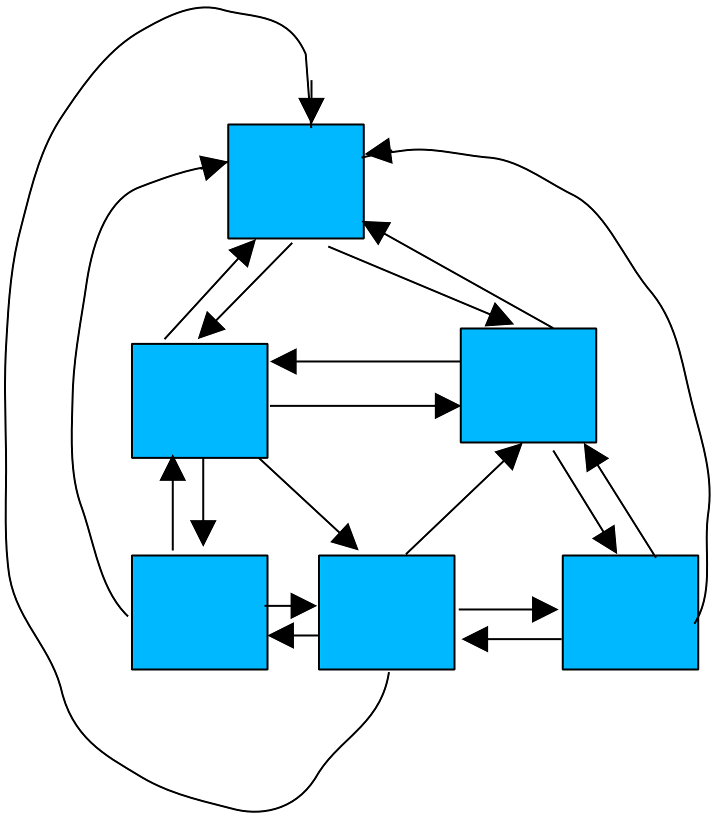
- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem into a clean set of modules:
  - Based on **divide and conquer principle.**

Layering

- Different modules should be neatly arranged in a hierarchy:
  - A tree-like diagram.
  - Called Layering



Superior



Inferior

Modularity

- Arrangement of modules in a hierarchy ensures:
  - **Low fan-out**
  - **Abstraction**

- In technical terms, modules should display:
  - **high cohesion**
  - **low coupling.**

- We next discuss:
  - **cohesion and coupling**.

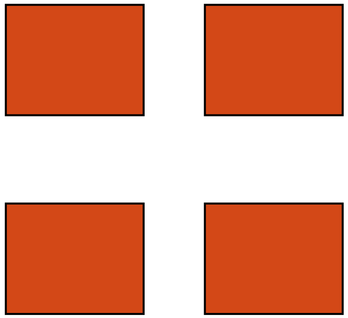
Cohesion and Coupling

- 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 interdependence or interaction between the two modules.**

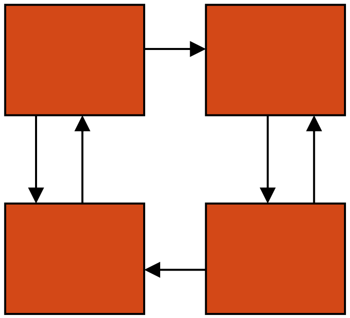
Cohesion and Coupling

- A module having **high cohesion and low coupling**:
  - **Called functionally independent** of other modules:
    - A functionally independent module needs very little help from other modules and therefore has minimal interaction with other modules.

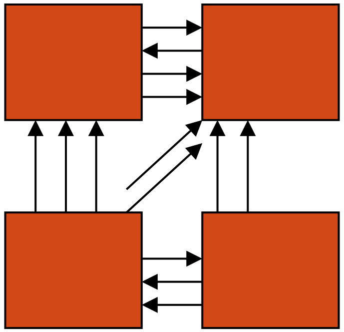
Coupling: Degree of dependence among components



No dependencies



Loosely coupled-some dependencies



Highly coupled-many dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.

Source:

Pfleeger, S., *Software Engineering Theory and Practice*. Prentice Hall, 2001.

Advantages of Functional Independence

- Better understandability
- Complexity of design is reduced,
- Different modules easily understood in isolation:
  - Modules are independent
- Functional independence **reduces error propagation**.
  - degree of interaction between modules is low.
  - an error existing in one module does not directly affect other modules.
- **Also: Reuse of modules is possible.**
  - can be easily taken out and reused in a different program

Measuring Functional Independence

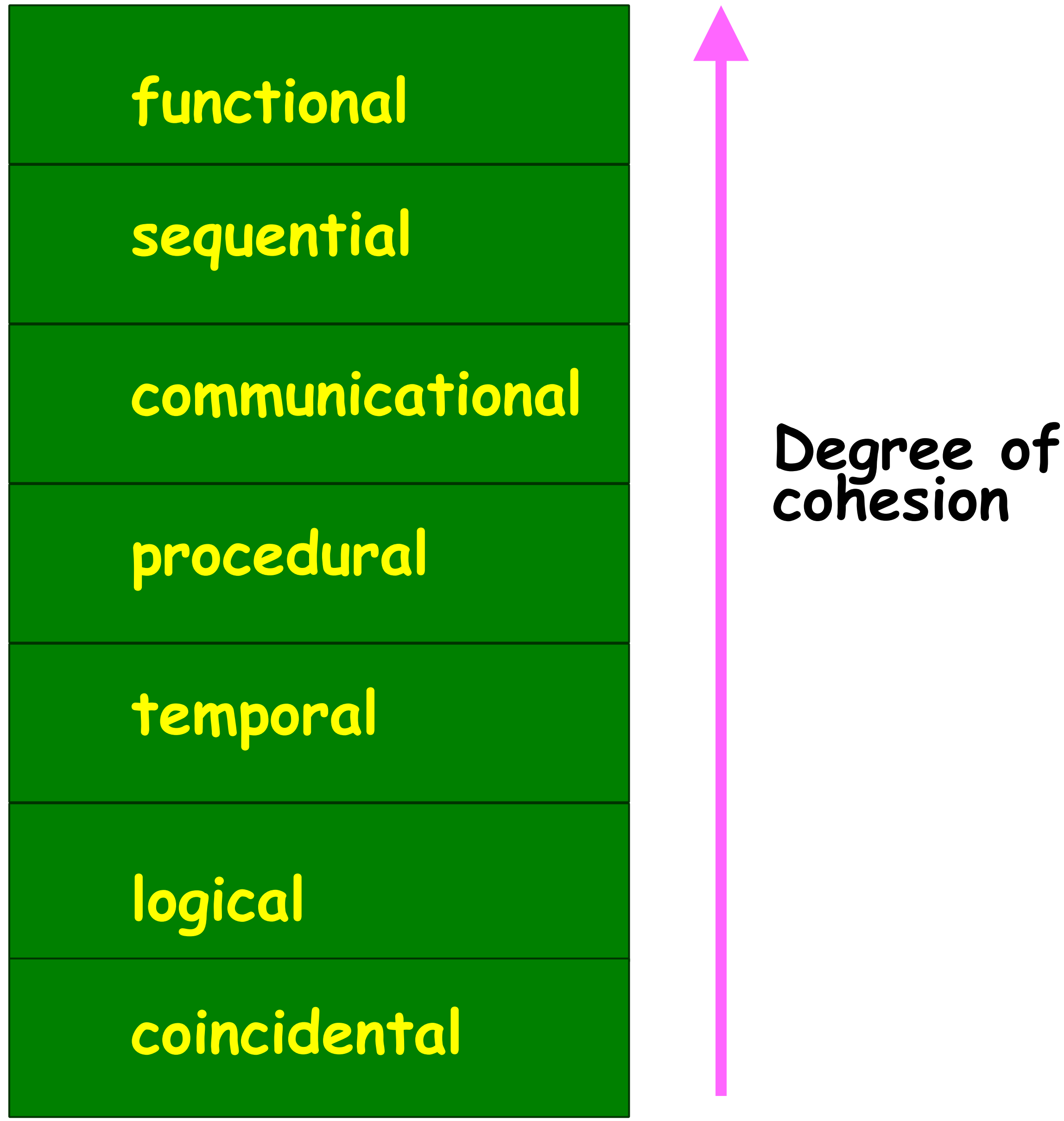
- Unfortunately, there are no ways:
  - to quantitatively measure the degree of cohesion and coupling:
  - At least classification of different kinds of cohesion and coupling:
    - will give us some idea regarding the degree of cohesiveness of a module.

Classification of Cohesiveness

- Classification can have scope for ambiguity:
  - yet gives us some idea about cohesiveness of a module.

- By examining the type of cohesion exhibited by a module:
  - we can roughly tell whether it displays high cohesion or low cohesion.

Classification of Cohesiveness



Coincidental cohesion

- The module performs a set of tasks:
  - which relate to each other very loosely, if at all.
    - That is, the module contains a random collection of functions.
    - **functions have been put in the module out of pure coincidence without any thought or design.**

```
Module AAA{
        Print-inventory();
        Register-Student();
        Issue-Book();
};
```

Logical cohesion

- **All elements of the module perform similar operations:**
  - e.g. error handling, data input, data output, etc.

- An example of logical cohesion:
  - a set of print functions to generate an output report arranged into a single module.

```
module print{
    void print-grades(student-file){ …}
    void print-certificates(student-file){…}
    void print-salary(teacher-file){…}
}
```

Temporal cohesion

- The module contains functions so that:
  - **all the functions must be executed in the same time span.**

- Example:
  - The set of functions responsible for
    - initialization,
    - start-up, shut-down of some process, etc.

```
init() {
        Check-memory();
        Check-Hard-disk();
        Initialize-Ports();
        Display-Login-Screen();
}
```

Procedural cohesion

- The set of functions of the module:
  - all part of a procedure (algorithm)
  - certain sequence of steps have to be carried out in a certain order for achieving an objective,

- **Elements of a component are related only to ensure a particular order of execution**

- **Actions are still weakly connected and unlikely to be reusable**

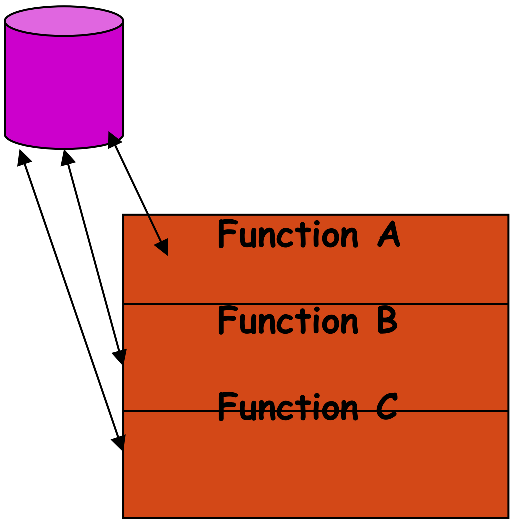
- Example:
  - …
  - Wrirte output record
  - Read new input record
  - Pad input with spaces
  - Return new record
  - …

Communicational cohesion

- All functions of the module:
  - Reference or update the same data structure,

- **Example:**
  - The set of functions defined on an array or a stack.

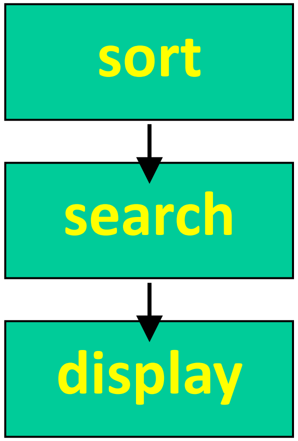
```
handle-Student- Data() {
        Static Struct  Student-data[10000];
        Store-student-data();
        Search-Student-data();
        Print-all-students();
};
```



Communicational
Access same data

Sequential cohesion

- Elements of a module form different parts of a sequence,
  - output from one element of the sequence is input to the next.
  - Example:



Functional cohesion

- Different elements of a module cooperate:
  - to achieve a single function,
  - e.g. managing an employee's pay-roll.

- When a module displays functional cohesion,
  - **we can describe the function using a single sentence.**

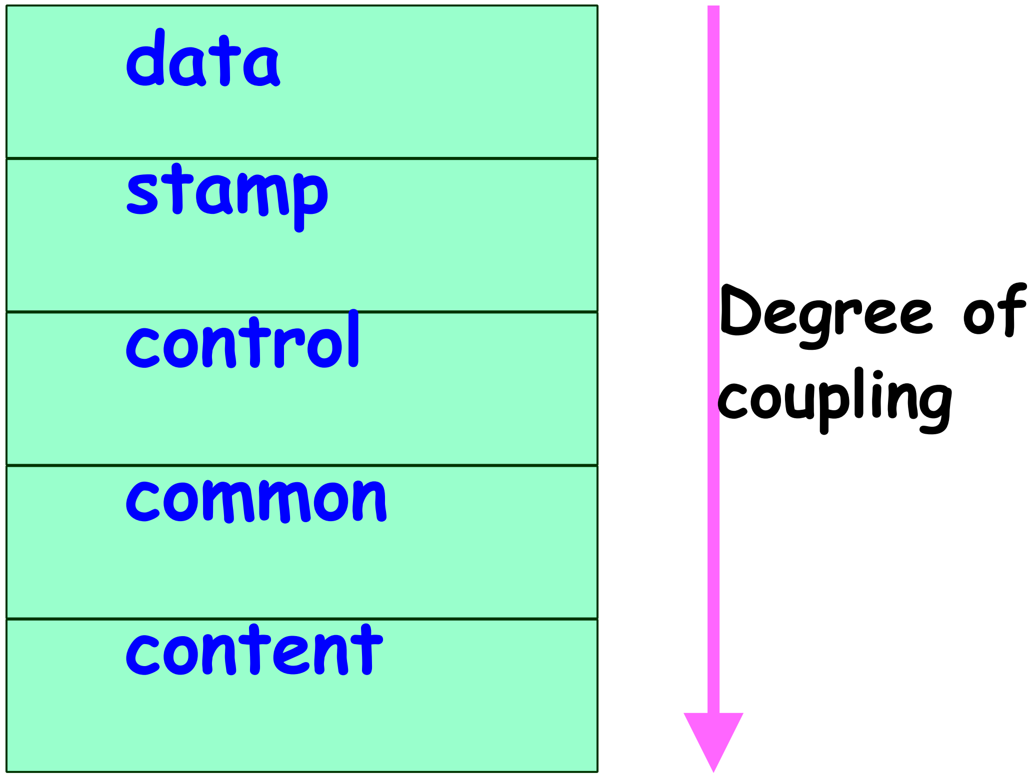
Determining Cohesiveness

- Write down a sentence to describe the function of the module
  - If the sentence is compound,
    - it has a sequential or communicational cohesion.
  - If it has words like “first”, “next”, “after”, “then”, etc.
    - it has sequential or temporal cohesion.
  - If it has words like initialize,
    - it probably has temporal cohesion.

Coupling

- Coupling indicates:
  - how closely two modules interact or how interdependent they are.
  - **The degree of coupling between two modules depends on their interface complexity.**
- There are no ways to precisely measure coupling between two modules:
  - **classification of different types of coupling will help us to approximately estimate the degree of coupling between two modules.**
- Five types of coupling can exist between any two modules.

Classes of coupling



Data coupling

- Two modules are data coupled,
  - if they communicate via a parameter:
    - an elementary data item,
    - e.g an integer, a float, a character, etc.
  - The data item should be problem related:
    - not used for control purpose.

Stamp coupling

- Two modules are stamp coupled,
  - if they communicate via a composite data item
    - or an array or structure in C.
  - Requires second modules to know how to manipulate the data structure

Control coupling

- Data from one module is used to direct
  - order of instruction execution in another.
  - Module passes control parameters to another module

- Example of control coupling:
  - a flag set in one module and tested in another module.

Common Coupling

- Two modules are common coupled,
  - if they share some global data.

  - Usually a poor design choice because:
    - Lack of clear responsibility for the data
    - Reduces readability
    - Difficult to determine all the modules which modifies data elements (Reduces maintainability)
    - Reduces the ability to control data access

Content coupling

- Content coupling exists between two modules:
  - if they share code,
  - One module directly modifies another module's data
  - e.g, branching from one module into another module.
- The degree of coupling increases
  - from data coupling to content coupling.

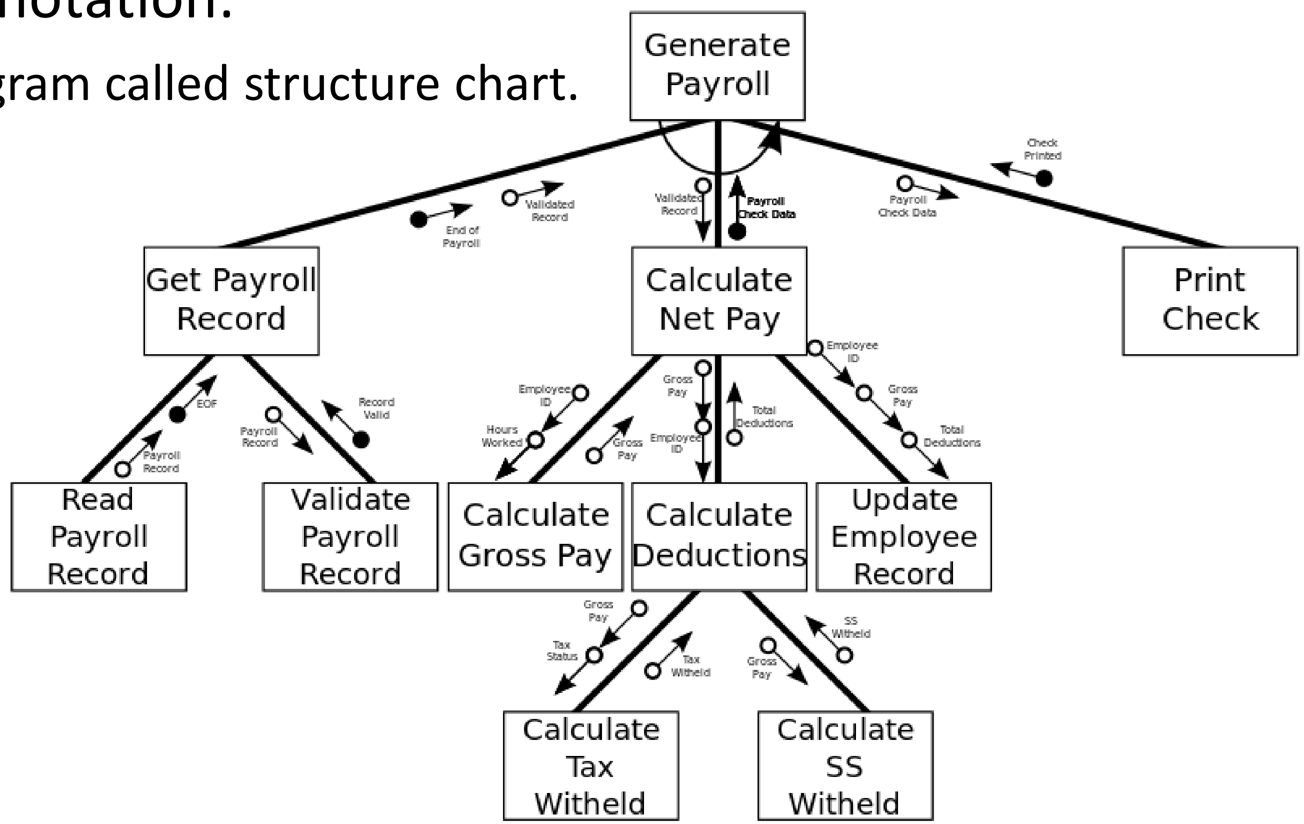
- **Discussed about cohesion and coupling to describe a designing**

Now, we will discussed other aspects of a good design

Hierarchical Design

- Control hierarchy represents:
  - organization of modules.
  - control hierarchy is also called program structure.

- Most common notation:
  - a tree-like diagram called structure chart.



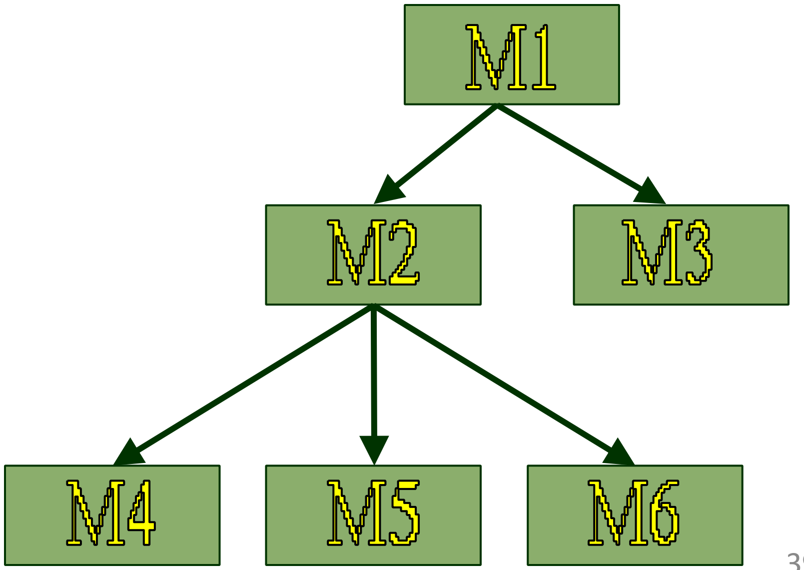
Ref: https://en.wikipedia.org/wiki/File:CPT-Structured_Chart_Example.svg

Control Relationship terminology

- Some terminologies regarding the control structure

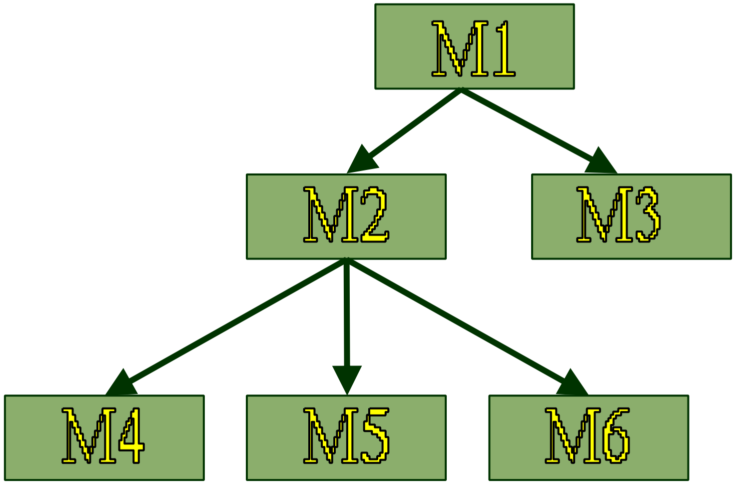
- **Superordinate:** A module that controls another module said to be to the superordinate later module.

- **Subordinate:** Conversely, a module controlled by another module said to be subordinate to the later module.



Control Relationship terminology

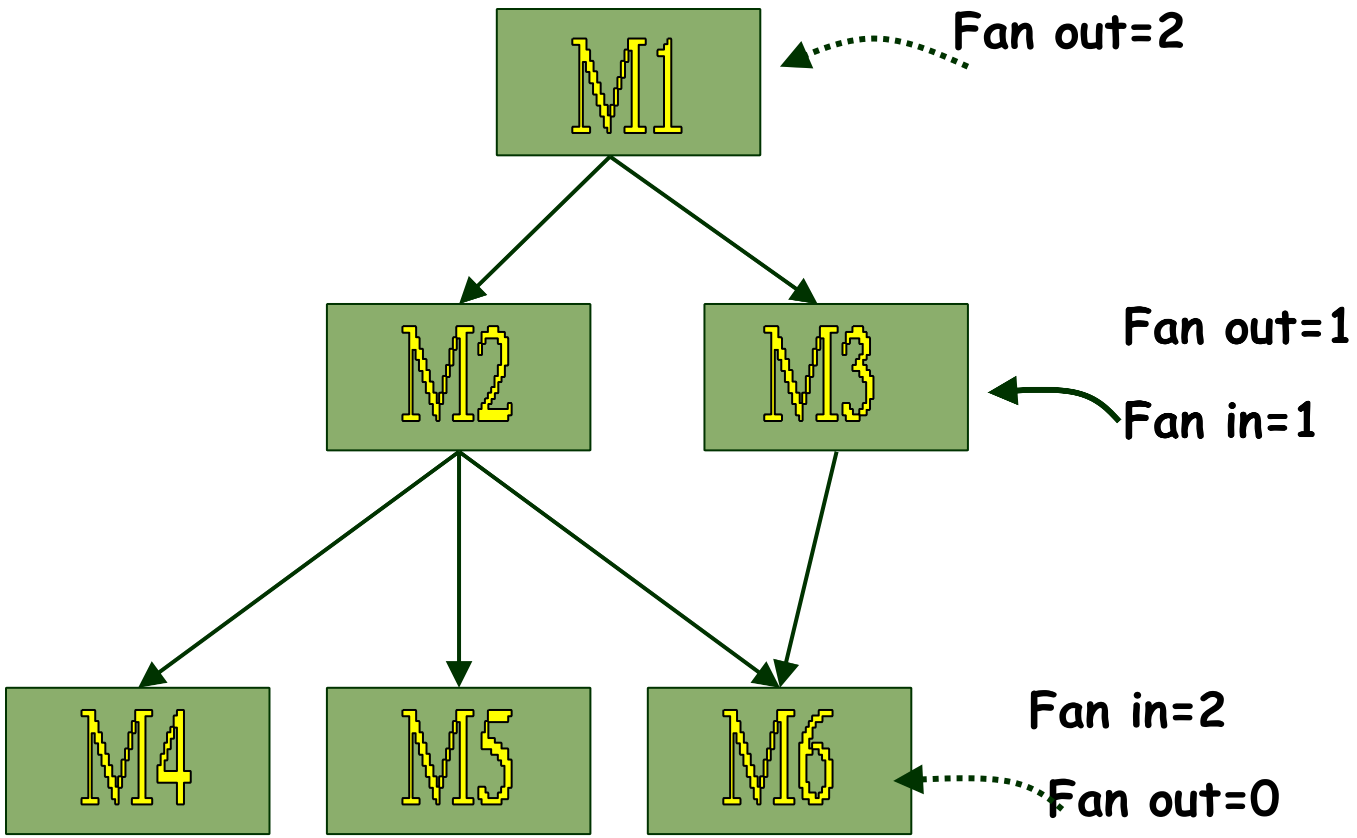
- **Visible modules:** a module ***A*** is said to be visible by another module B,
  - if A directly or indirectly calls B.
- **Layering:** Layering principle requires:
  - modules at a layer can call only the modules immediately below it.



