Function Oriented Design

Lecture#16-18

Dr. Sanjeev Patel

Asst. Professor,

Department of Computer Science and Engineering

National Institute of Technology Rourkela, Odisha

Outline

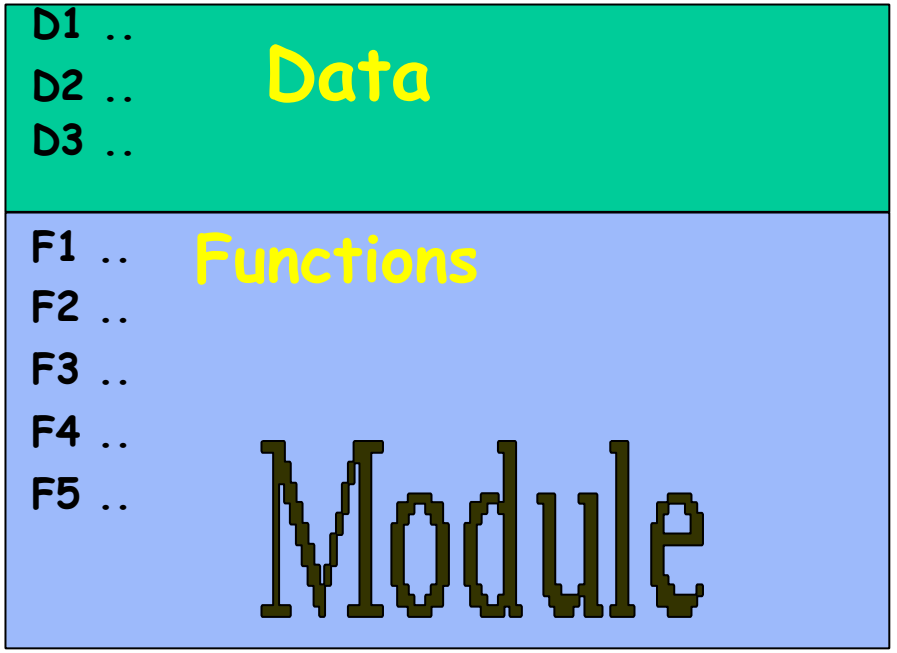
- Items Designed During Design Phase
- What Is a Good Software Design?
- Cohesion and coupling
- Approaches to software design
- Function-oriented software design
- Overview of SA/SD methodology
- Structured analysis
- DFDs
- Structured Design: Structure Chart