Control Relationship terminology

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

Example: Module Structure

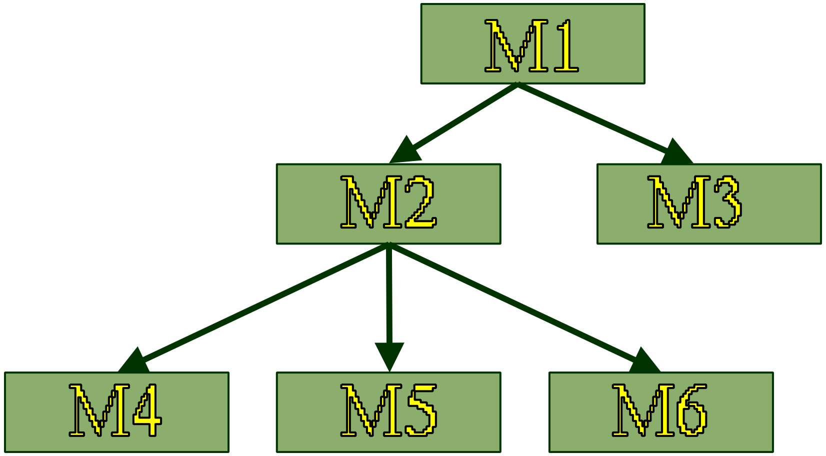


Good Hierarchical Arrangement of modules

- How to say a designed control structure is good or bad
  - Characteristic of control structure
    - **Low fan out**
    - **High Fan in**
    - **Layering**
    - **abstraction**

Goodness of Design

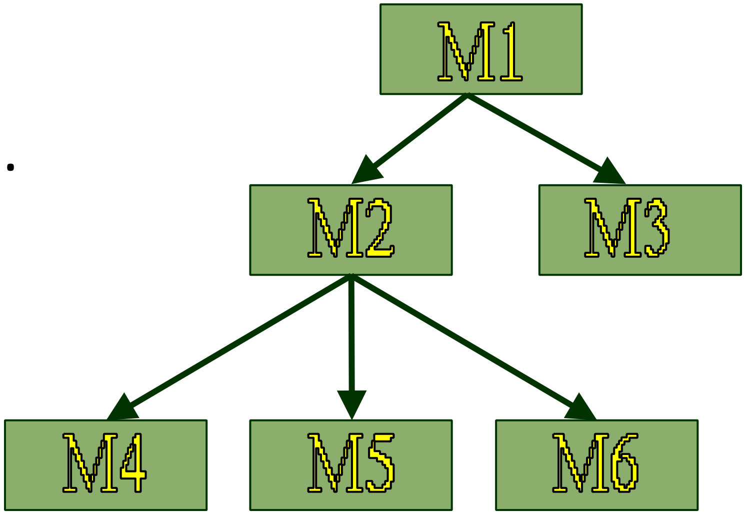
- A design having modules:
  - With **High fan-out** number
    - not a good design.
    - a module having high fan-out mean it invokes a large number of other modules and likely to implement several different functions:
    - Thus lacks cohesion.
  - High fan-in represents code reuse and is in general encouraged.



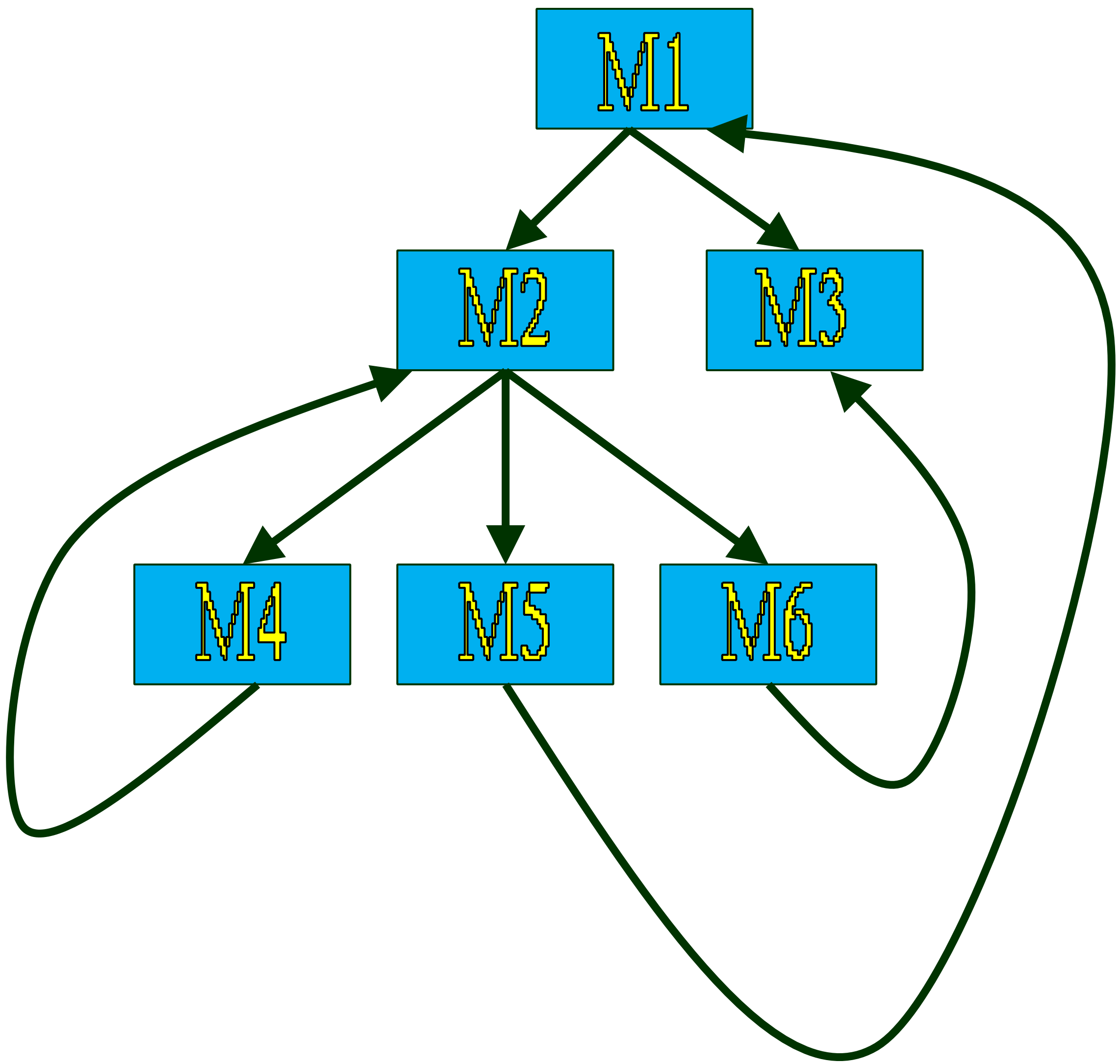
Abstraction

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

- **Lower-level modules:**
  - Perform input/output and other low-level functions.
- **Upper-level modules:**
  - Perform more managerial functions.



Bad Design



Design Approaches

- Two fundamentally different software design approaches:
  - **Function-oriented design**
  - **Object-oriented design**

- These two design approaches are radically different.
  - However, are complementary rather than competing techniques.
  - Each technique is applicable at different stages of the design process.

Function-Oriented Design

- A system is looked upon as something
  - that performs a set of functions. (**Structure analysis**)

- Starting at this high-level view of the system:
  - each function is successively refined into more detailed functions (top-down decomposition).
  - Functions are mapped to a module structure. **(Structured design)**

Example

- The function create-new-library- member:
  - creates the record for a new member,
  - assigns a unique membership number
  - prints a bill towards the membership

Function-Oriented Design

- Several function-oriented design approaches have been developed:
  - Structured design (Constantine and Yourdon, 1979)
  - Jackson's structured design (Jackson, 1975)
  - Warnier-Orr methodology
  - Wirth's step-wise refinement
  - **Hatley and Pirbhai's Methodology**

Object-Oriented Design

- System is viewed as a collection of objects (i.e. entities).
- System state is decentralized among the objects:
  - each object manages its own state information.
- **For example:**
- Library Automation Software:
  - each library member is a separate object
    - with its own data and functions.
  - Functions defined for one object cannot directly refer to or change data of other objects.

Object-Oriented Design

- Objects have their own internal data:
  - defines their state.
- Similar objects constitute a class.
  - each object is a member of some class.
- Classes may inherit features
  - from a super class.
- Conceptually, objects communicate by message passing.

Object-Oriented versus Function-Oriented Design

- Unlike function-oriented design,
  - in OOD the basic abstraction is not functions such as "sort", "display", "track", etc.,
  - but real-world entities such as "employee", "picture", "machine", "radar system", etc.

- In OOD:
  - software is not developed by designing functions such as:
    - update-employee-record,
    - get-employee-address, etc.
  - but by designing objects such as:
    - employees,
    - departments, etc.

Object-Oriented versus Function-Oriented Design

- Grady Booch sums up this fundamental difference saying:
  - **"Identify verbs if you are after procedural design and nouns if you are after object-oriented design."**

Object-Oriented versus Function-Oriented Design

- In OOD:
  - state information is not shared in a centralized data.
  - but is distributed among the objects of the system.

Object-Oriented versus Function-Oriented Design

- Objects communicate by message passing.
  - one object may discover the state information of another object by interrogating it.

- Of course, somewhere or other the functions must be implemented:
  - the functions are usually associated with specific real-world entities (objects)
  - directly access only part of the system state information.

Object-Oriented versus Function-Oriented Design

- Function-oriented techniques 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.

Object-Oriented versus Function-Oriented Design

- To illustrate the differences between object-oriented and function-oriented design approaches,
    - let us consider an example ---

An automated fire-alarm system for a large building.

Fire-Alarm System

- We need to develop a computerized fire alarm system for a large multi-storied building:
  - There are 80 floors and 2000 rooms in the building.
- Different rooms of the building:
  - fitted with smoke detectors and fire alarms.
- The fire alarm system would monitor:
  - status of the smoke detectors.

Fire-Alarm System

- Whenever a fire condition is reported by any smoke detector:
  - the fire alarm system should:
    - determine the location from which the fire condition was reported
    - sound the alarms in the neighbouring locations.
- The fire alarm system should:
  - flash an alarm message on the computer console:
    - fire fighting personnel manange the console round the clock.
- After a fire condition has been successfully handled,
  - the fire alarm system should let fire fighting personnel reset the alarms.

Function-Oriented Approach:

```
/* Global data (system state) accessible by various functions */
BOOL        detector_status[2000];
int         detector_locs[2000];
BOOL        alarm-status[2000]; /* alarm activated when  set */
int         alarm_locs[2000]; /* room number where alarm is located */
int         neighbor-alarms[2000][10];/*each detector has at most*/
                                       /* 10 neighboring alarm locations */

interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();
```

Function-Oriented Approach

Object-Oriented Approach:

class detector

attributes: status, location, neighbors

operations: create, sense-status, get-location, find-neighbors

class alarm

attributes: location, status

operations: create, ring-alarm, get_location, reset-alarm

- **Appropriate number of instances of the class detector and alarm are created.**

Object-Oriented Approach

Object-Oriented versus Function-Oriented Design

- In a function-oriented program :
  - the system state is centralized
  - several functions accessing these data are defined.

- In the object oriented program,
  - the state information is distributed among various sensor and alarm objects.

Object-Oriented versus Function-Oriented Design

- Use OOD to design the classes:
  - then applies top-down function oriented techniques to design the internal methods of classes.

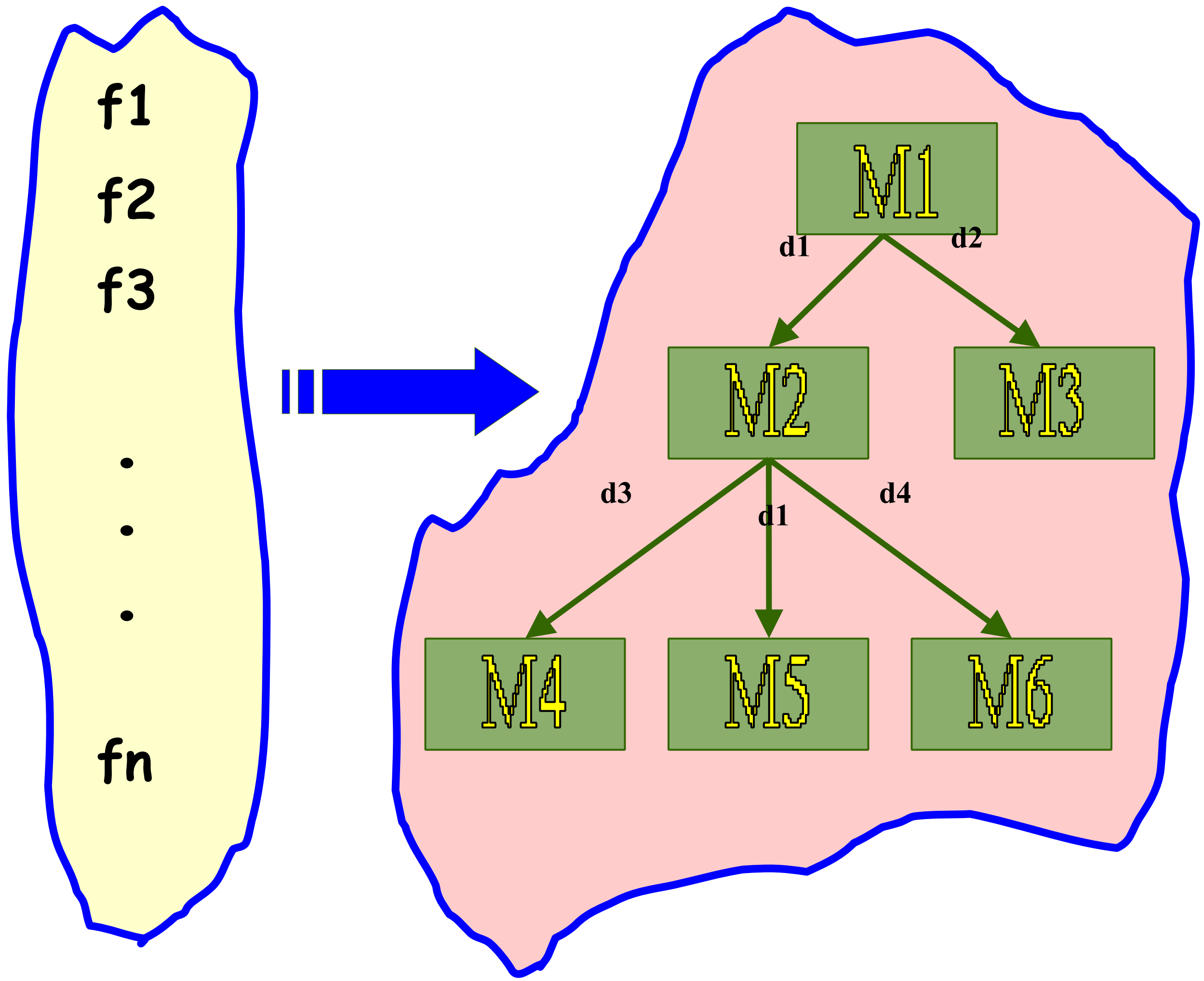
- Though outwardly a system may appear to have been developed in an object oriented fashion,
  - but inside each class there is a small hierarchy of functions designed in a top-down manner.

Function-oriented vs. Object-oriented Design

- **Function-oriented or Procedural**
  - Top-down approach
  - Carried out using **Structured analysis and structured design**
  - Coded using languages such as C

- **Object-oriented**
  - Bottom-up approach
  - Carried out using UML
  - Coded using languages such as Java, C++, C#

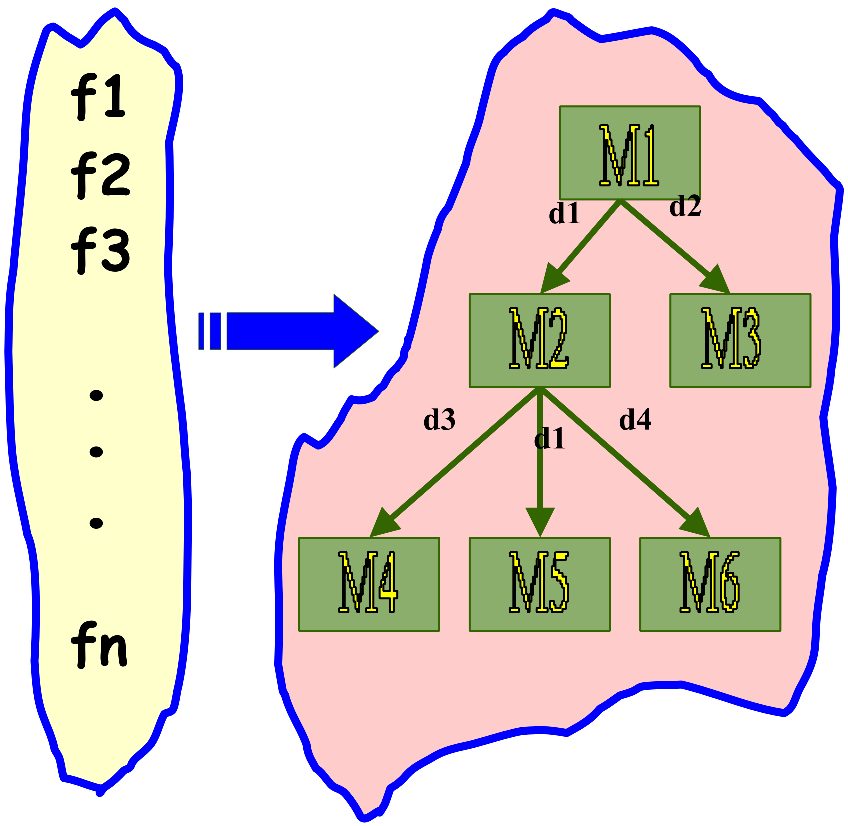
High-level Design



Objective of high level design is to organise the functions in a good control structure

Structured analysis and Structured Design

- During Structured analysis:
  - Capture the detailed structure of the system as the user views it.
  - High-level functions are successively decomposed:
    - Into more detailed functions.
- During Structured design:
  - Arrive at a form that is suitable for implementation in some programming language.
  - The detailed functions are mapped to a module structure.



SA/SD (Structured Analysis/Structured Design)

- SA/SD technique draws heavily from the following methodologies:
  - **Constantine and Yourdon's methodology**
  - **Hatley and Pirbhai's methodology**
  - **Gane and Sarson's methodology**
  - **DeMarco and Yourdon's methodology**

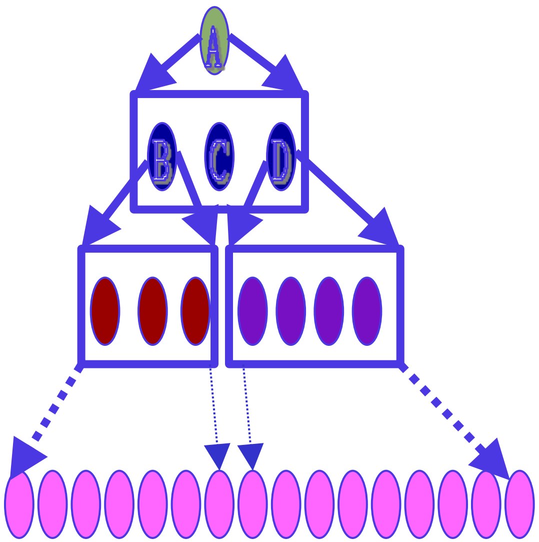
We largely use

- SA/SD technique results in:
  - high-level design.

Structured Analysis

- Successive decomposition of high-level functions:
  - Into more detailed functions.
  - Technically known as **top-down decomposition**.

- Simultaneous decomposition of high-level data Into more detailed data.



- Why model functionalities?
  - **Functional requirements exploration and validation**
  - **Serves as the starting point for design.**

Structured Analysis

- The results of structured analysis can be easily understood even by ordinary customers:
  - **Does not require computer knowledge.**
  - **Directly represents customer's perception of the problem.**
  - **Uses customer's terminology for naming different functions and data.**

- Results of structured analysis:
  - Can be reviewed by customers to check whether it captures all their requirements.

Structured Analysis

- Textual problem description converted into a graphic model.
  - Done using **data flow diagrams (DFDs).**
  - DFD (Data Flow Diagram) is the modelling technique
  - DFD is used to modelled and decomposed functional requirements.
  - DFD graphically represents the results of structured analysis.

Structured Design

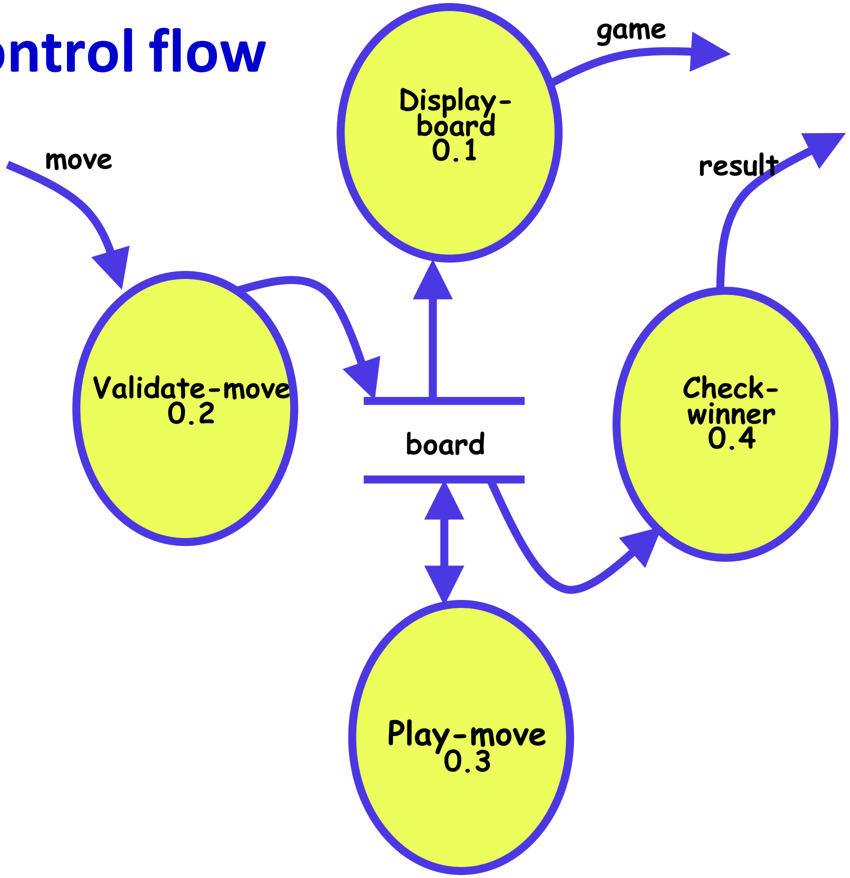
- The functions represented in the DFD:
  - Mapped to a **module structure**.
- Module structure:
  - Also called **software architecture**

Structured Analysis

- Based on principles of:
  - **Top-down decomposition approach.**
  - **Divide and conquer principle:**
    - Each function is considered individually (i.e. isolated from other functions).
    - Decompose functions totally disregarding what happens in other functions.
  - Graphical representation of results using
    - **Data flow diagrams (or bubble charts).**

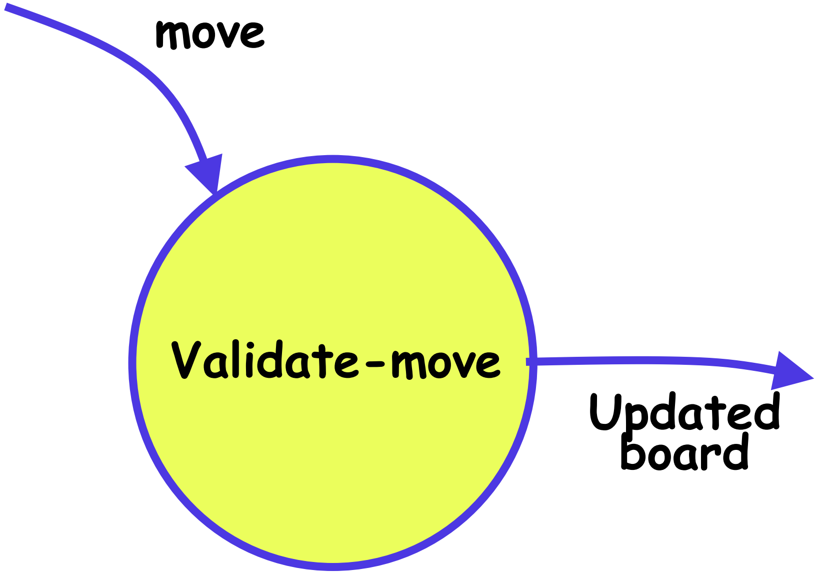
Data Flow Diagram

- DFD is a hierarchical graphical model:
  - Shows the different functions (or processes) of the system
  - Data interchange among the processes.
  - **Represents the data flow not control flow**

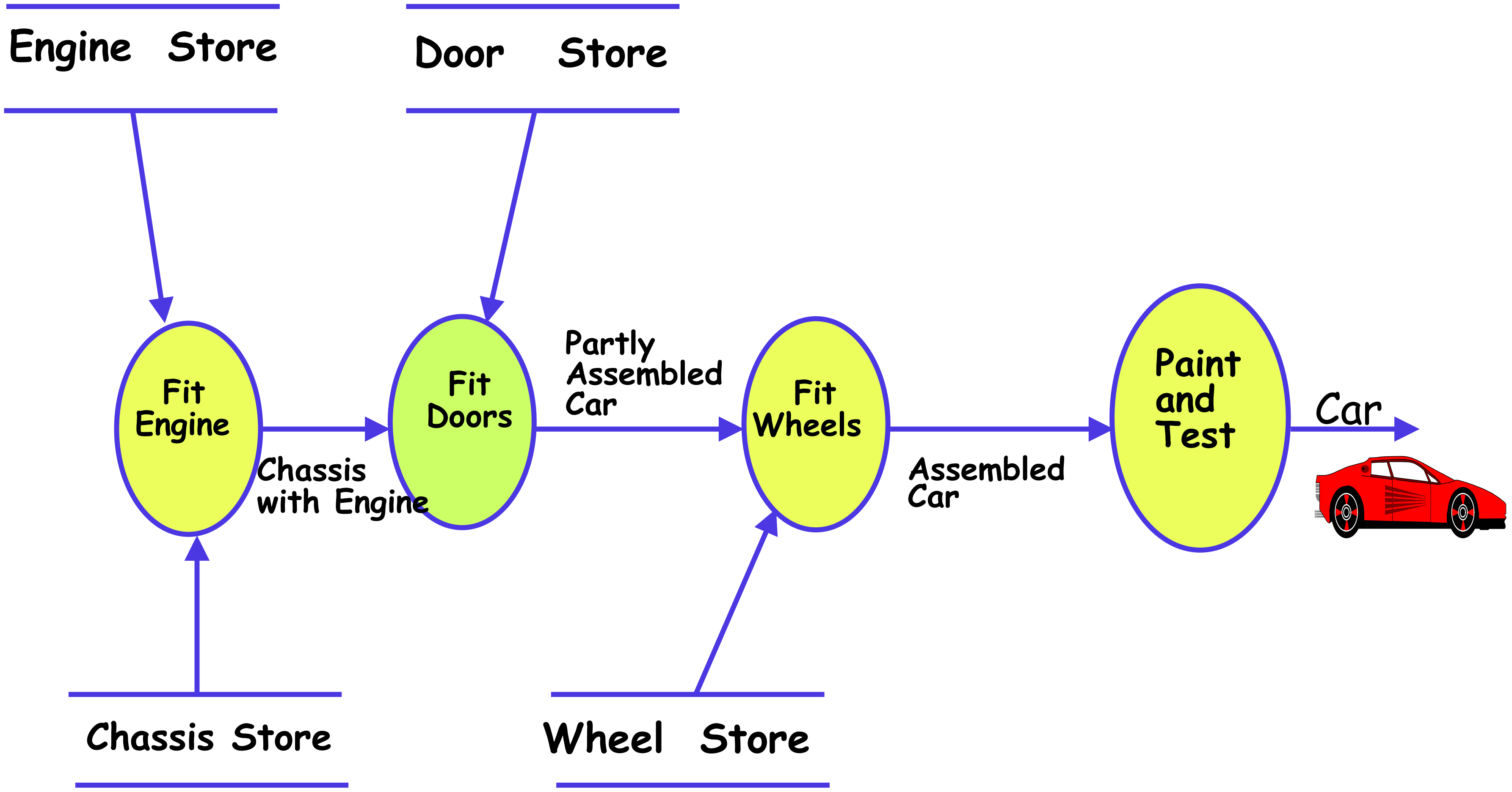


DFD Concepts

- It is useful to consider each function as a processing station:
  - Each function consumes some input data.
  - Produces some output data.



Data Flow Model of a Car Assembly Unit

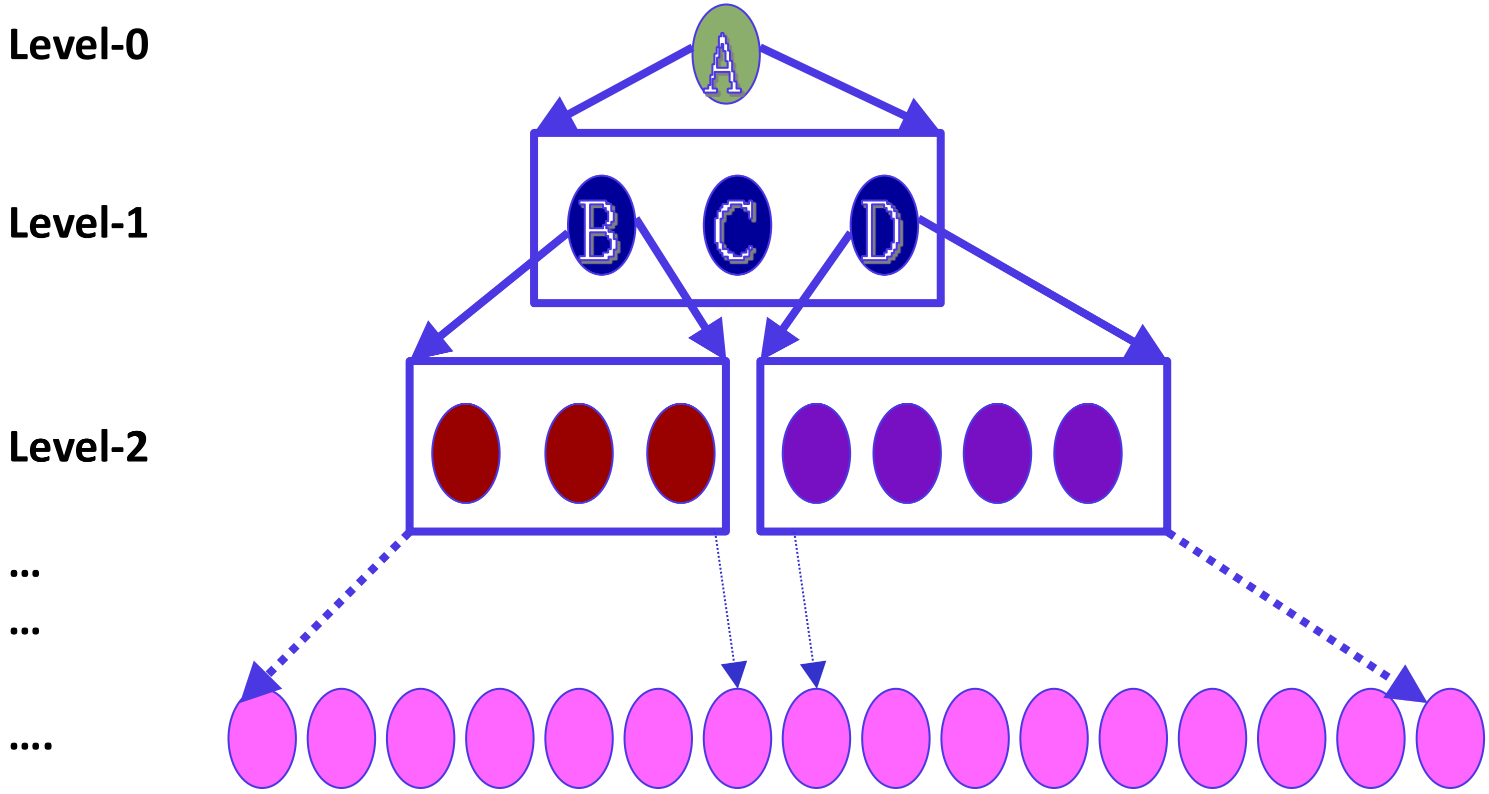


Pros of Data Flow Diagrams (DFDs)

- A DFD model:
  - Uses limited types of symbols.
  - Simple set of rules
  - Easy to understand --- a hierarchical model.

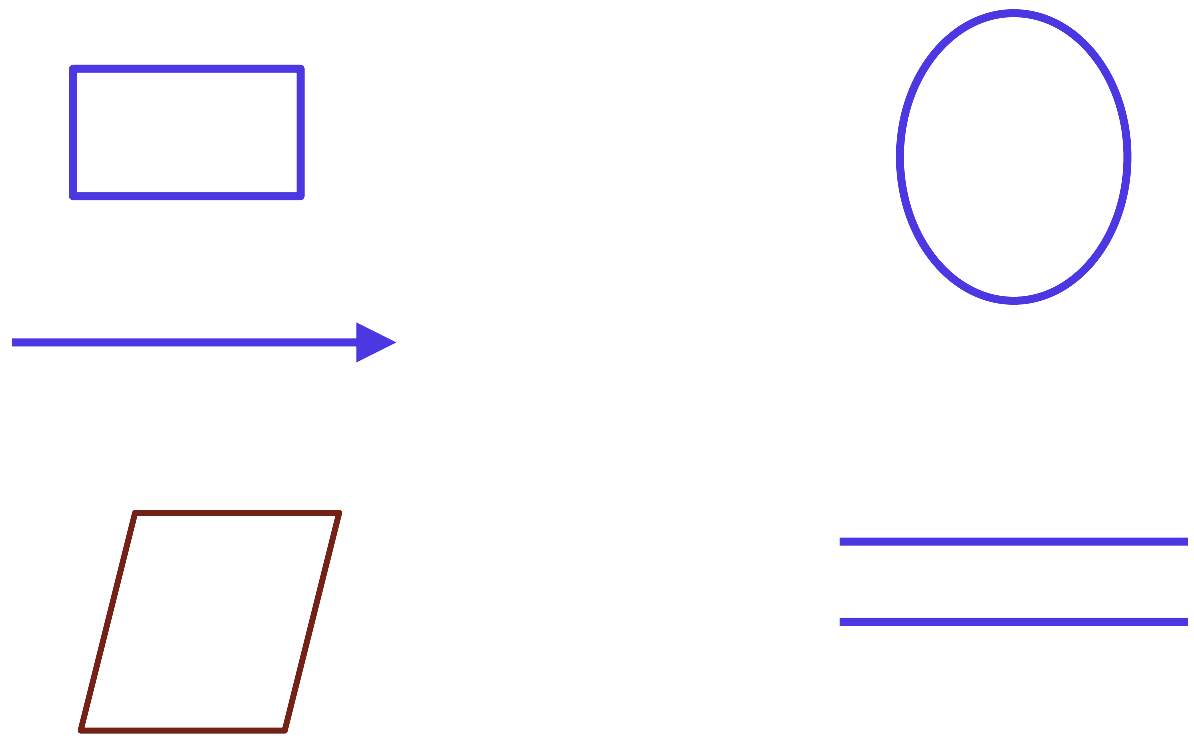
Hierarchical Model

- In a hierarchical model:
  - We start with a very simple and abstract model of a system,
  - Details are slowly introduced through the hierarchies.



Data Flow Diagrams (DFDs)

- Basic Symbols Used for Constructing DFDs:



DFD symbol: rectangle

- Rectangle: external Entity Symbol

Entity_name

- For example: In Library software, librarian is the user

Librarian

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

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.**

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.
  - For example:

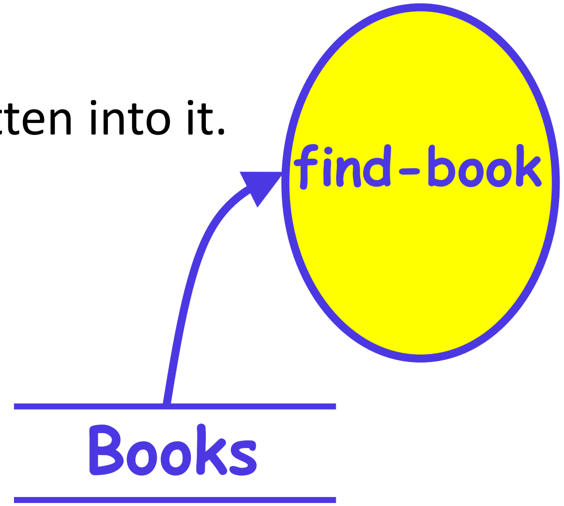
Data Store Symbol

- 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 (not to a external user):
    - By means of a data flow symbol.

Data Store Symbol

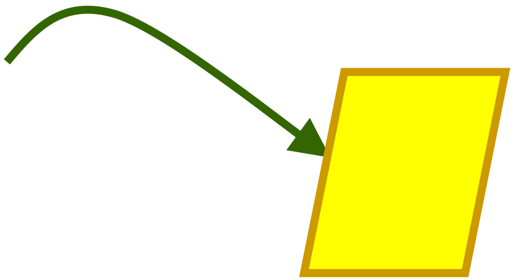
- Direction of data flow arrow:
  - Shows whether data is being read from or written into it.



- 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.
  - In other cases (arrow from process to user) needs annotation

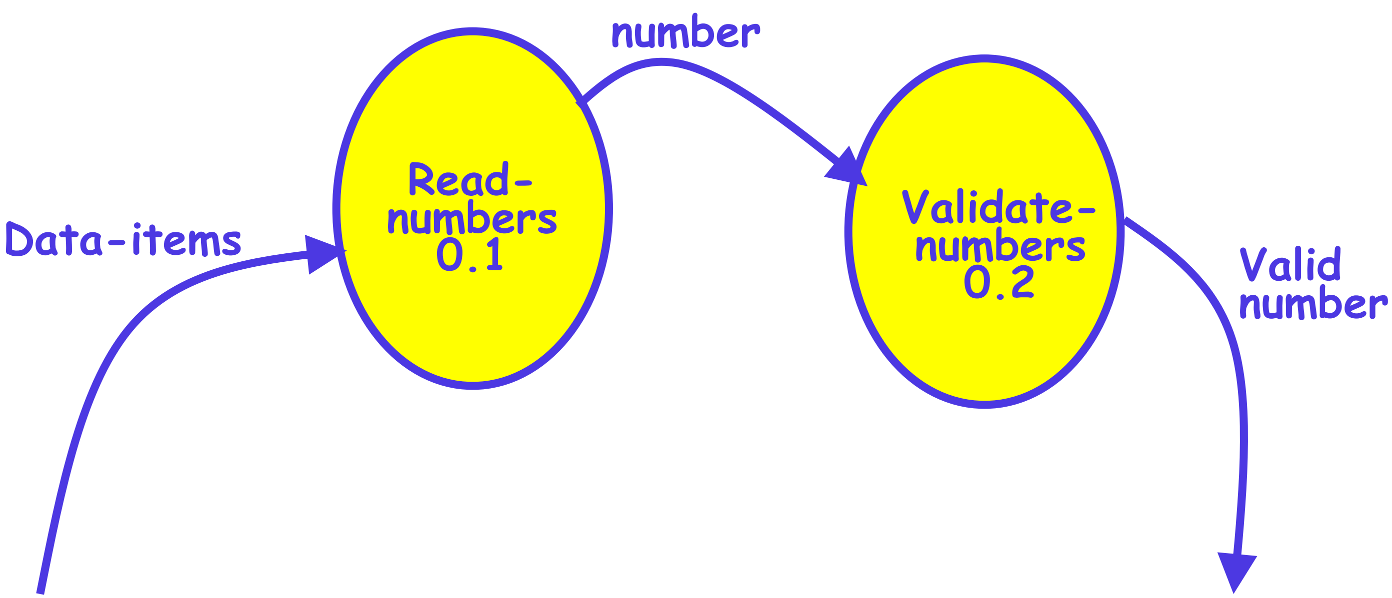
Output Symbol: Parallelogram

- Output produced by the system
  - for example: print-out, display…



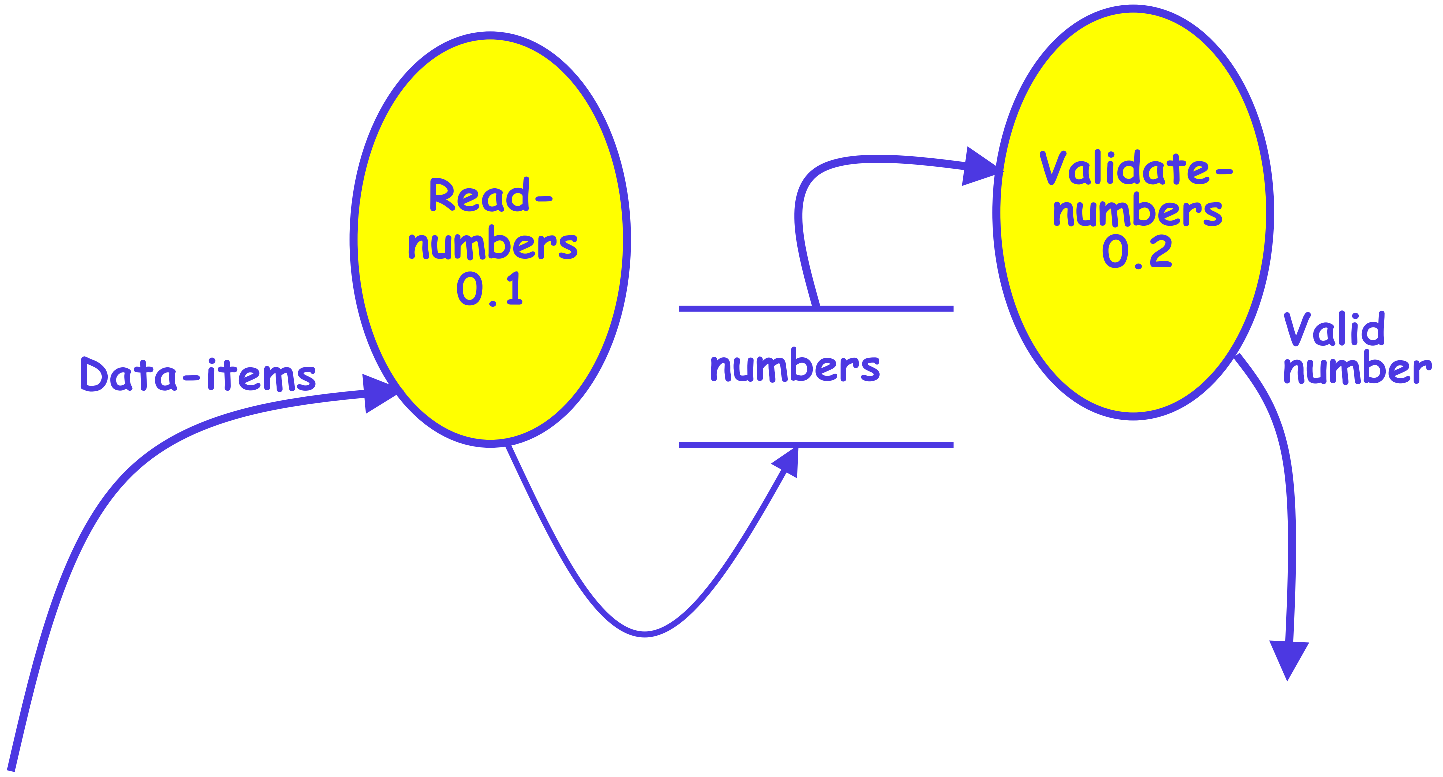
Synchronous Operation

- If two bubbles are directly connected by a data flow arrow:
  - They are synchronous



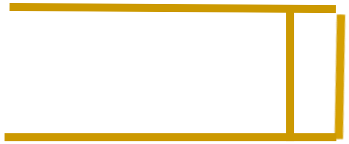
Asynchronous Operation

- If two bubbles are connected via a data store:
  - They are not synchronous.



Yourdon's vs. Gane Sarson Notations

- The notations that we are following:
  - Are closer to the Yourdon's notations
- You may sometimes find notations in books and used in some tools that are slightly different:
  - For example, the data store may look like a box with one end closed

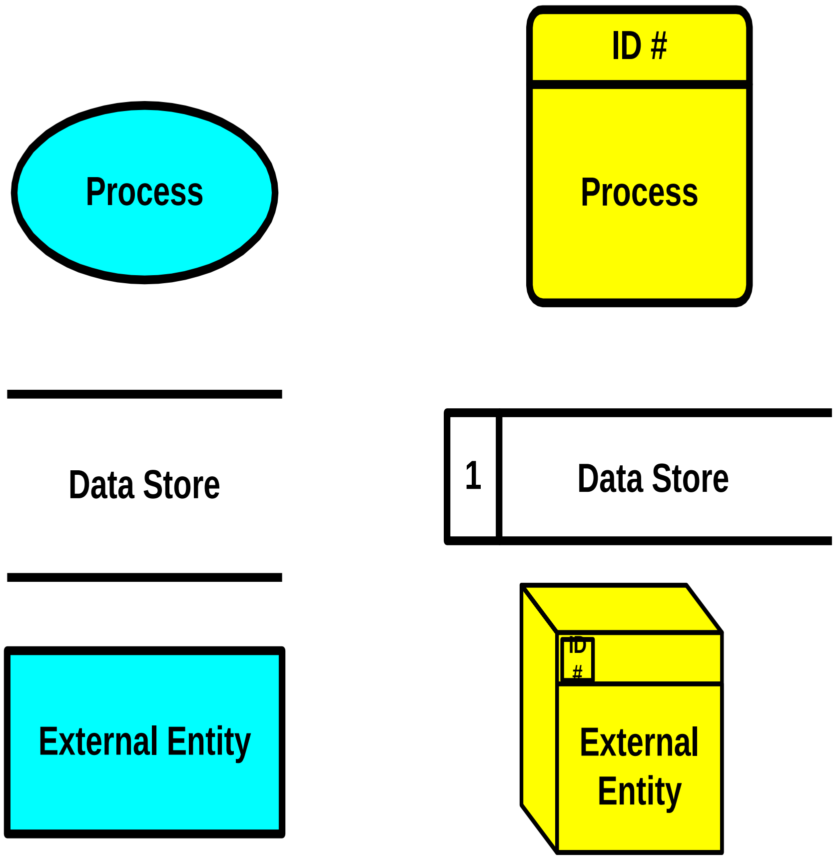


DFD Shapes from Visio

Visio 5.x

From Flow Chart / Data Flow Diagram

From Software Diagram / Gane-Sarson DFD



Visio 2000

Data Flow Diagram

Structured Analysis: Level-0 DFD

- Initially represent the software at the most abstract level:
  - Called the **context diagram.**
  - The entire system is represented as a single bubble labelled according to the main function of the system.

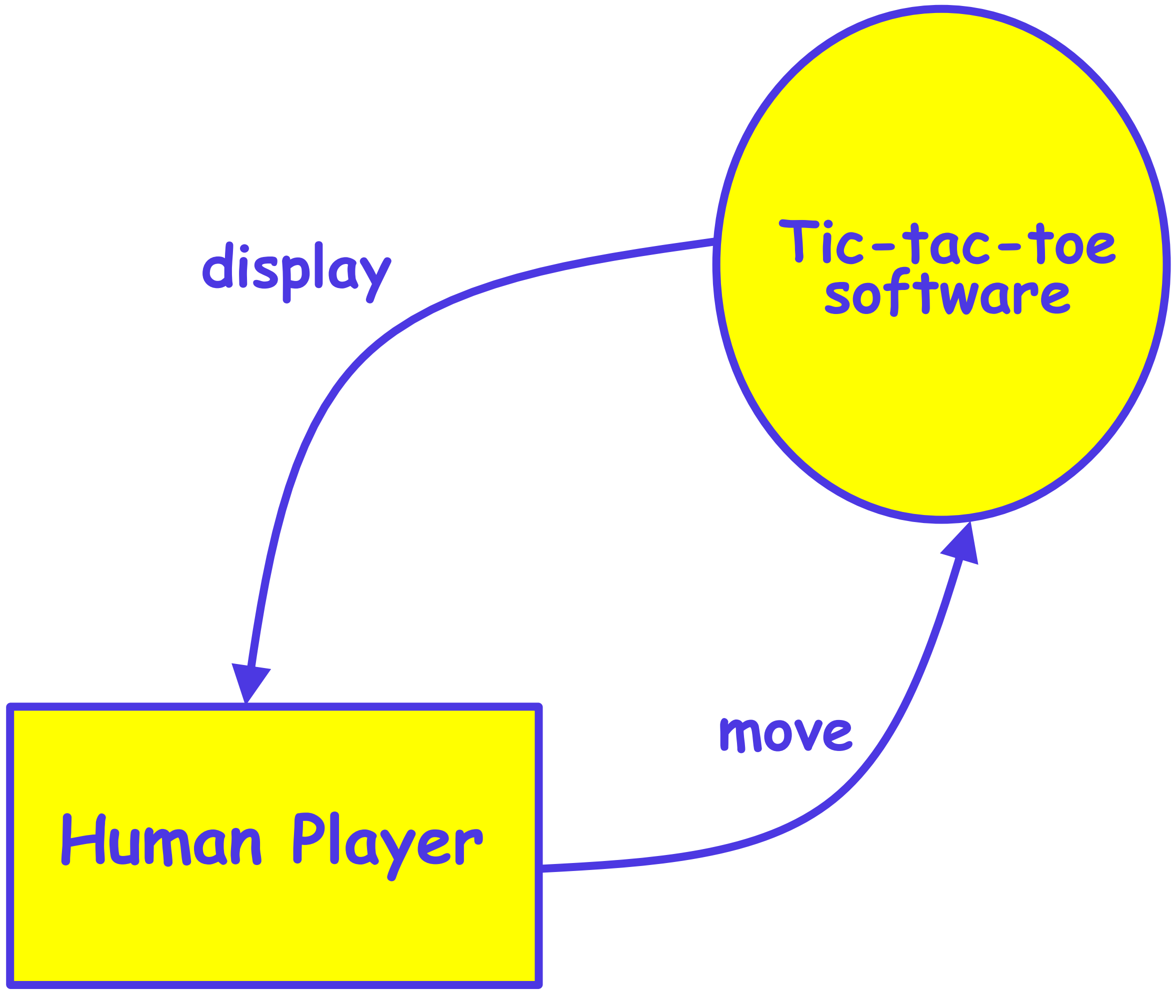
- A context diagram shows:
  - External entities.
  - Data input to the system by the external entities,
  - Output data generated by the system.

- The context diagram is also called the **level 0 DFD.**

Context Diagram

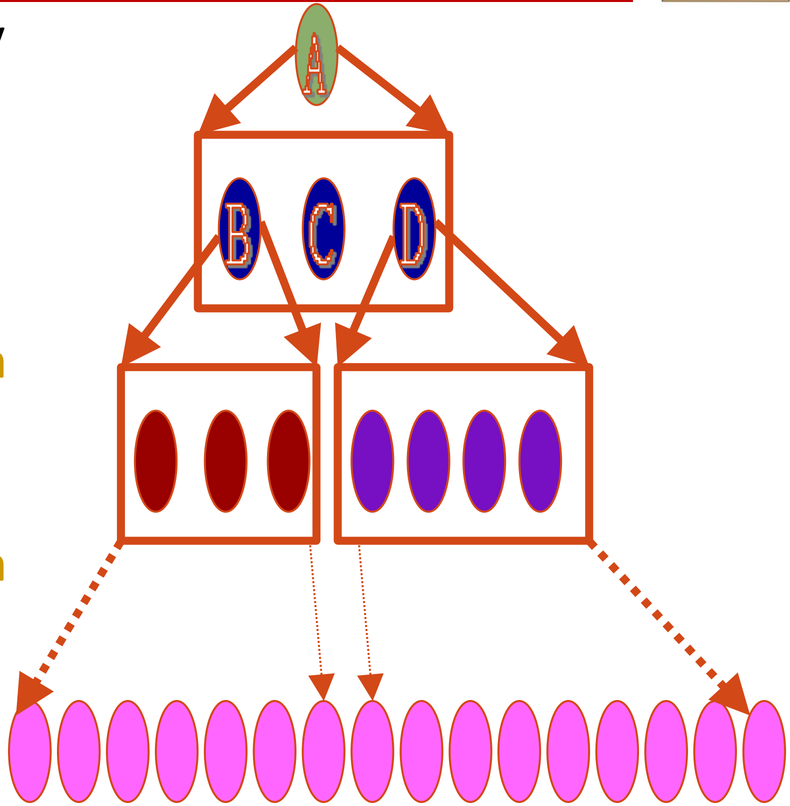
- Establishes the context of the system, i.e.
  - Represents the system level
    - **Data sources**
    - **Data sinks.**

Example1: Tic-tac-toe: Context Diagram



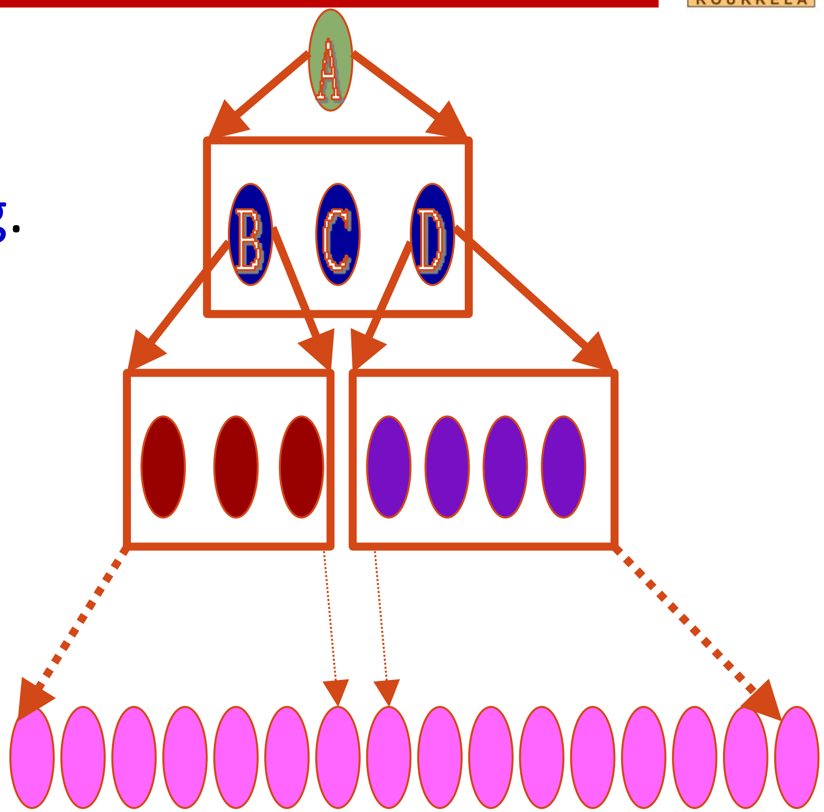
Higher Level DFDs

- Each high-level function is separately decomposed into subfunctions:
  - **Identify the subfunctions of the function**
  - **Identify the data input to each subfunction**
  - **Identify the data output from each subfunction**
- These are represented as DFDs.



Decomposition

- Decomposition of a bubble:
  - Also called factoring or exploding.



Decomposition Pitfall

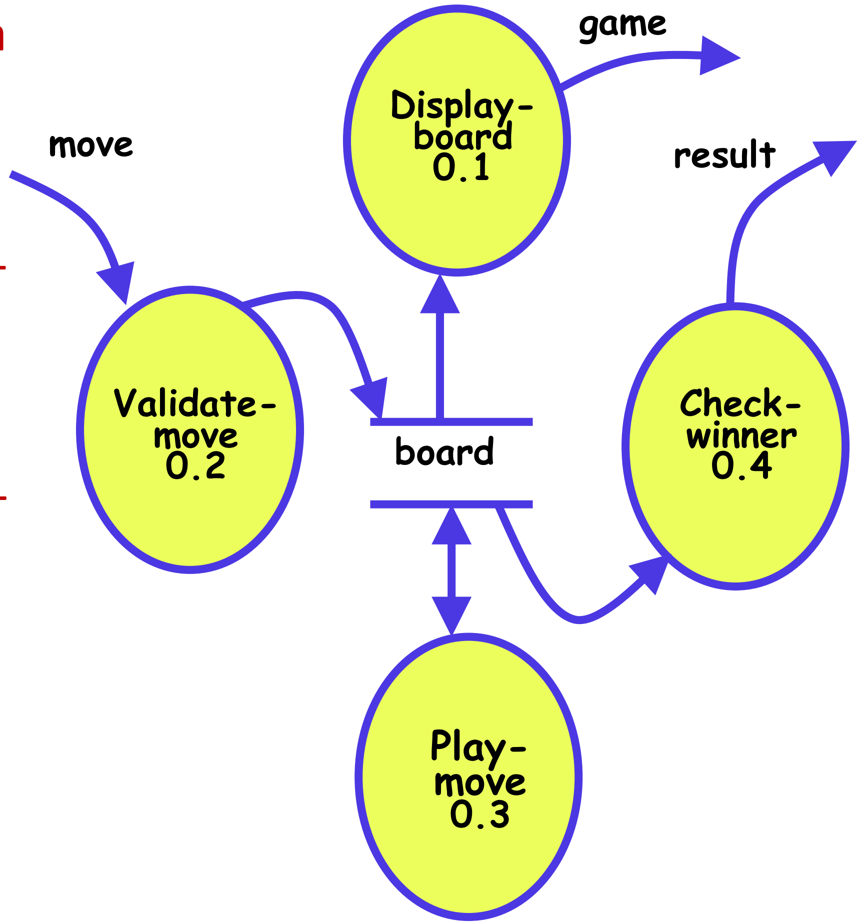
- Each bubble should be decomposed into
  - **Between 3 to 7 bubbles.**
  - **Too few bubbles(just one or two) make decomposition superfluous:**
- Too many bubbles at a level, a sign of poor modelling:
  - **More than 7 bubbles at any level of a DFD.**
  - **Make the DFD model hard to understand.**

Decompose How Long?

- Decomposition of a bubble should be carried on until:
  - A level at which the function of the bubble can be described using a simple algorithm.

Example1: Level 1 DFD Construction

- Examine the SRS document:
  - Represent each high-level function as a bubble.
  - Represent data input to every high-level function.
  - Represent data output from every high-level function.



Tic-tac-toe example

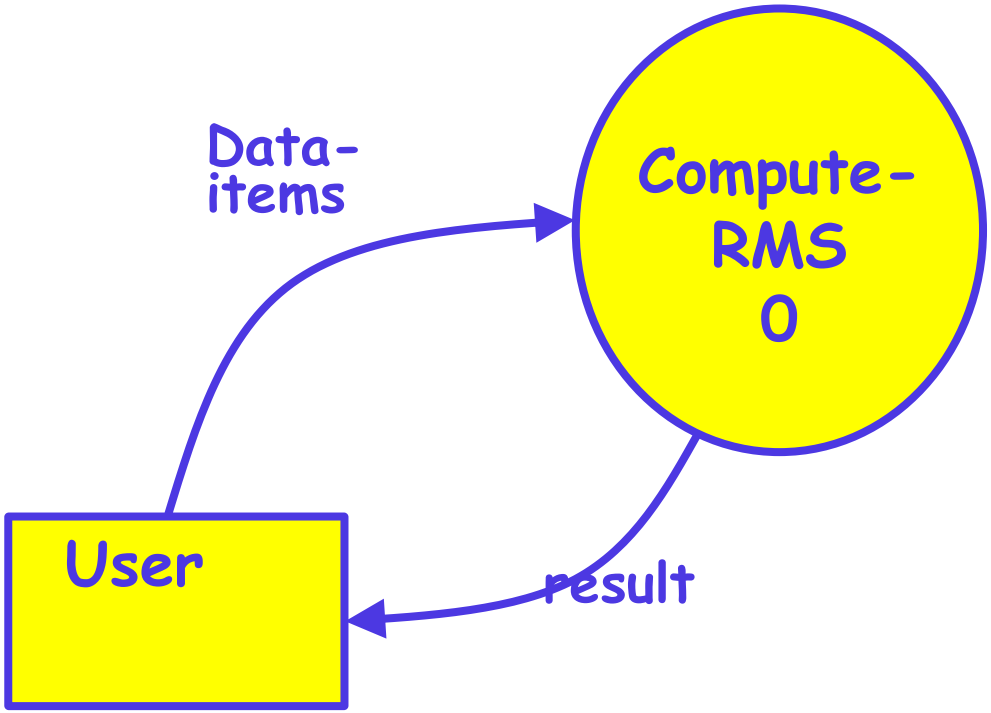
Example 2: RMS Calculating Software

- Consider a software called RMS calculating software:
  - Reads three integers in the range of -1000 and +1000
  - Finds out the root mean square (rms) of the three input numbers
  - Displays the result.

Example 2: RMS Calculating Software

The context diagram is simple to develop:

The system accepts 3 integers from the user Returns the result to him.

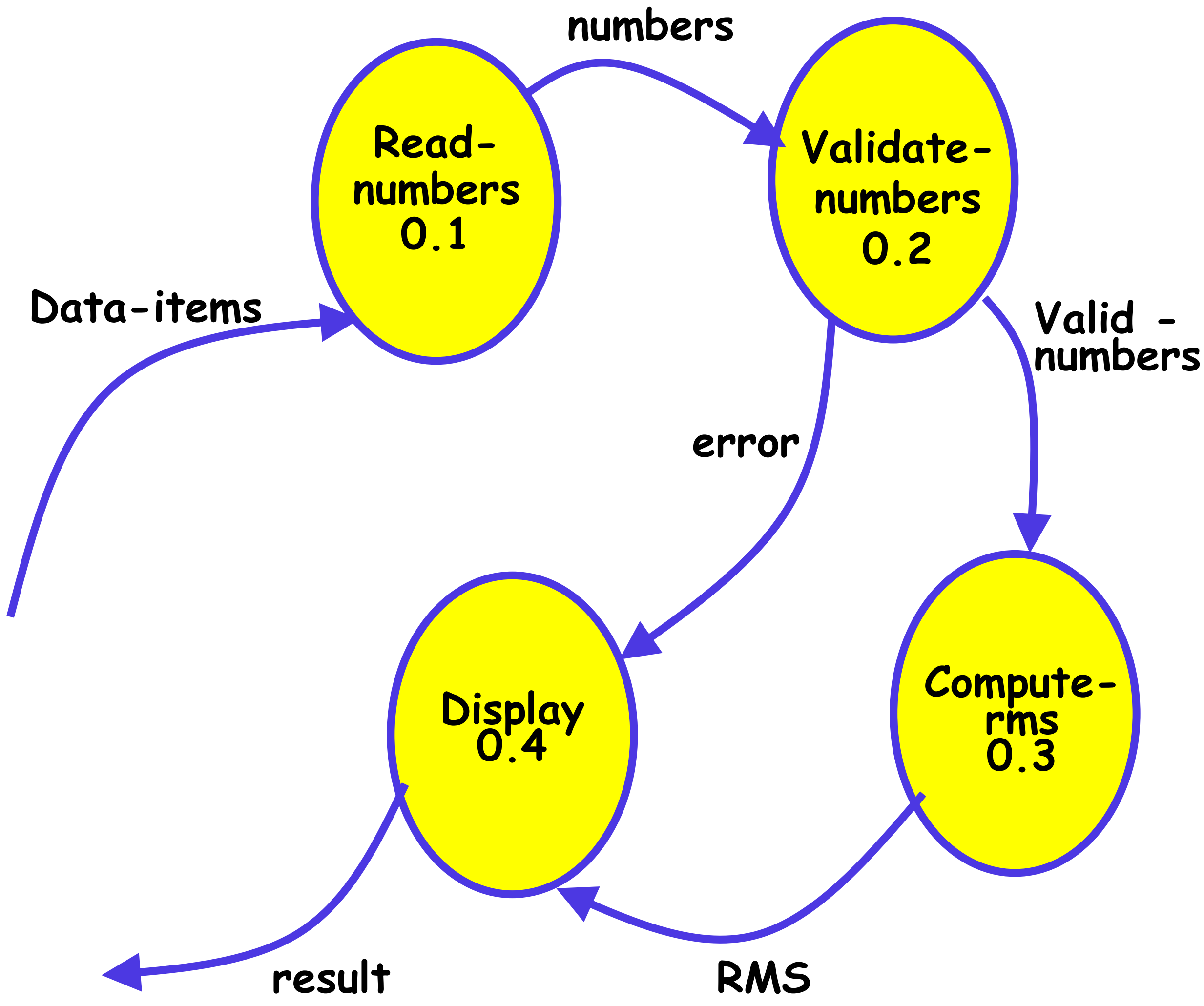


Context Diagram (Level 0 DFD)

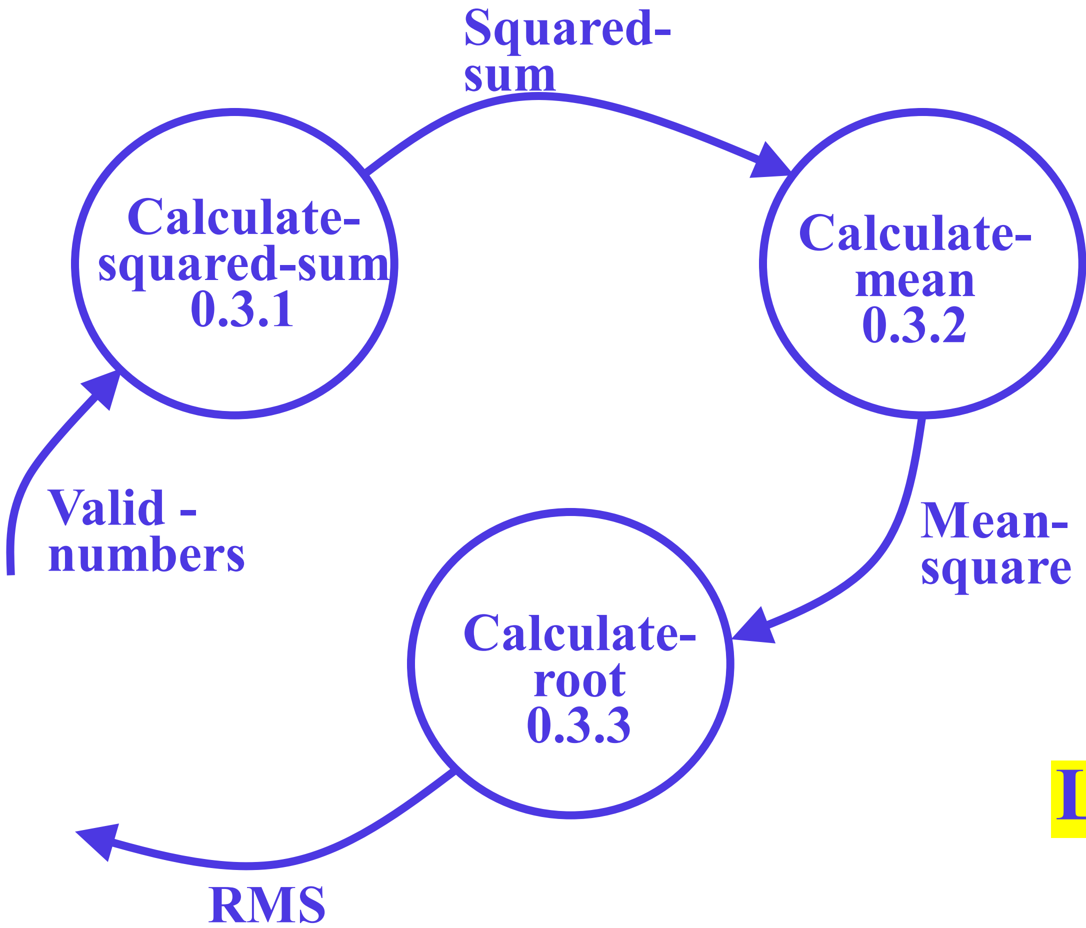
Example 2: RMS Calculating Software

- From a cursory analysis of the problem description:
  - We can see that the system needs to perform several things.
    1. **Accept input numbers from the user:**
    2. **Validate the numbers,**
    3. **Calculate the root mean square of the input numbers**
    4. **Display the result.**

Example 2: Level 1 DFD RMS Calculating Software

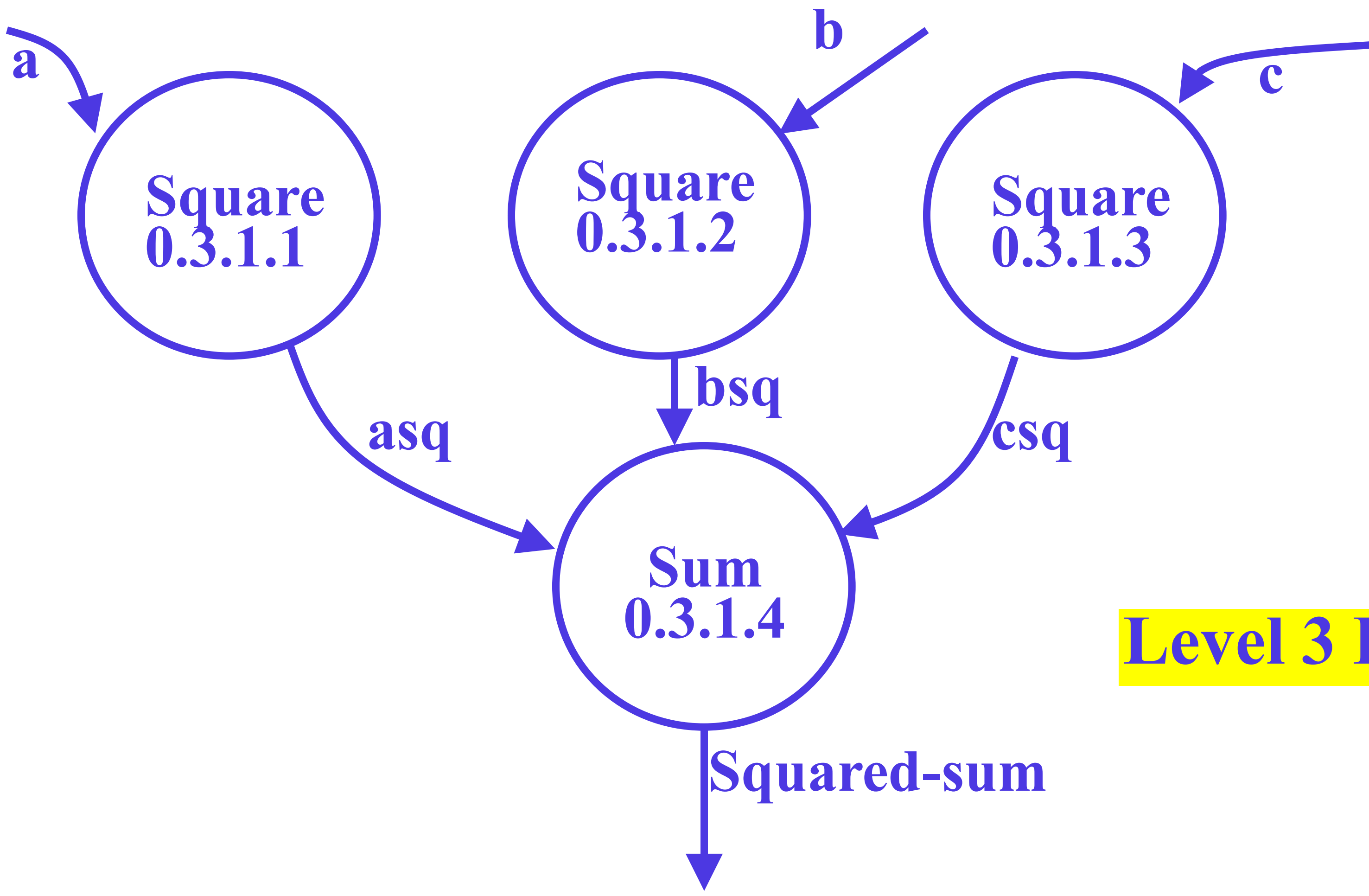


Example 2: RMS Calculating Software



Level 2 DFD

Example 2: RMS Calculating Software



Level 3 DFD

Example2: RMS Calculating Software

- Decomposition is never carried on up to basic instruction level:
  - A bubble is not decomposed any further:
    - If it can be represented by a simple set of instructions.

Data Dictionary

- A DFD is always accompanied by a data dictionary.
- A data dictionary lists all data items appearing in a DFD:
  - definition of all composite data items in terms of their component data items.
  - all data names along with the purpose of data items.
- For example, a data dictionary entry may be:
  - grossPay = regularPay+overtimePay

Data Dictionary

- A DFD is always accompanied by a data dictionary.
- A data dictionary lists all data items appearing in a DFD:
  - Definition of all composite data items in terms of their component data items.
  - All data names along with the purpose of the data items.
- For example, a data dictionary entry may be:
  - **grossPay = regularPay+overtimePay**

Importance of Data Dictionary

- Provides the team of developers with standard terminology for all data:
  - A consistent vocabulary for data is very important

- In the absence of a data dictionary, different developers 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 Dictionary

- CASE (Computer Aided Software Engineering) tools come handy:
  - CASE tools capture the data items appearing in a DFD automatically to generate the data dictionary.
- CASE tools support queries:
  - About definition and usage of data items.
- For example, queries may be made to find:
  - Which data item affects which processes,
  - A process affects which data items,
  - The definition and usage of specific data items, etc.
- Query handling is facilitated:
  - If data dictionary is stored in a relational database management system (RDBMS).

Data Definition

- Composite data are defined in terms of primitive data items using simple operators:
- **+:** denotes composition of data items, e.g
  - **a+b: represents data a together with b.**
- **[,,,]:** represents selection,
  - Any one of the data items listed inside the square bracket can occur.
  - For example, **[a,b] represents either a occurs or b**
- **( ):** contents inside the bracket represent optional data which may or may not appear.
  - **a+(b) represents either a or a+b**
- {}: represents iterative data definition,
  - **{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.

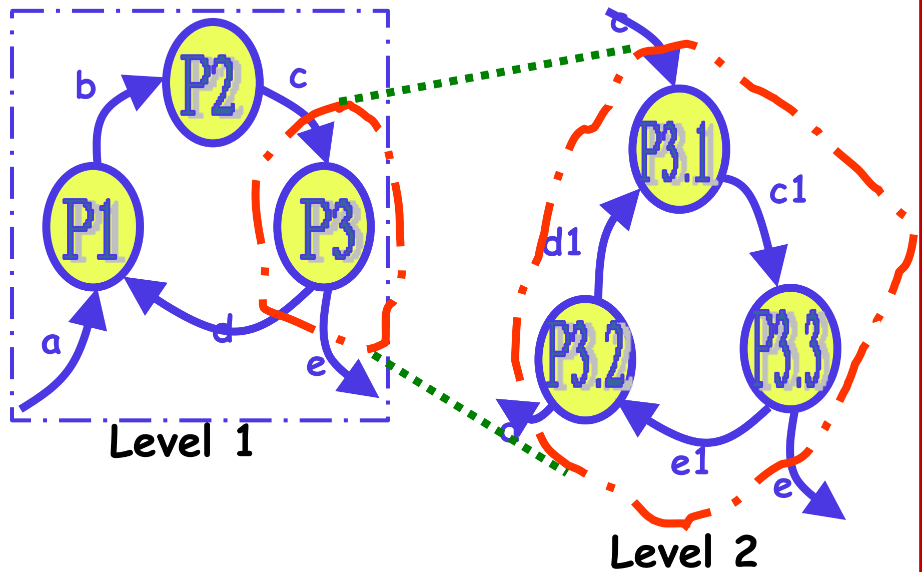
- * *: Anything appearing within * * is considered as comment.

Data Dictionary for RMS Software

- **numbers=valid-numbers=a+b+c**
- **a:integer * input number ***
- **b:integer * input number ***
- **c:integer * input number ***
- **asq:integer**
- **bsq:integer**
- **csq:integer**
- **squared-sum: integer**
- **Result=[RMS,error]**
- **RMS: integer * root mean square value***
- **error:string * error message***

Balancing a DFD

- **Data flowing into or out of a bubble:**
  - **Must match the data flows at the next level of DFD.**



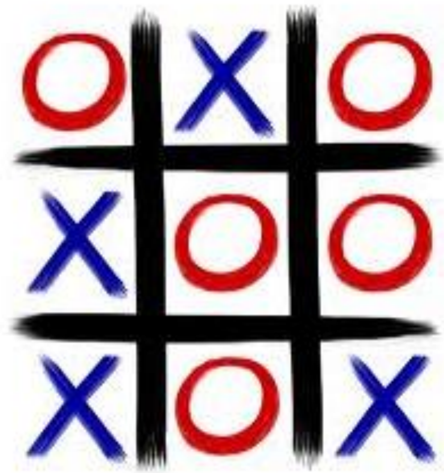
- In the level 1 of the DFD,
  - Data item c flows into the bubble P3 and the data item d and e flow out.
- In the next level, bubble P3 is decomposed.
  - The decomposition is balanced as data item c flows into the level 2 diagram and d and e flow out.

Numbering of Bubbles

- Number the bubbles in a DFD:
  - **Numbers help in uniquely identifying any bubble from its bubble number.**
- The bubble at context level:
  - Assigned number 0.
- Bubbles at level 1:
  - Numbered 0.1, 0.2, 0.3, etc
- When a bubble numbered x is decomposed,
  - Its children bubble are numbered x.1, x.2, x.3, etc.

Example: Tic-Tac-Toe Computer Game

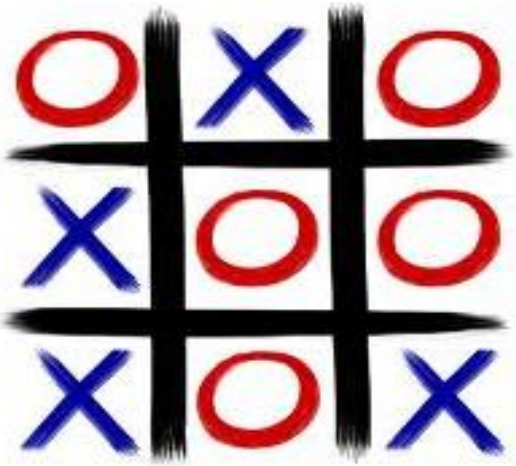
- A human player and the computer make alternate moves on a 3 X 3 square.



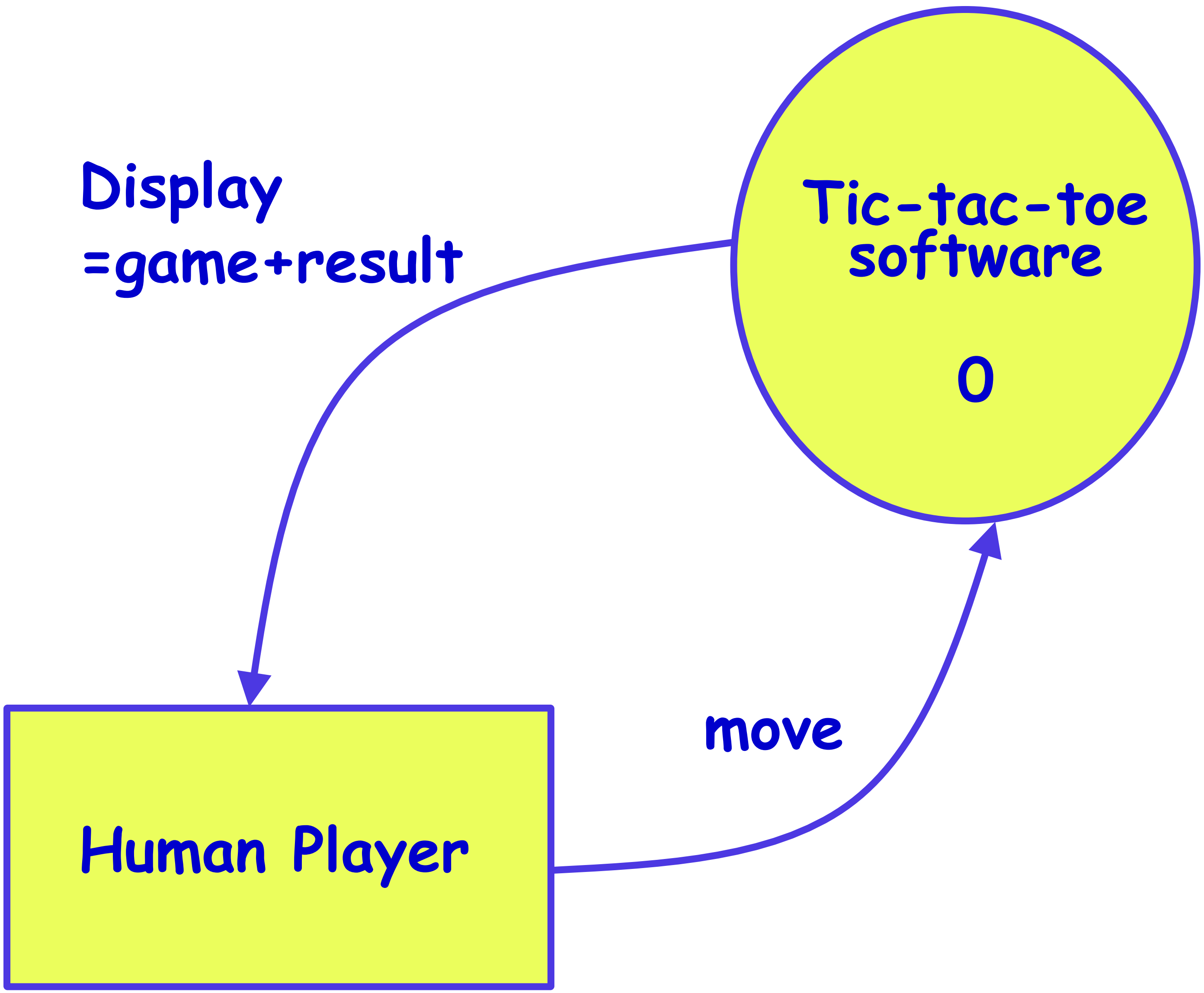
- A move consists of marking a previously unmarked square.
- The user inputs a number between 1 and 9 to mark a square
- Whoever is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins.

Example: Tic-Tac-Toe Computer Game

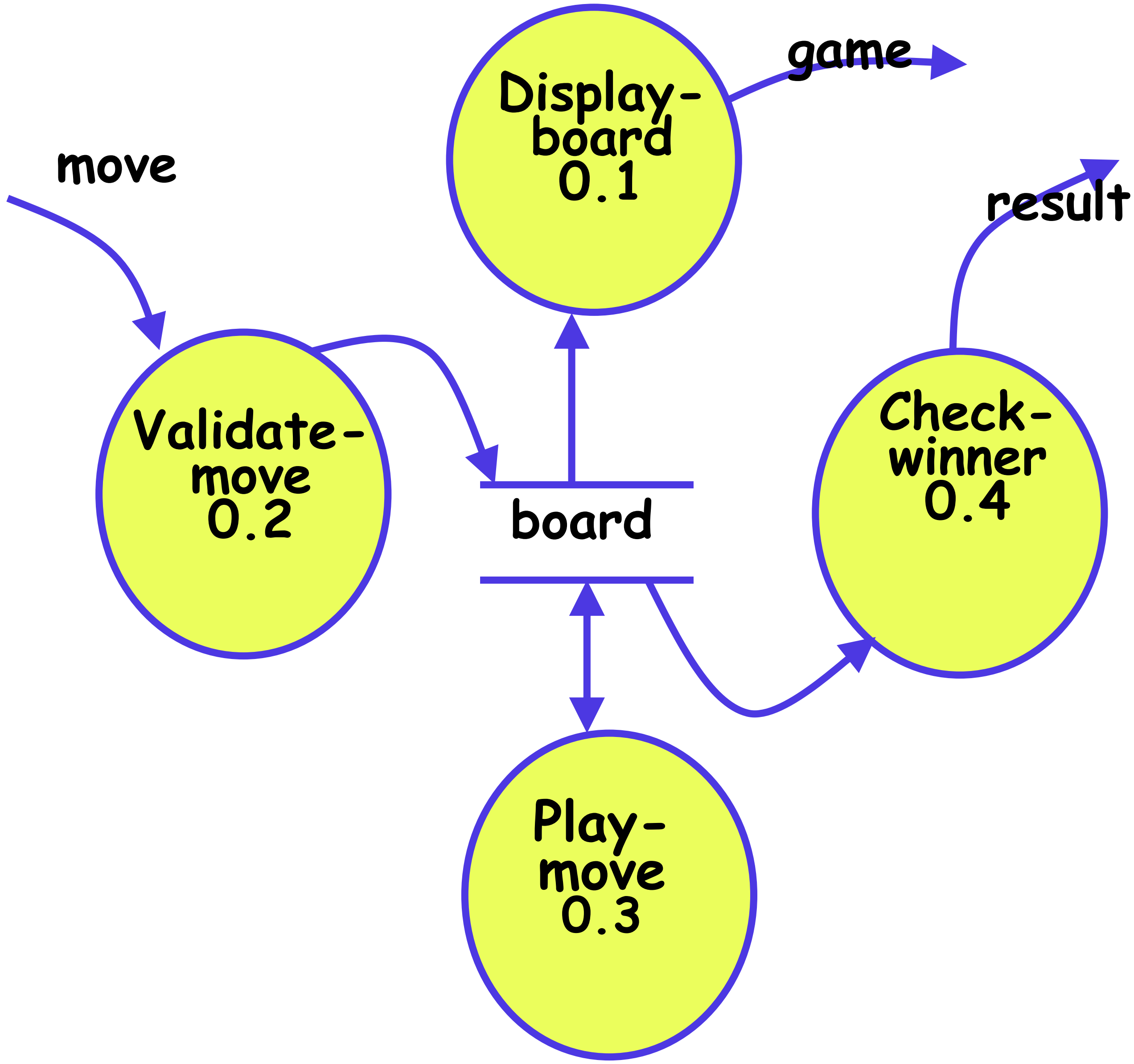
- As soon as either of the human player or the computer wins,
  - A message announcing the winner should be displayed.
- If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up,
  - Then the game is drawn.
- The computer always tries to win a game.



Context Diagram: Tic-tac-toe



Level 1 DFD



game
move
Display-
board
0.1
result
Validate-
move
0.2
board
Check-
winner
0.4
Play-
move
0.3

Data Dictionary

Display=game + result

move = integer

board = {integer}9

game = {integer}9

result=string

Example 3: Trading-House Automation System (TAS)

- A large trading house wants us to develop a software:
  - To automate book keeping activities associated with its business.
- It has many regular customers:
  - They place orders for various kinds of commodities.
- The trading house maintains names and addresses of its regular customers.
- Each customer is assigned a unique customer identification number (**CIN**).

Example: Trading-House Automation System (TAS)

- As per current practice when a customer places order:
  - The accounts department first checks the **credit-worthiness** of the customer.
- The credit worthiness of a customer is determined:
  - By analyzing the history of his payments to the bills sent to him in the past.
- If a customer is not credit-worthy:
  - His orders are not processed any further
  - An appropriate order rejection message is generated for the customer.
- If a customer is credit-worthy:
  - Items he/she has ordered are checked against the list of items the trading house deals with.

Example: Trading-House Automation System (TAS)

- **The items that the trading house does not deal with:**
  - Are not processed any further
  - An appropriate message for the customer for these items is generated.

- The items in a customer's order that the trading house deals with:
  - Are checked for availability in inventory.

- If the items are available in the inventory in desired quantities:
  - A bill with the forwarding address of the customer is printed.
  - A material issue slip is printed.

- If an ordered item is not available in the inventory in sufficient quantity:
  - To be able to fulfil pending orders store details in a "pending-order" file :
    - out-of-stock items along with quantity ordered.
    - customer identification number

Example: Trading-House Automation System (TAS)

- The customer can produce the material issue slip at the store house:
  - Take delivery of the items.
  - Inventory data adjusted to reflect the sale to the customer.

Example: Trading-House Automation System (TAS)

- The purchase department:
  - would periodically issue commands to generate indents.
- When **generate indents** command is issued:
  - The system should examine the "pending-order" file
  - Determine the orders that are pending
  - Total quantity required for each of the items.

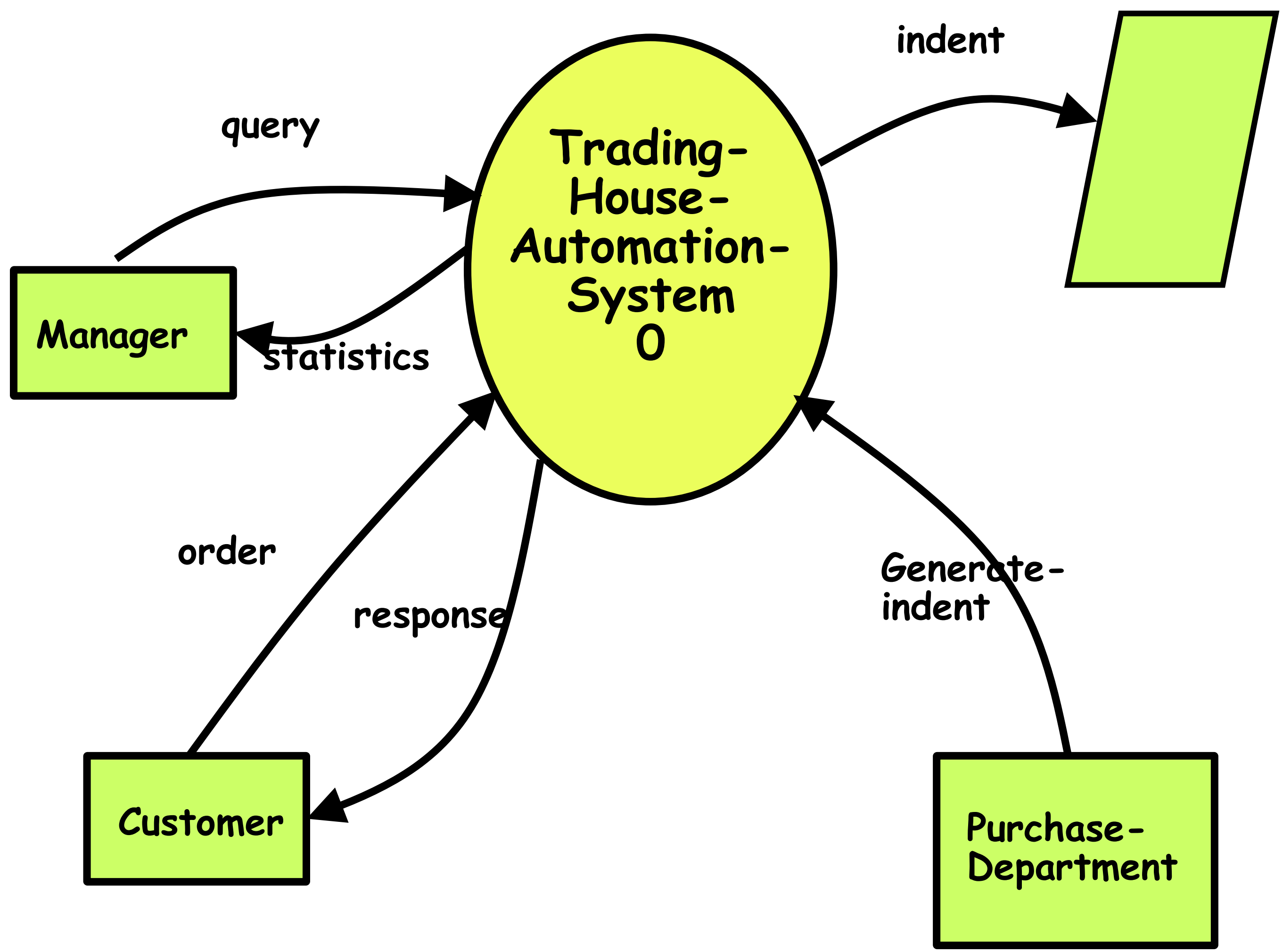
Example: Trading-House Automation System (TAS)

- TAS should find out the addresses of the vendors who supply the required items:
  - Examine the file containing vendor details (their address, items they supply etc.)
  - Print out indents to those vendors.

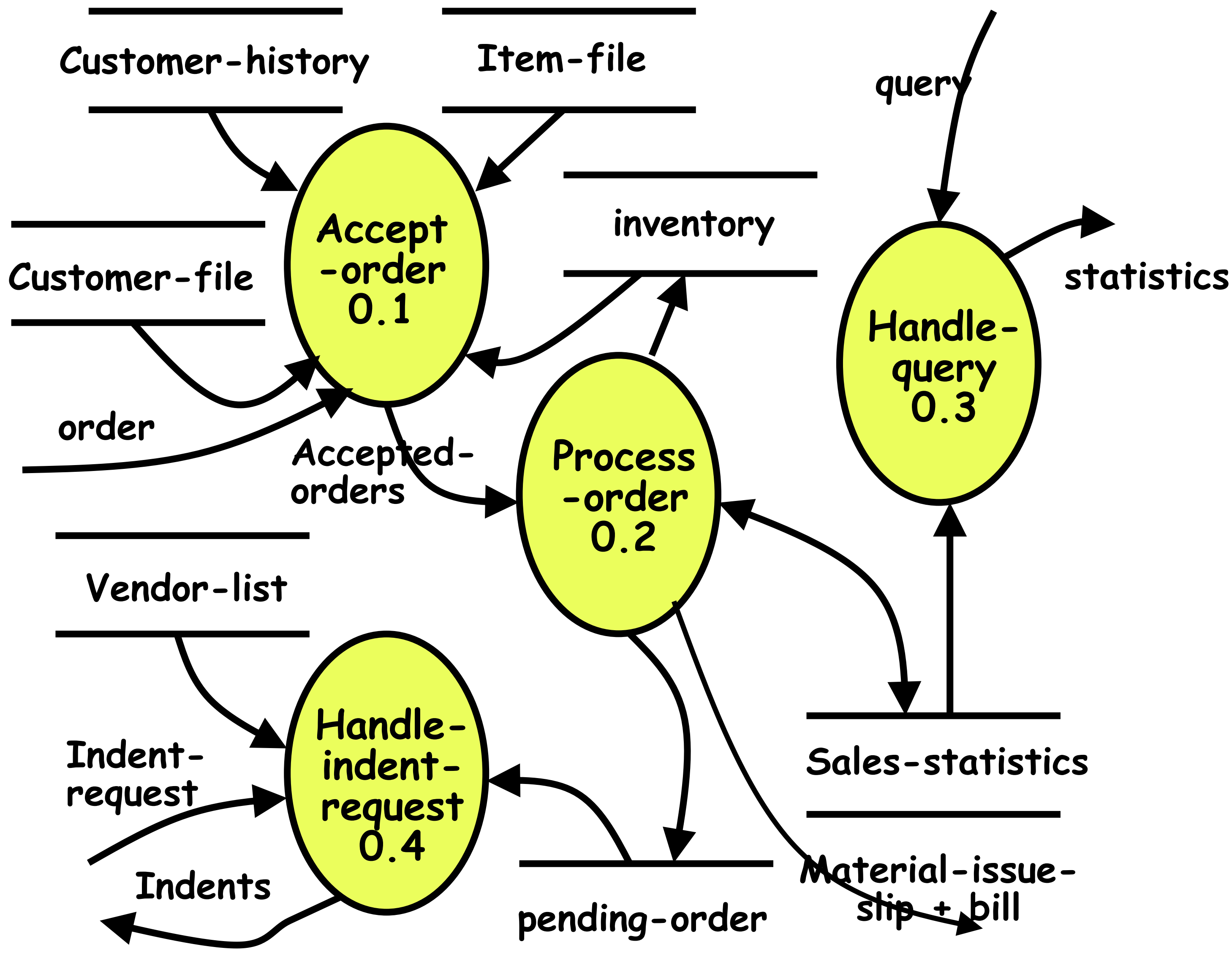
Example: Trading-House Automation System (TAS)

- TAS should also answers managerial queries:
  - Statistics of different items sold over any given period of time
  - Corresponding quantity sold and the price realized.

Context Diagram



Level 1 DFD



Customer-history
Item-file
query
Accept
-order
0.1
inventory
Customer-file
statistics
Handle-
query
0.3
order
Accepted-
orders
Process
-order
0.2
Vendor-list
Handle-
indent-
request
0.4
Indent-
request
Sales-statistics
Indents
pending-order
Material-issue-
slip + bill

Example: Data Dictionary

- **response: [bill + material-issue-slip, reject-message]**
- **query: period** /* query from manager regarding sales statistics*/
- **period: [date+date,month,year,day]**
- **date: year + month + day**
- **year: integer**
- **month: integer**
- **day: integer**
- **order: customer-id + {items + quantity}***
- **accepted-order: order** /* ordered items available in inventory */
- **reject-message: order + message** /* rejection message */
- **pending-orders: customer-id + {items+quantity}***
- **customer-address: name+house#+street#+city+pin**

Example: Data Dictionary

- **item-name: string**
- **house#: string**
- **street#: string**
- **city: string**
- **pin: integer**
- **customer-id: integer**
- **bill: {item + quantity + price}* + total-amount + customer-address**
- **material-issue-slip: message + item + quantity + customer-address**
- **message: string**
- **statistics: {item + quantity + price }***
- **sales-statistics: {statistics}***
- **quantity: integer**

Observation

- From the discussed examples,
  - Observe that DFDs help create:
    - **Data model**
    - **Function model**

- As a DFD is refined into greater levels of detail:
  - **The analyst performs an implicit functional decomposition.**
  - **At the same time, refinements of data takes place.**

Guidelines For Constructing DFDs

- Context diagram should represent the system as a single bubble:
  - **Many beginners commit the mistake of drawing more than one bubble in the context diagram.**
- All external entities should be represented in the context diagram:
  - External entities should not appear at any other level DFD.
- Only 3 to 7 bubbles per diagram should be allowed:
  - Each bubble should be decomposed to between 3 and 7 bubbles.

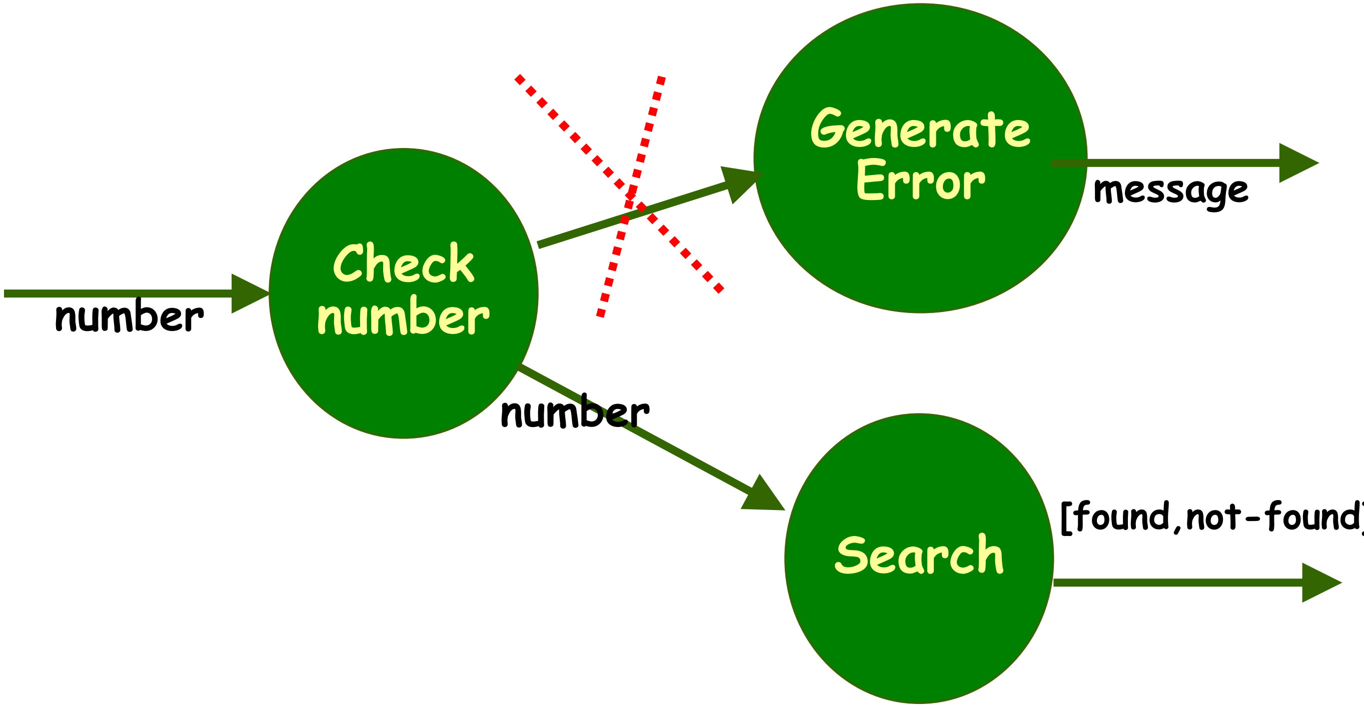
Guidelines For Constructing DFDs

- A common mistake committed by many beginners:
  - Attempting to represent control information in a DFD.
  - e.g. trying to represent the order in which different functions are executed.

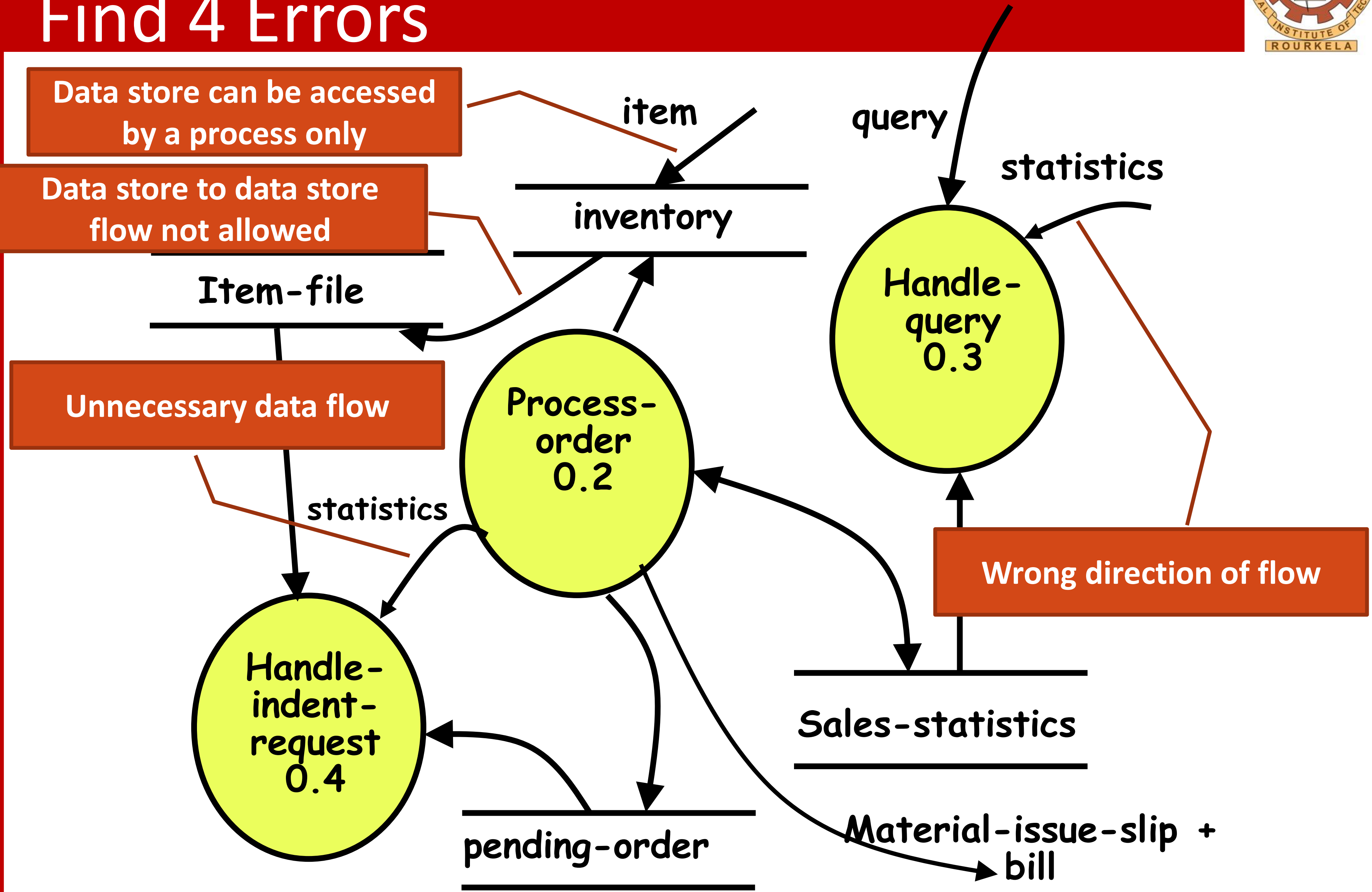
- A DFD model does not represent control information:
  - When or in what order different functions (processes) are invoked The conditions under which different functions are invoked are not represented.
  - For example, a function might invoke one function or another depending on some condition.
  - **Many beginners try to represent this aspect by drawing an arrow between the corresponding bubbles.**

Find Error Example-1

- Functionality: Check the input value:
  - If the input value is less than -1000 or greater than +1000 generate an error message
  - otherwise search for the number



Find 4 Errors



Guidelines For Constructing DFDs

- All functions of the system must be captured in the DFD model:
  - **No function specified in the SRS document should be overlooked.**

- Only those functions specified in the SRS document should be represented:
  - **Do not assume extra functionality of the system not specified by the SRS document.**

Commonly Made Errors

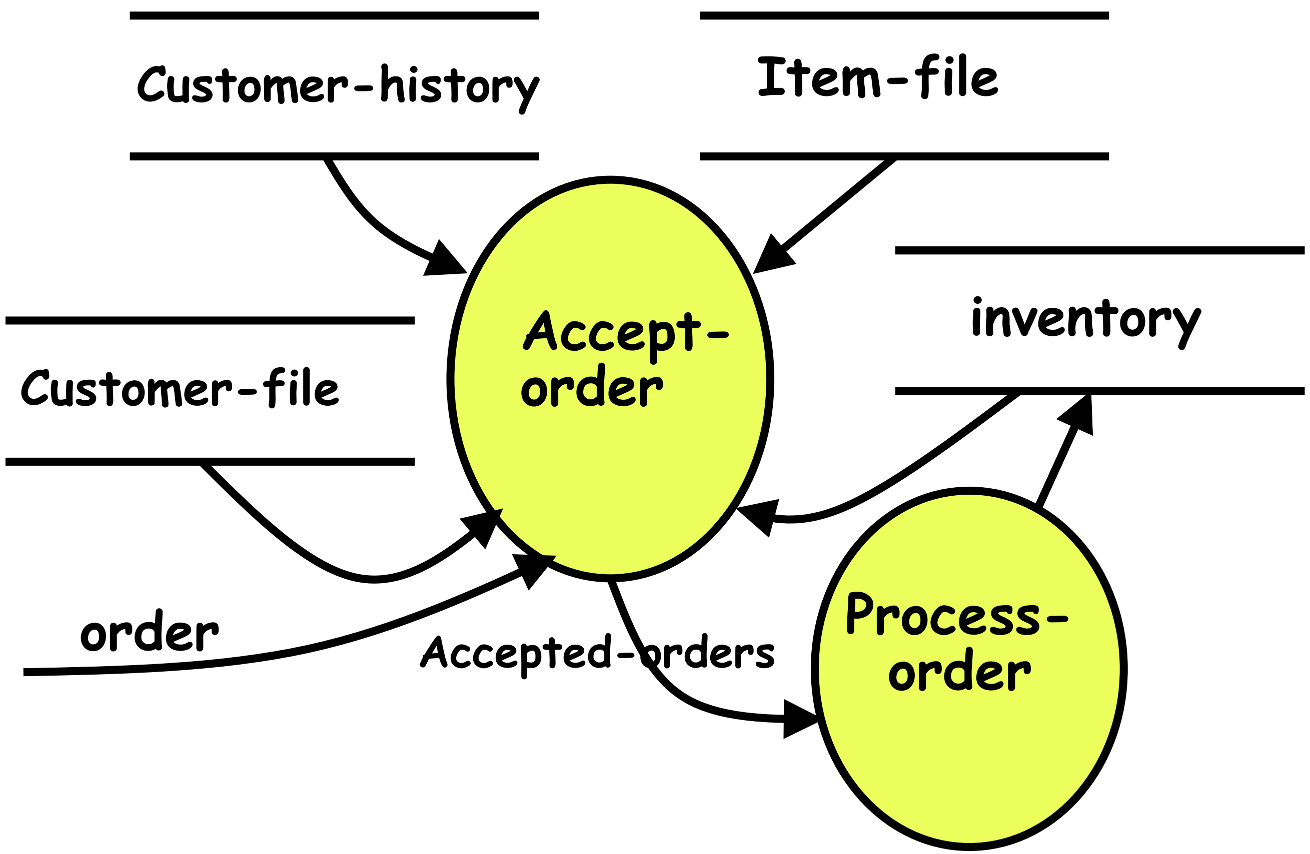
- Unbalanced DFDs
- Forgetting to name the data flows
- Unrepresented functions or data
- External entities appearing at higher level DFDs
- Trying to represent control aspects
- Context diagram having more than one bubble
- A bubble decomposed into too many bubbles at next level
- Terminating decomposition too early
- Nouns used in naming bubbles

Shortcomings of the DFD Model

- DFD models suffer from several shortcomings:
- DFDs leave ample scope to be imprecise.
  - In a DFD model, we infer about the function performed by a bubble from its label.
  - A label may not capture all the functionality of a bubble.
- For example, a bubble named **find-book-position** has only intuitive meaning:
  - Does not specify several things:
    - What happens when some input information is missing or is incorrect.
    - Does not convey anything regarding what happens when book is not found
    - What happens if there are books by different authors with the same book title.

Shortcomings of the DFD Model

- Control information is not represented:
  - For instance, order in which inputs are consumed and outputs are produced is not specified.



Shortcomings of the DFD Model

- Decomposition is carried out to arrive at the successive levels of a DFD is subjective.

- **The ultimate level to which decomposition is carried out is subjective:**
  - Depends on the judgement of the analyst.

- **Even for the same problem,**
  - **Several alternative DFD representations are possible:**
  - **Many times it is not possible to say which DFD representation is superior or preferable.**

Shortcomings of the DFD Model

- DFD technique does not provide:
  - Any clear guidance as to how exactly one should go about decomposing a function:
  - One has to use subjective judgement to carry out decomposition.

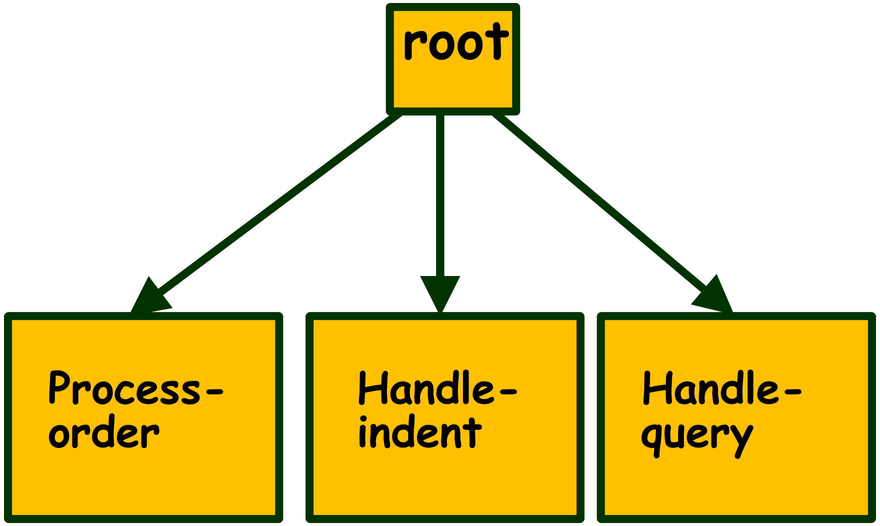
- Structured analysis techniques do not specify when to stop a decomposition process:
  - To what length decomposition needs to be carried out.

DFD Tools

- Several commercial and free tools available.
- **Commercial:**
  - Visio
  - Smartdraw (30 day free trial)
  - Edraw
  - Creately
  - Visual analyst
- **Free:**
  - Dia (GNU open source)

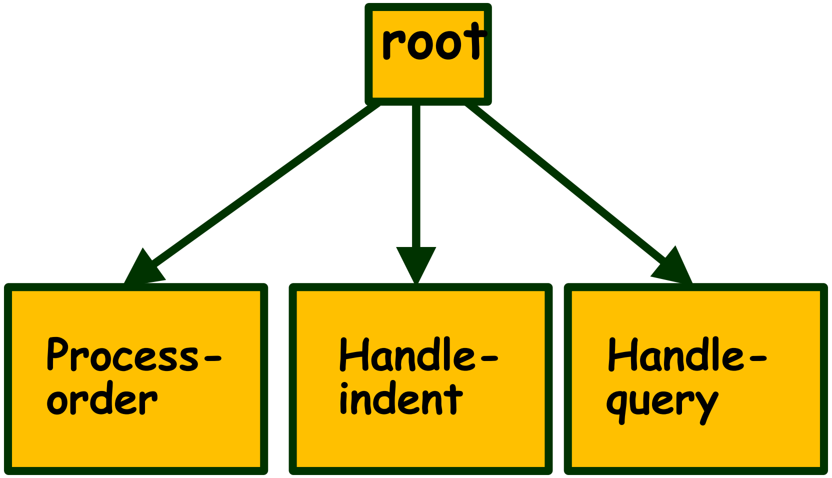
Structured Design

- The aim of structured design
  - **Transform the results of structured analysis (DFD representation) into a structure chart.**



Structure Chart

- Structure chart representation
  - Easily implementable using programming languages.



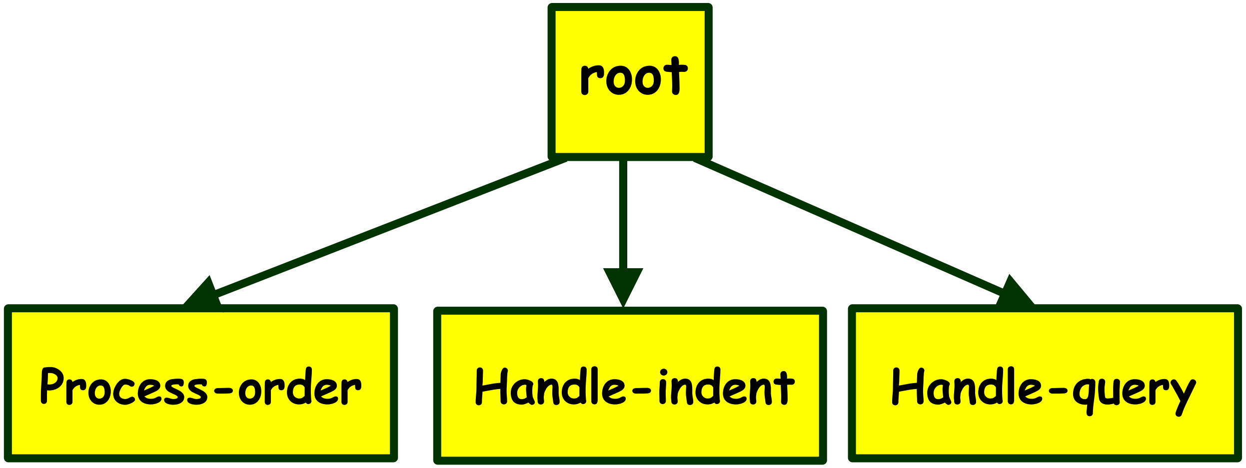
- Main focus of a structure chart:
  - Define the module structure of a software,
  - Interaction among different modules, (call relationship)
  - **Procedural aspects (e.g, how a particular functionality is achieved) are not represented.**

Basic Building Blocks of Structure Chart

- Rectangular box:
  - A rectangular box represents a module.
  - Annotated with the name of the module it represents.

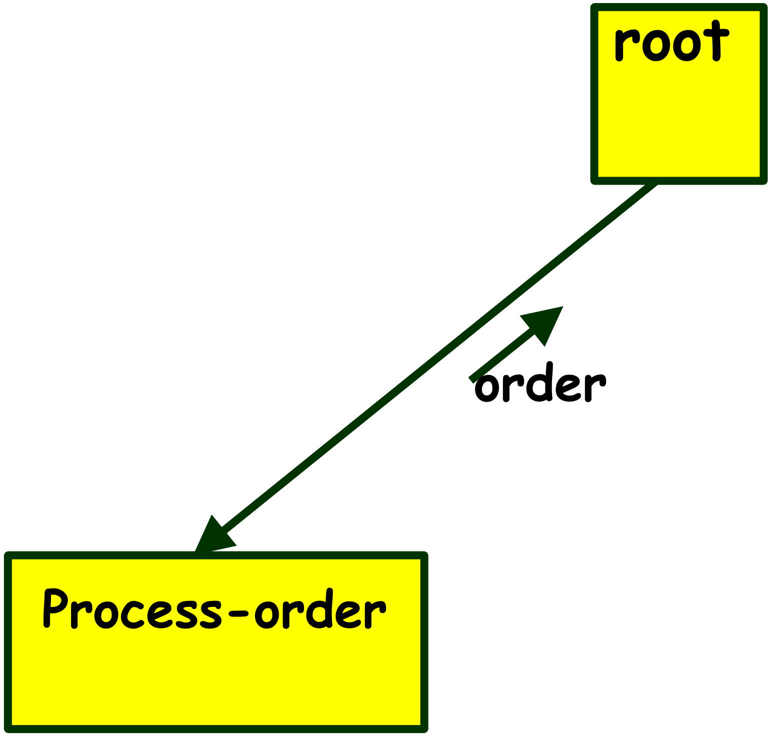
Arrows

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



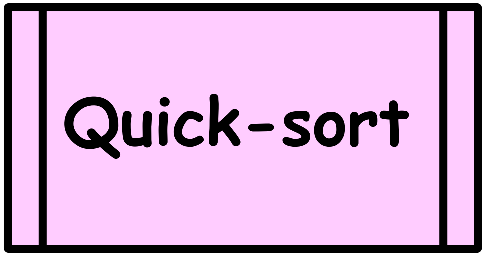
Data Flow Arrows

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



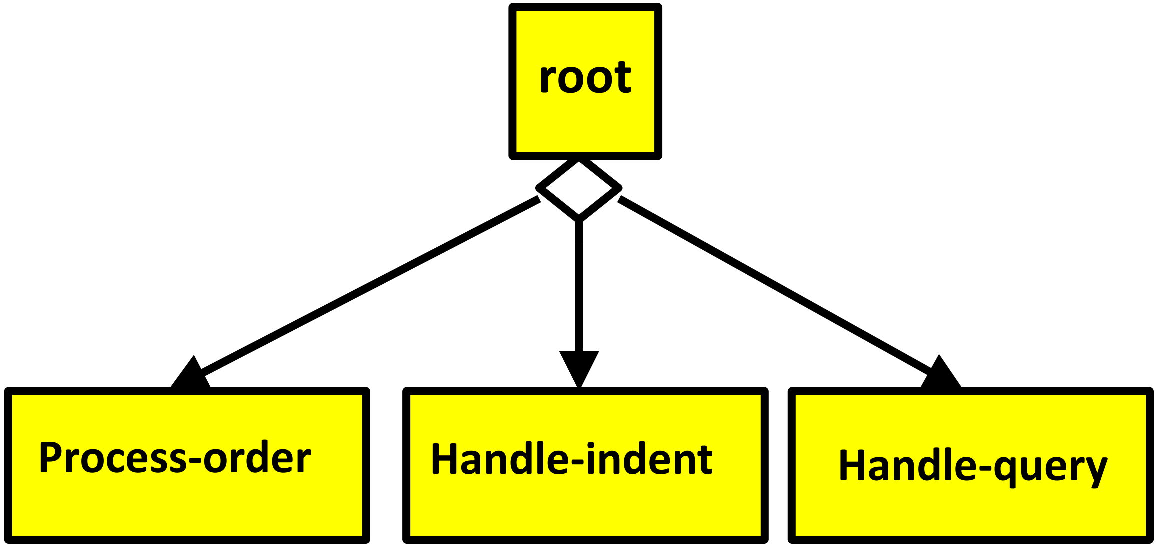
Library Modules

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



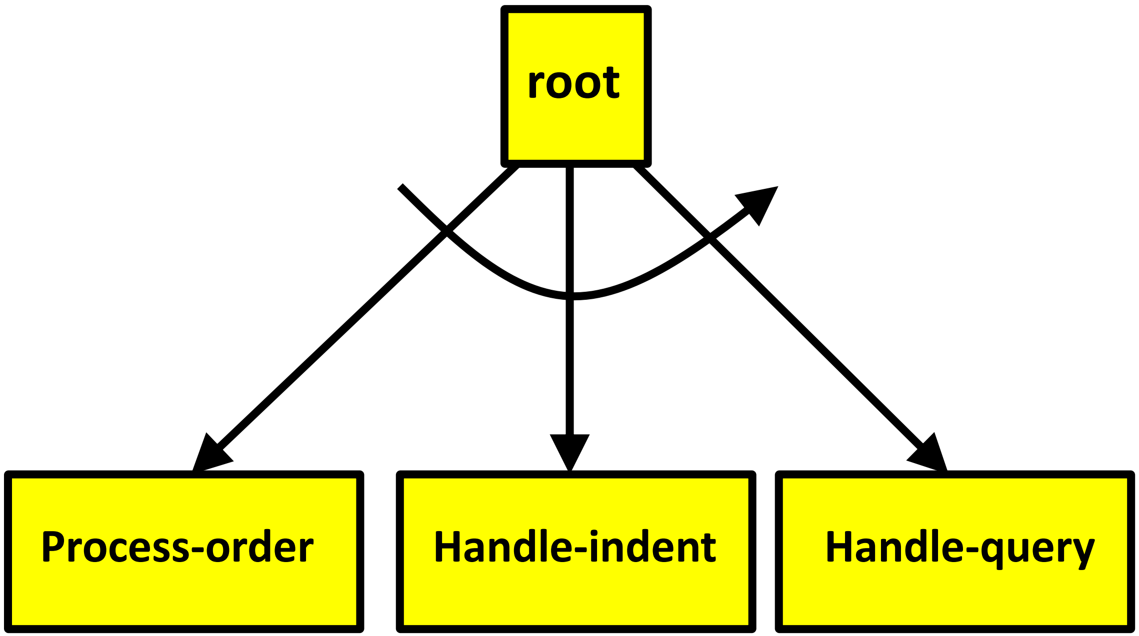
Selection

- The diamond symbol represents:
  - **Each one of several modules connected to the diamond symbol is invoked depending on some condition.**



Repetition

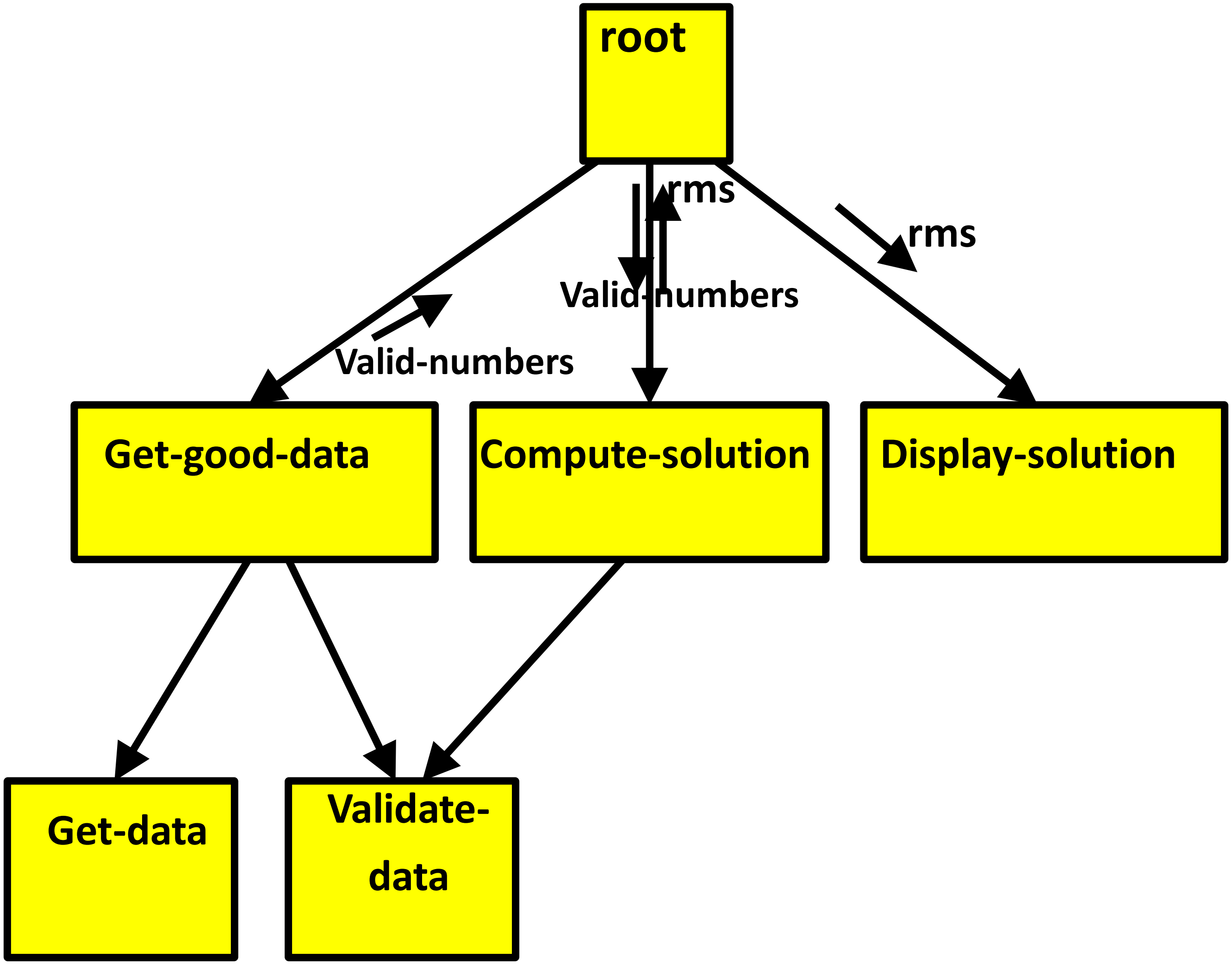
- A loop around control flow arrows denotes that the concerned modules are invoked repeatedly.



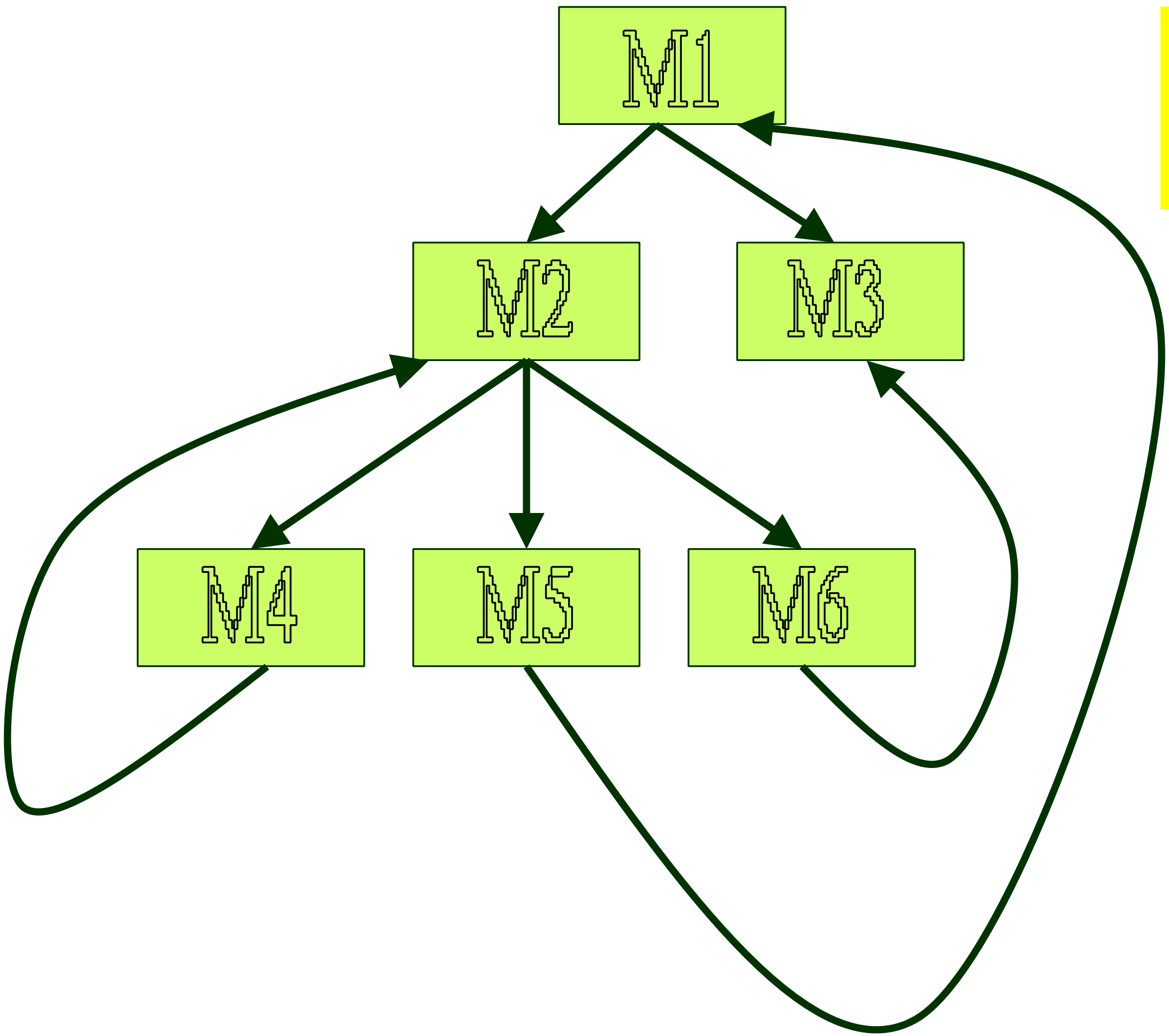
Structure Chart

- There is only one module at the top:
  - the **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:
  - **Modules in a structure chart should be arranged in layers or levels.**
- Makes use of 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: Good Design



Example: Bad Design



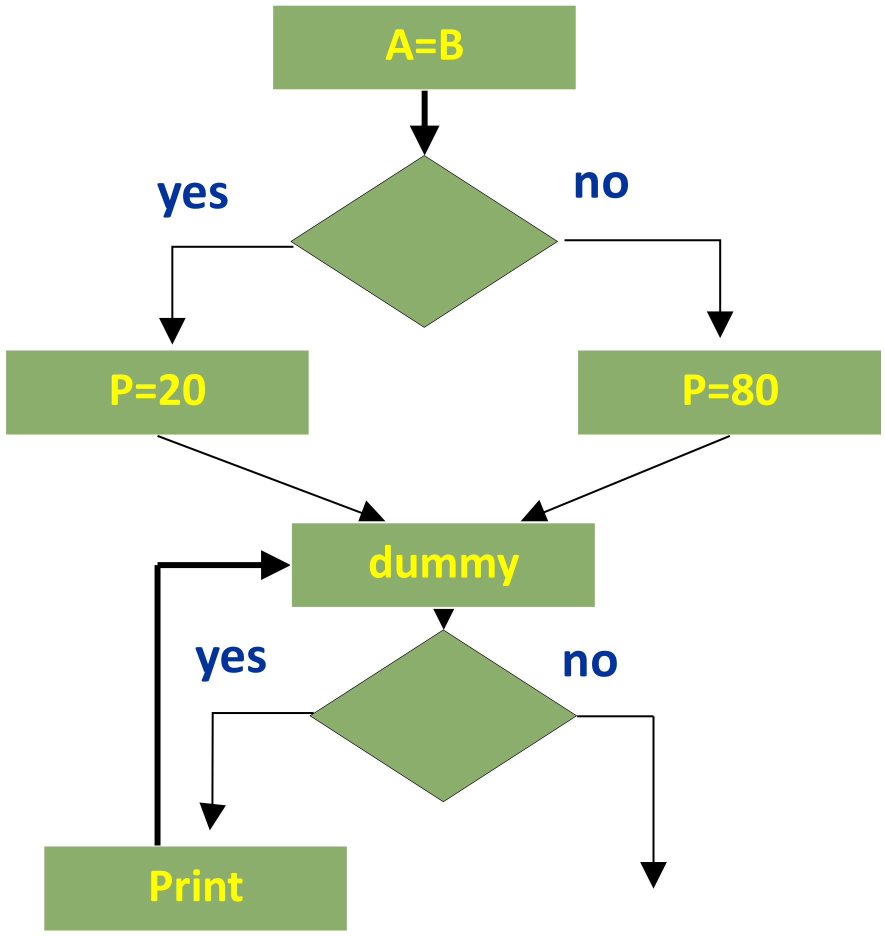
Shortcomings of Structure Chart

- By examining 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 (Aside)

- 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 versus Structure Chart

1. It is difficult to identify modules of a software from its flow chart representation.

2. Data interchange among the modules is not represented in a flow chart.

3. **Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.**

Transformation of a DFD Model into Structure Chart

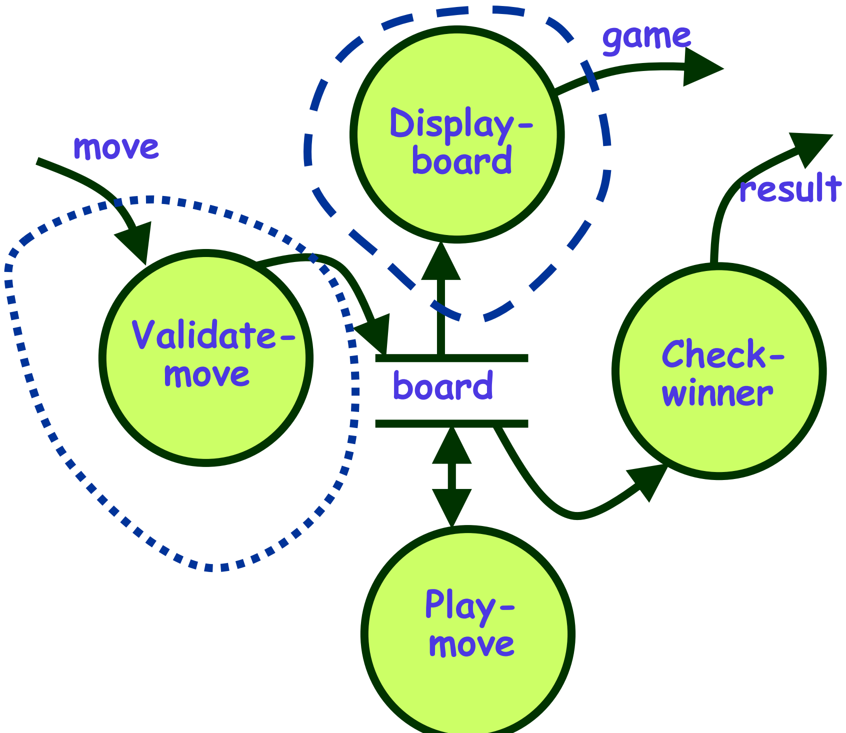
- Two strategies exist to guide transformation of a DFD into a structure chart:
  - **Transform Analysis**
  - **Transaction Analysis**

Transform Analysis

- The first step in transform analysis:
  - Divide the DFD into 3 parts:
    - **Input**
      - Processes dealing with input to the system
    - **output**
      - Processes dealing with output from the system
    - **logical processing**
      - Processes dealing with logical processing

Transform Analysis

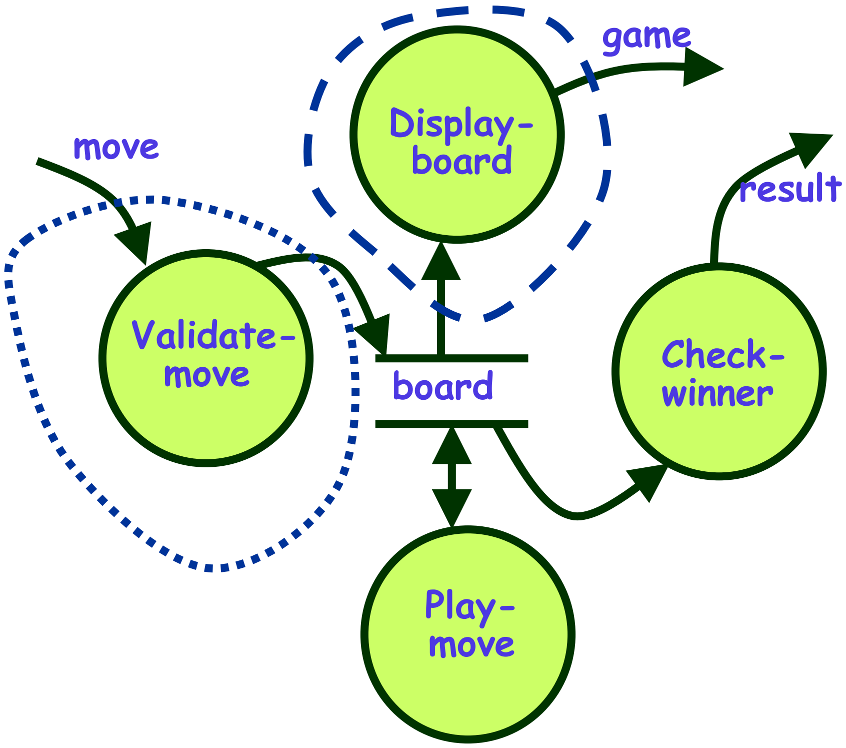
- Input portion in the DFD:
  - processes which convert input data from physical to logical form.
  - e.g. read characters from the terminal and store in internal tables or lists.



- Each input portion:
  - called an **afferent branch.**
  - Possible to have more than one afferent branch in a DFD.

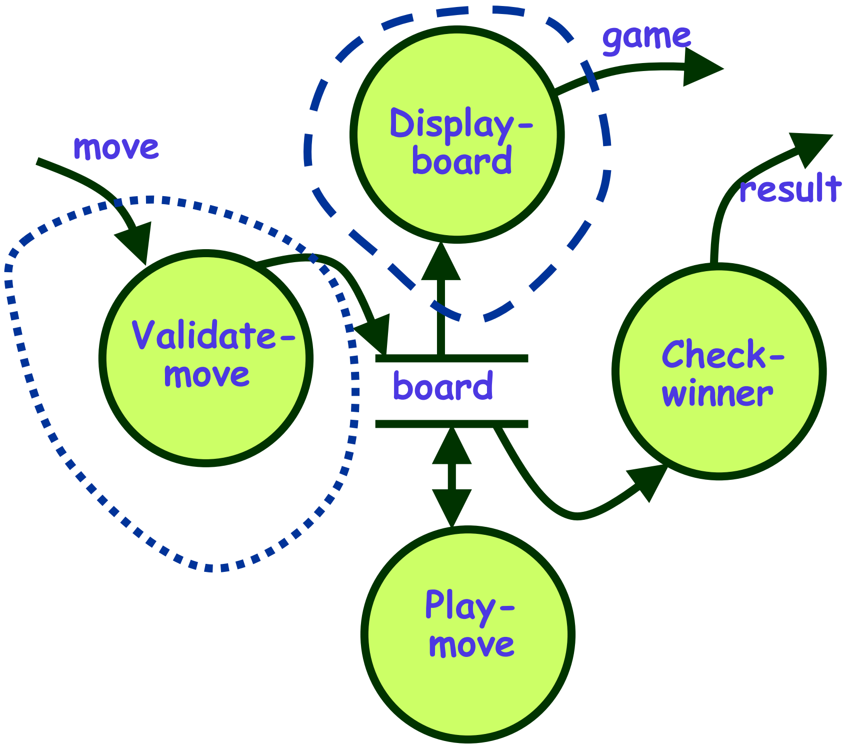
Transform Analysis

- Output portion of a DFD:
  - transforms output data from logical form to physical form.
    - e.g., from list or array into output characters.
  - Each output portion:
    - called an **efferent branch.**
- The remaining portions of a DFD
  - called **central transform**



Transform Analysis

- Derive structure chart by drawing one functional component for:
  - **afferent branch,**
  - **central transform,**
  - **efferent branch.**



Transform Analysis

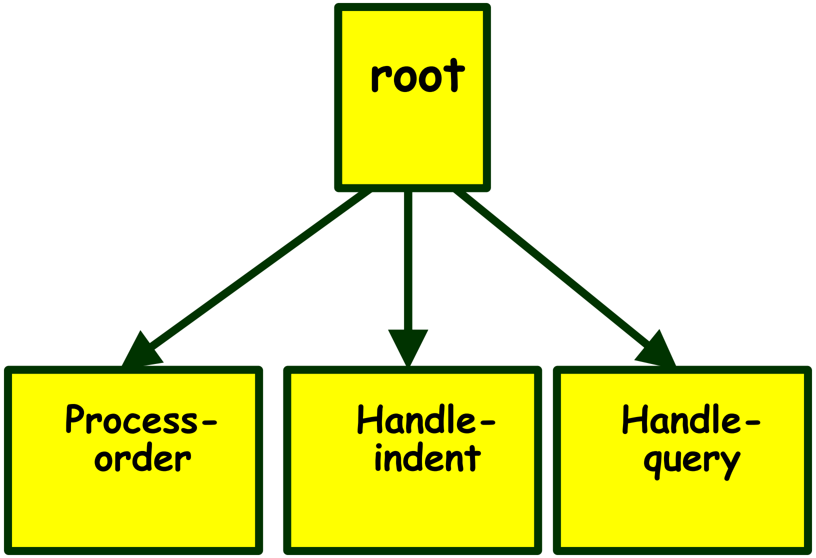
- Identifying input and output transforms:
  - requires experience and skill.

- Some guidelines for identifying central transforms:
  - **Trace inputs until a bubble is found whose output cannot be deduced from the inputs alone.**
  - **Processes which sort input or filter data from it.**

  - **Processes which validate input are not central transforms.**

Transform Analysis

- First level of structure chart:
  - Draw a box for each input and output units
  - A box for the central transform.
- Next, refine the structure chart:
  - Add sub-functions required by each high-level module.
  - Many levels of modules may required to be added.



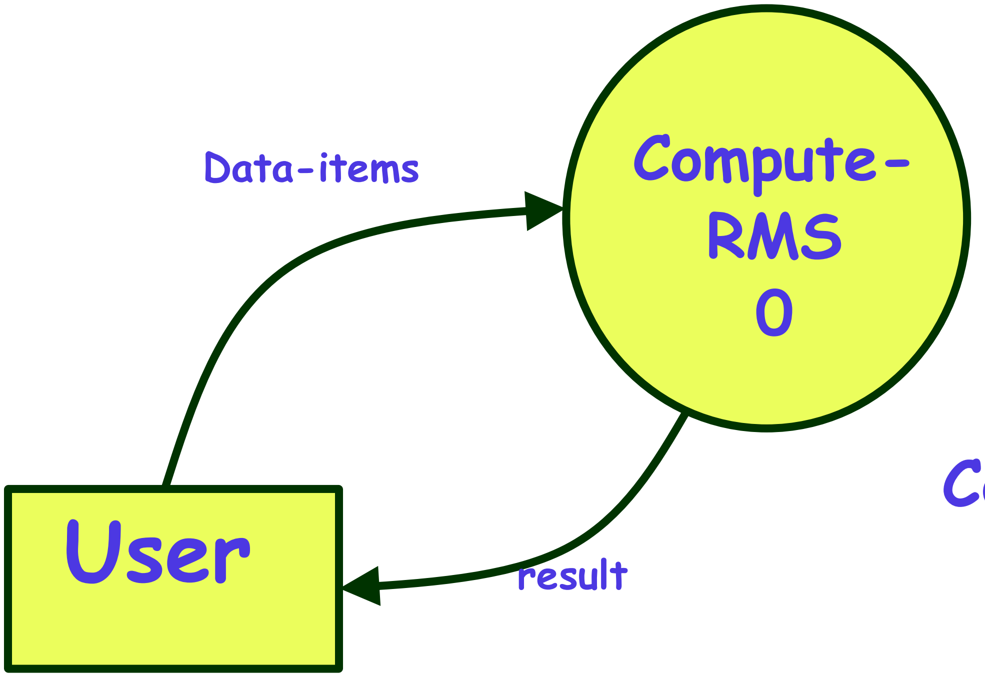
Factoring

- The process of breaking functional components into subcomponents.

- Factoring includes adding:
  - **Read and write modules,**
  - **Error-handling modules,**
  - **Initialization and termination modules,** etc.

- Finally check:
  - Whether all bubbles have been mapped to modules.

Example 1: RMS Calculating Software

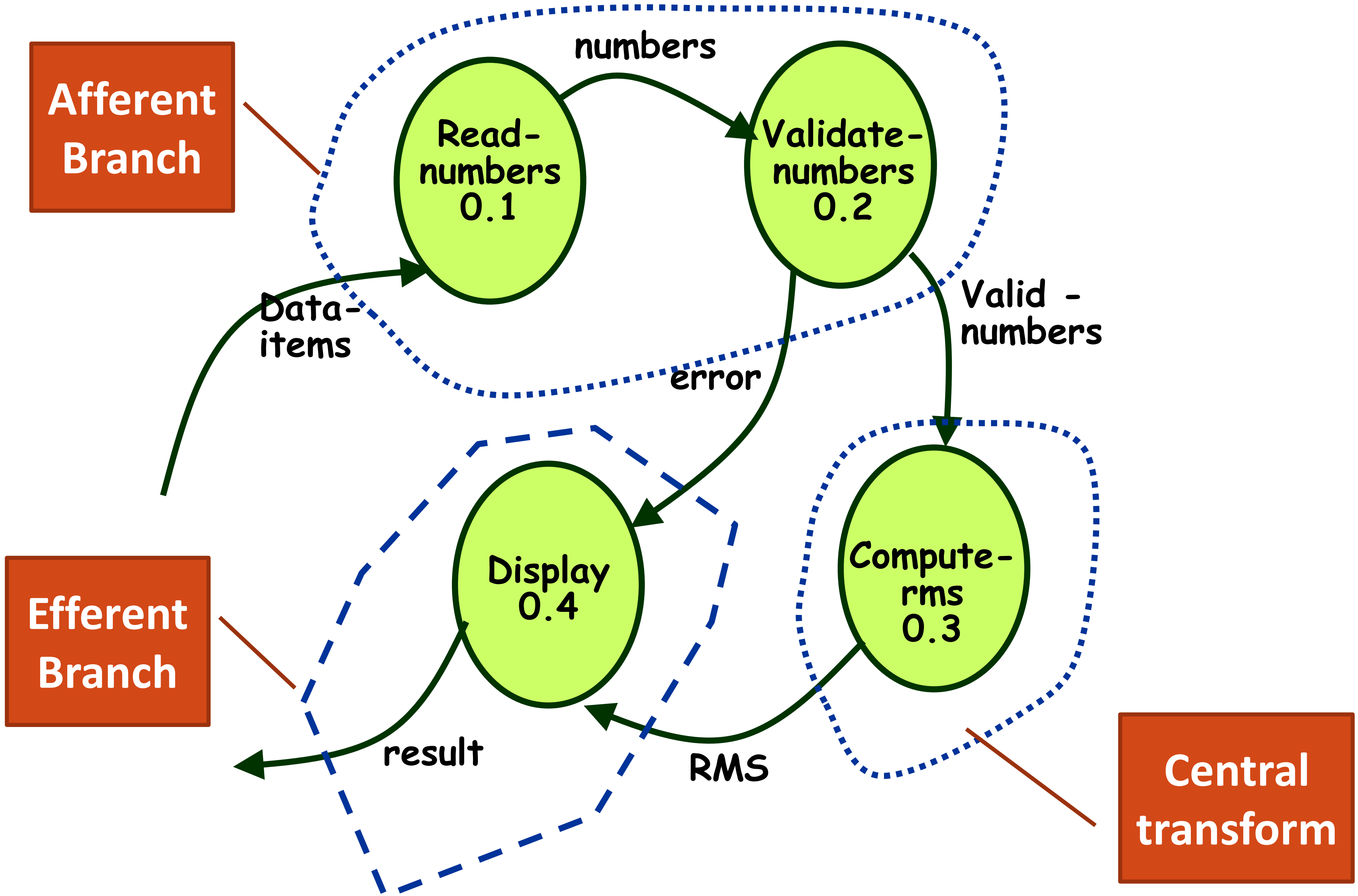


Context Diagram

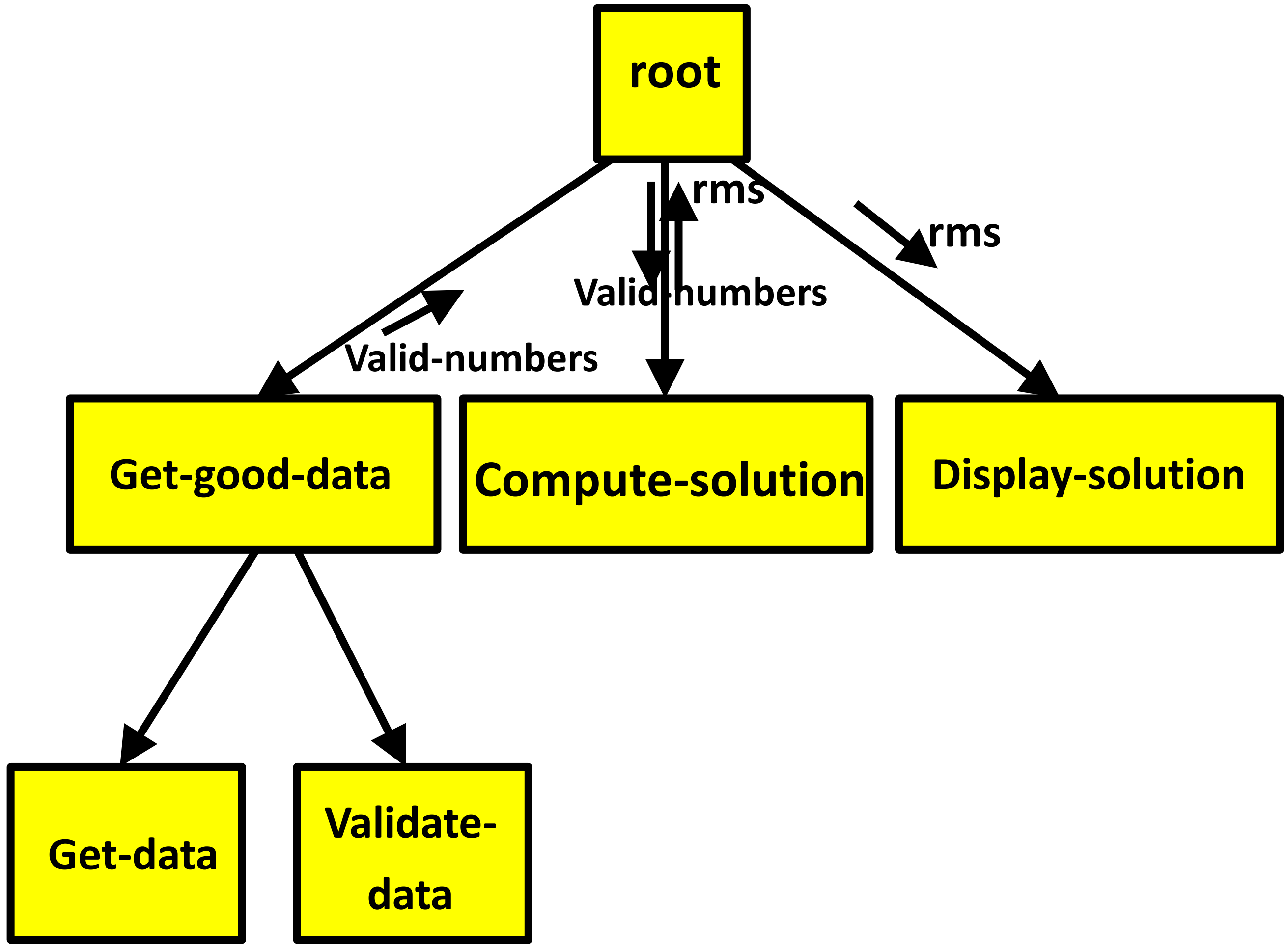
easy to see that the system needs to perform:

- accept the input numbers from the user,
- validate the numbers,
- calculate the root mean square of the input numbers,
- display the result.

Example 1: RMS Calculating Software



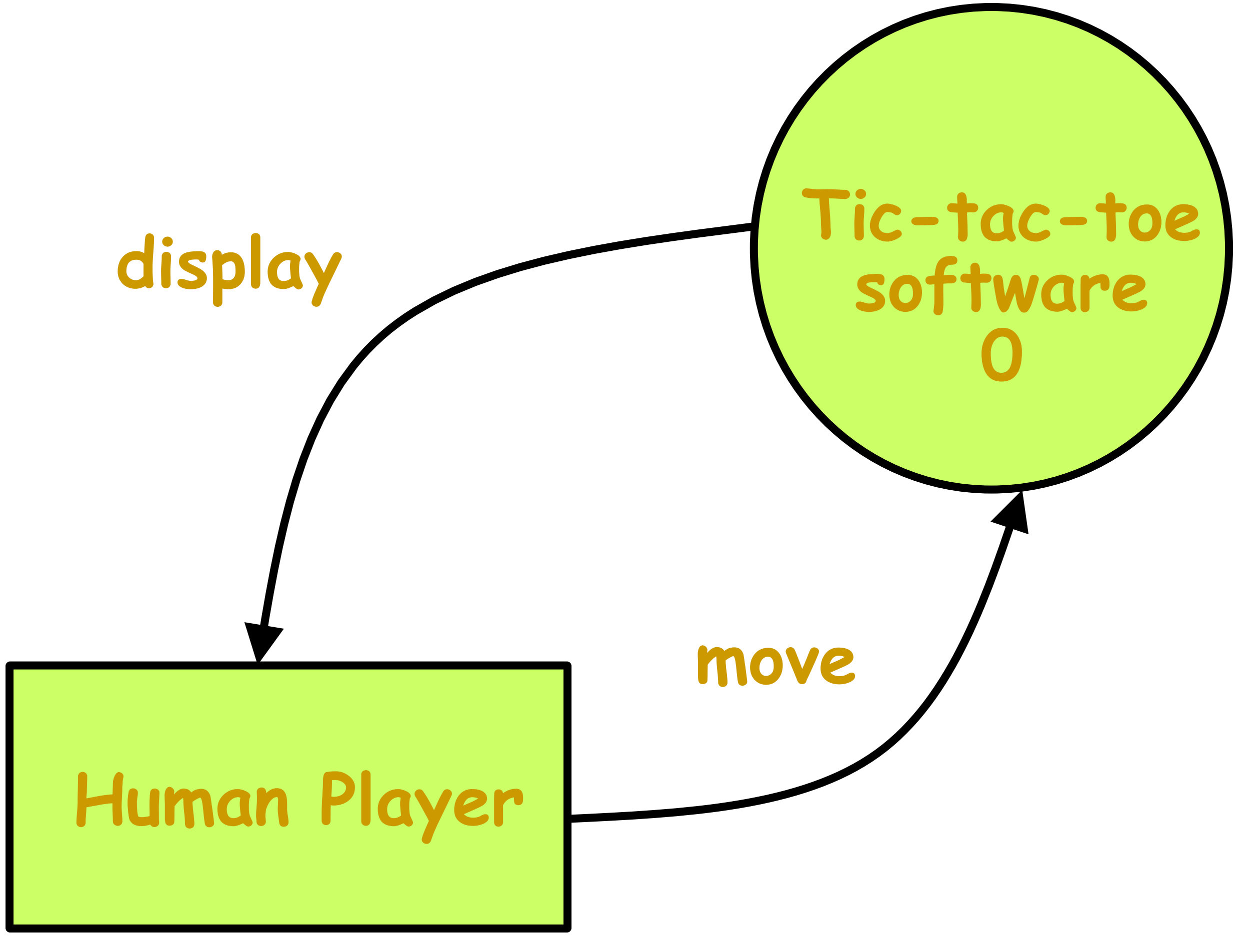
Example 1: RMS Calculating Software



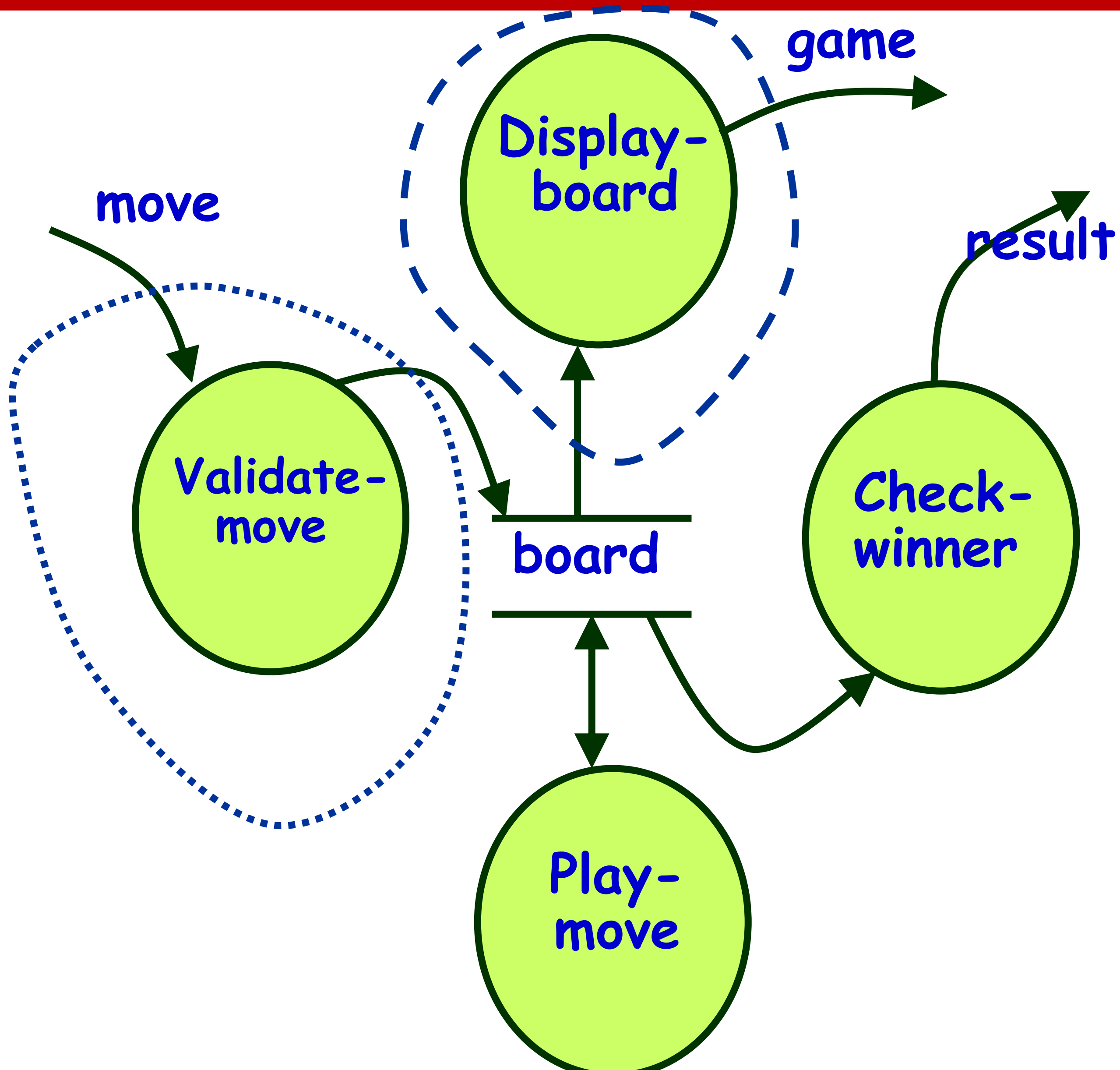
Example 2: Tic-Tac-Toe Computer Game

- As soon as either of the human player or the computer wins,
  - A message congratulating the winner should be displayed.
- If neither player manages to get three consecutive marks along a straight line and all the squares on the board are filled up,
  - Then the game is drawn.
- The computer always tries to win a game.

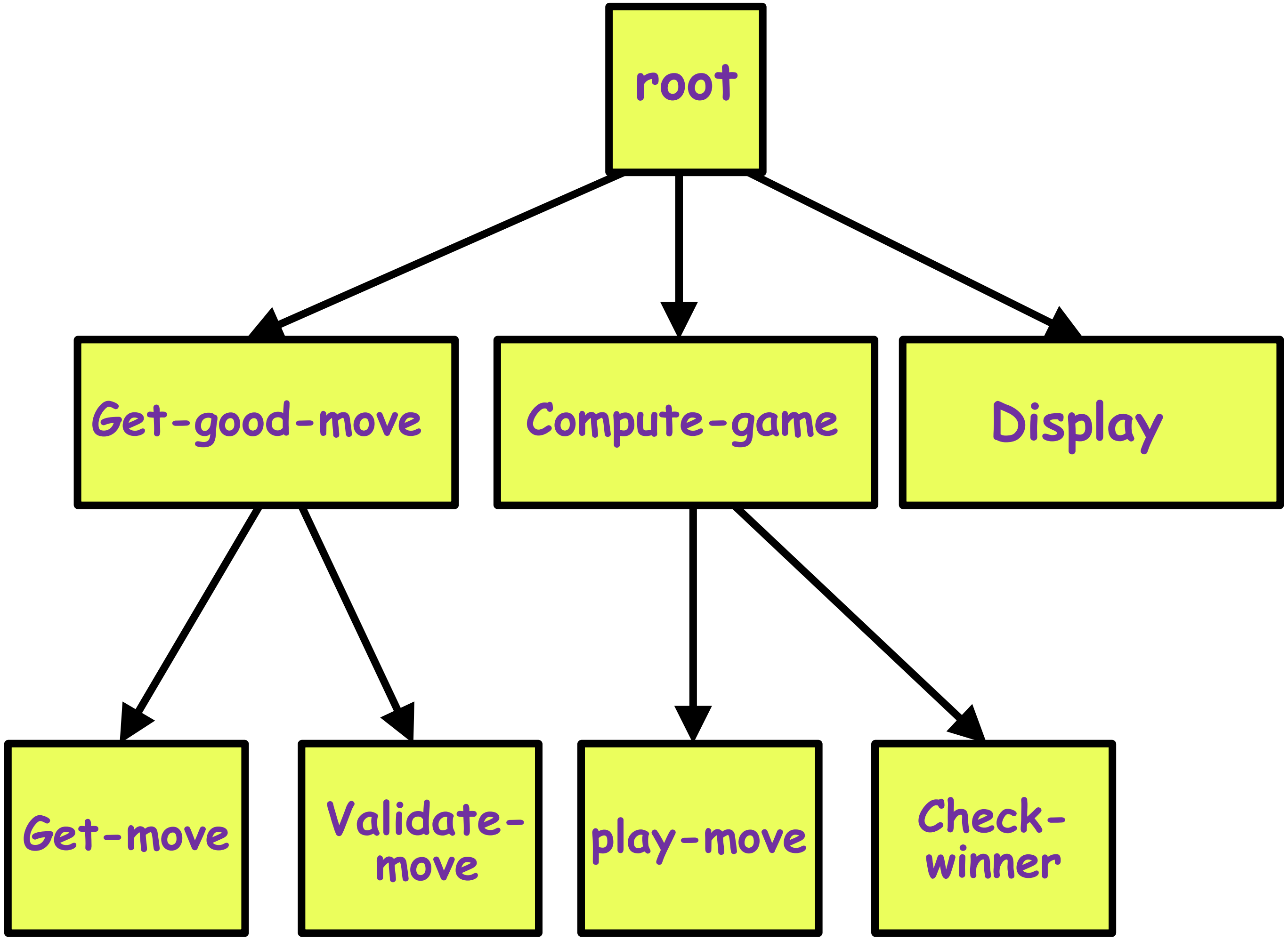
Context Diagram for Example 2



Level 1 DFD


game
Display-board
move
result
Validate-move
board
Check-winner
Play-move

Structure Chart



Transaction Analysis

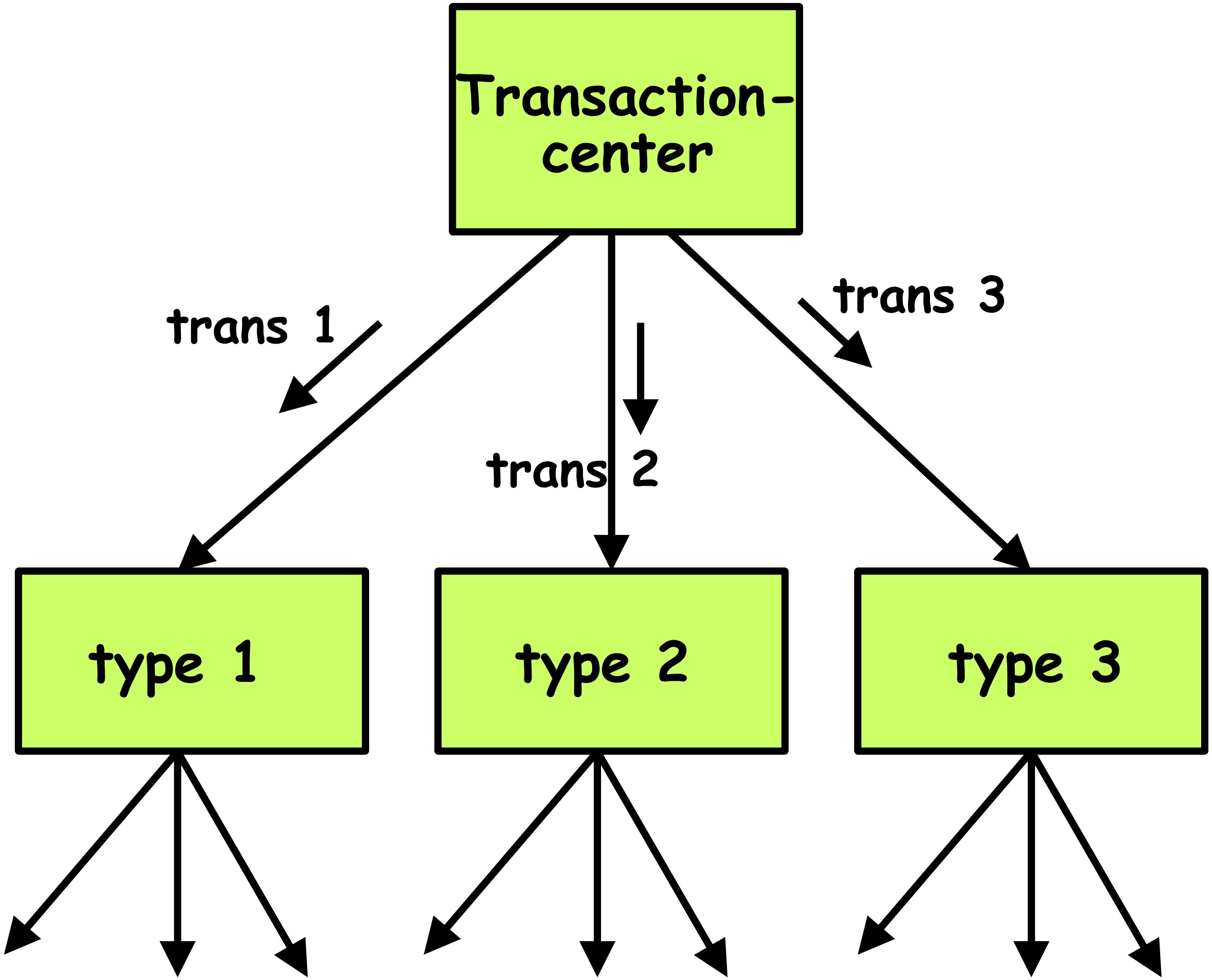
- Useful for designing transaction processing programs.
  - **Transform-centered systems:**
    - Characterized **by similar processing steps for every data item** processed by input, process, and output bubbles.
  - **Transaction-driven systems,**
    - **One of several possible paths** through the DFD is traversed depending upon the input data value.

Transaction Analysis

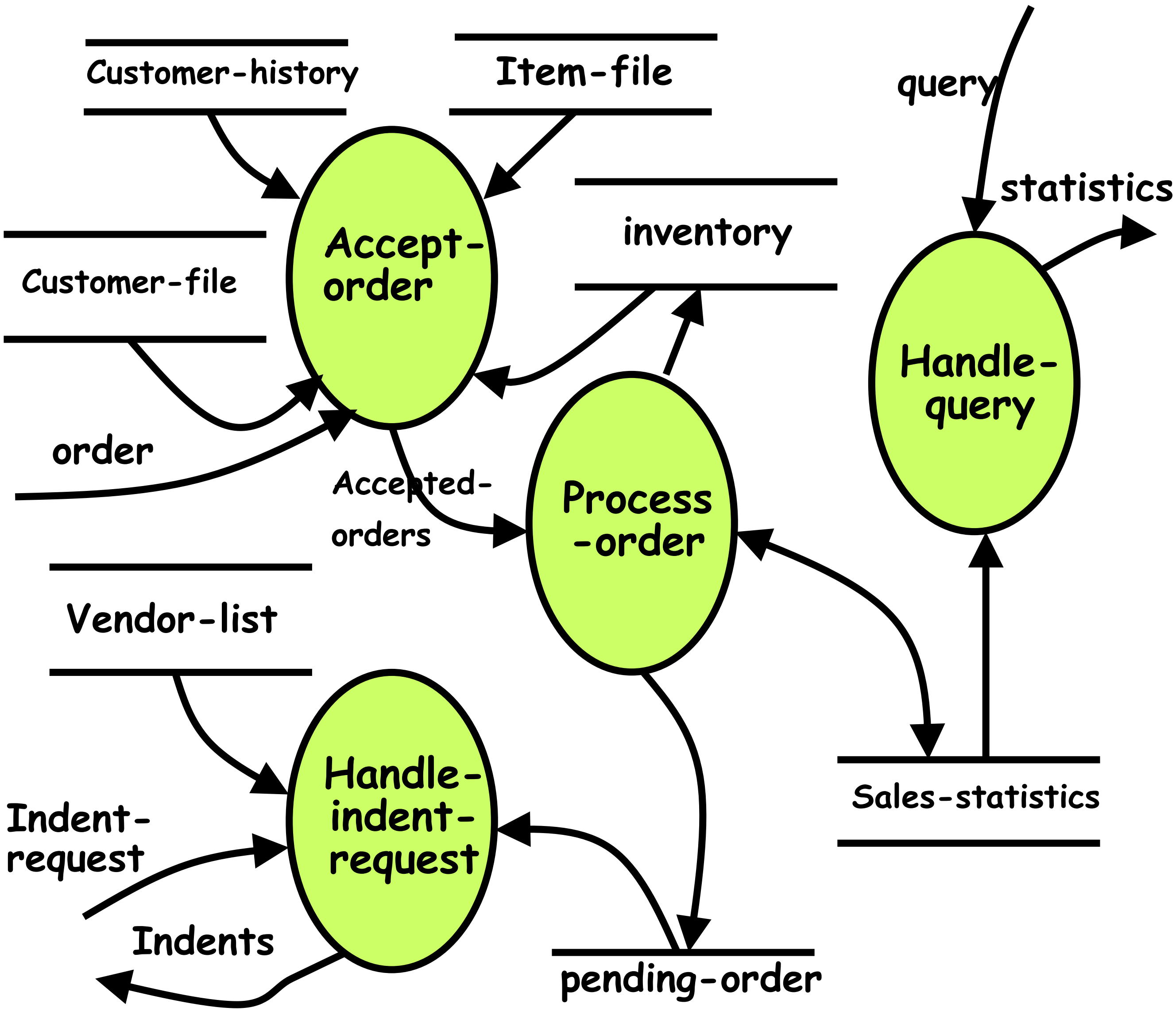
- **Transaction:**
  - Any input data value that triggers an action:
  - For example, a menu option selection might trigger a set of functions.
  - Represented by a tag identifying its type.

- Transaction analysis uses this tag to divide the system into:
  - **Several transaction modules**
  - **One transaction-center module.**

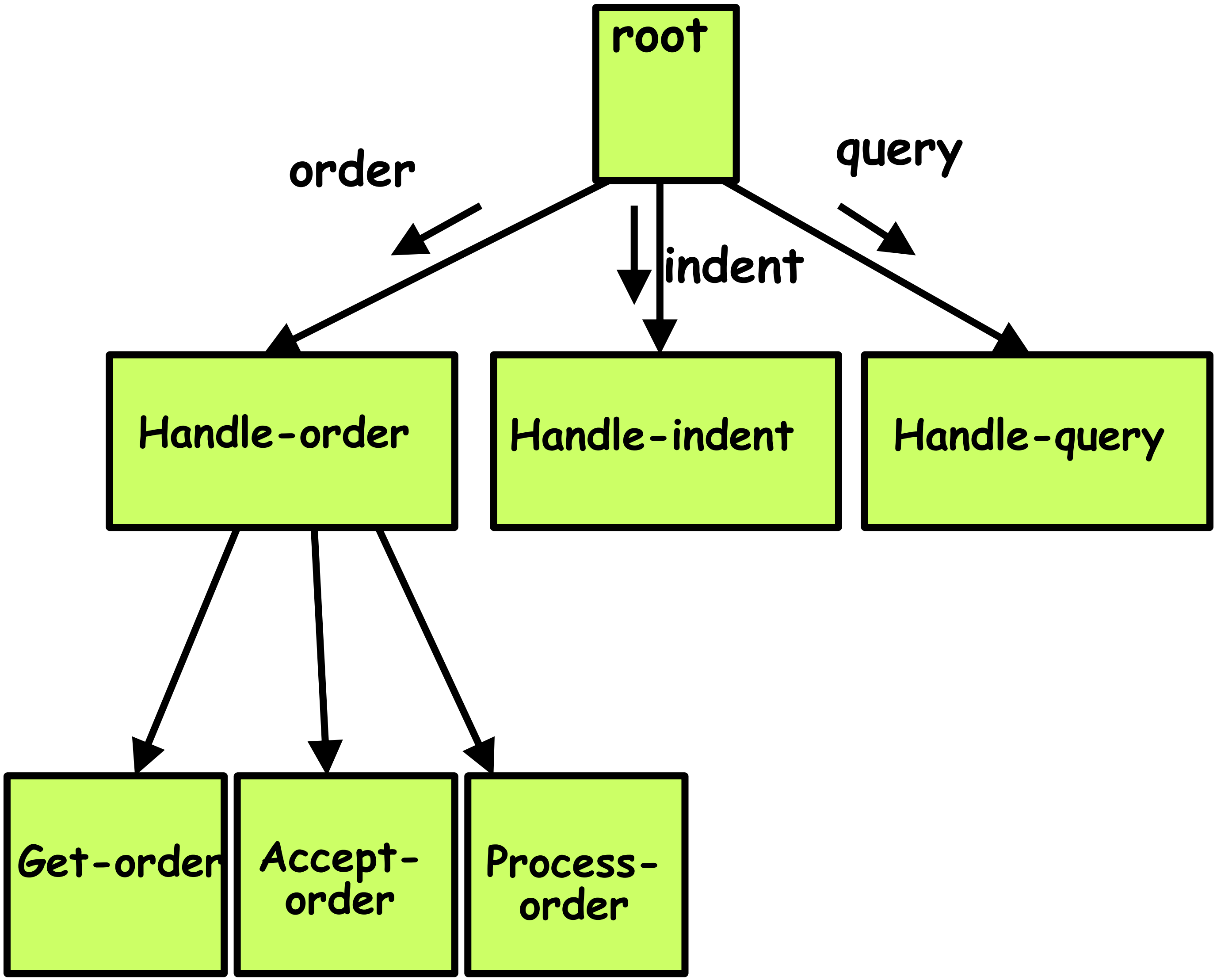
Transaction analysis



Level 1 DFD for TAS



Structure Chart



Summary

- We discussed a sample function-oriented software design methodology:
  - Structured Analysis/Structured Design(SA/SD)
  - Incorporates features from some important design methodologies.

- SA/SD consists of two parts:
  - Structured analysis
    - During structured analysis, the user requirements are converted into a graphical format using DFD.
  - Structured design.
    - During structured design, The DFD representation is transformed to a structure chart representation.

- Several CASE tools are available:
  - Support structured analysis and design.
  - Maintain the data dictionary,
  - Check whether DFDs are balanced or not.

- **End of Chapter**

Thanks