Items Designed During Design Phase

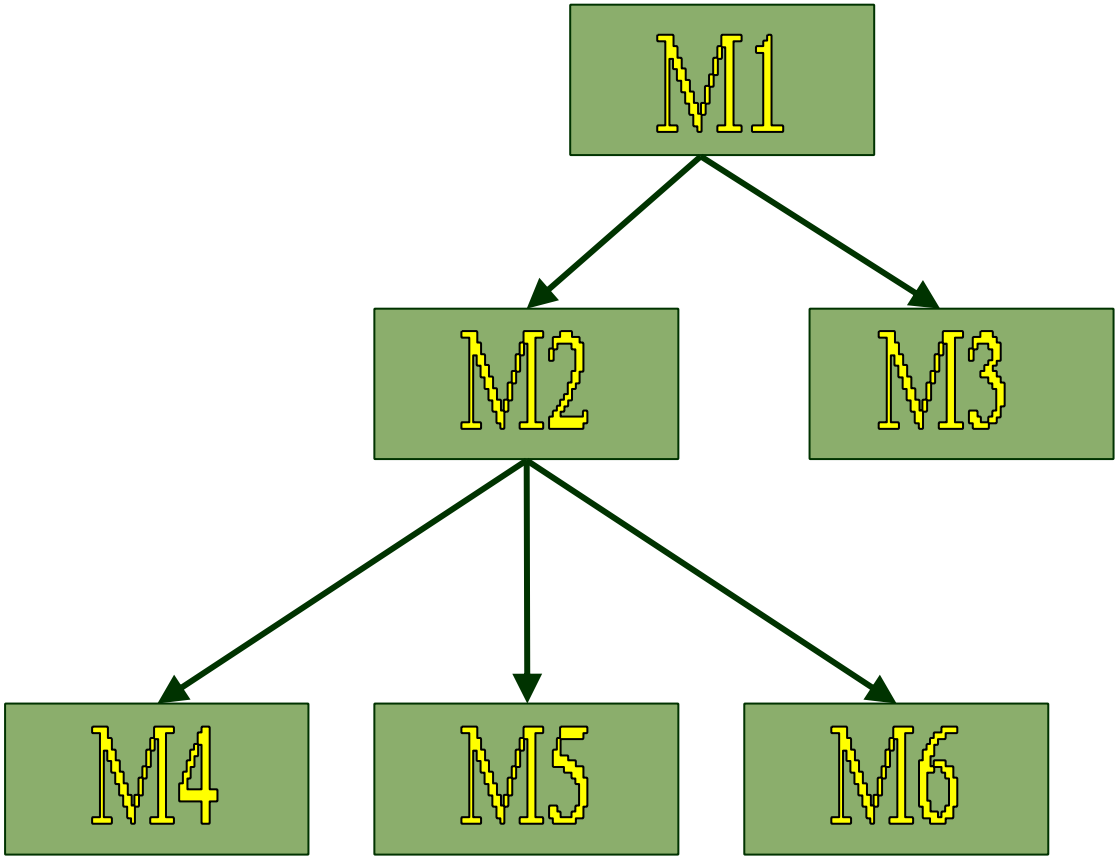
- Module structure,
- Control relationship among the modules
  - call relationship or invocation relationship
- Interface among different modules,
  - data items exchanged among different modules,
- Data structures of individual modules,
- Algorithms for individual modules.

Module

- A module consists of:
  - several functions
  - associated data structures.



Module Structure



What Is a Good Software Design?

- Should implement all functionalities of the system correctly.
- **Should be easily understandable.**
- Should be efficient.
- Should be easily amenable to change,
  - i.e. easily maintainable.

What Is Good Software Design?

- Understand-ability of a design is a major issue:
  - Largely determines goodness of a design:
  - a design that is easy to understand:
    - also easy to maintain and change.

What Is a Good Software Design?

- Unless a design is easy to understand,
  - Tremendous effort needed to maintain it
  - We already know that about 60% effort is spent in maintenance.

- If the software is not easy to understand:
  - maintenance effort would increase many times.

How to Improve Understand-ability?

- Use consistent and meaningful names
  - for various design components,
- Design solution should consist of:
  - A set of well decomposed modules **(modularity)**,
- Different modules should be neatly arranged in a hierarchy:
  - A tree-like diagram.
  - Called Layering

Modularity

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

Modularity

- If modules are independent:
  - Each module can be understood separately,
    - reduces complexity greatly.
  - To understand why this is so,
    - remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.

Modularity

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

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

Cohesion and Coupling

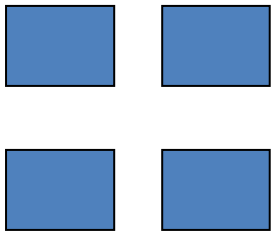
- Cohesion is a measure of:
  - functional strength of a module.
  - **A cohesive module performs a single task or function.**

- Coupling between two modules:
  - **A measure of the degree of interdependence or interaction between the two modules.**

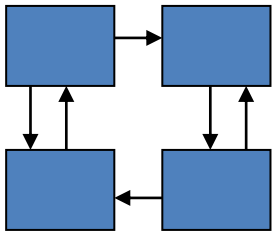
Cohesion and Coupling

- 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.

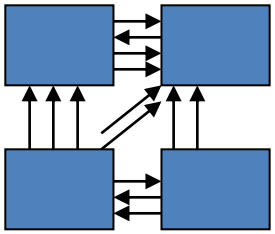
Coupling: Degree of dependence among components



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.*

Design Approaches

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

Design Approaches

- 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

- The system state is centralized:
  - accessible to different functions,
  - For example: member-records available for reference and updating the several functions:
    - create-new-member
    - delete-member
    - update-member-record

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.

Object-Oriented Design Example

- **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

- 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.

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

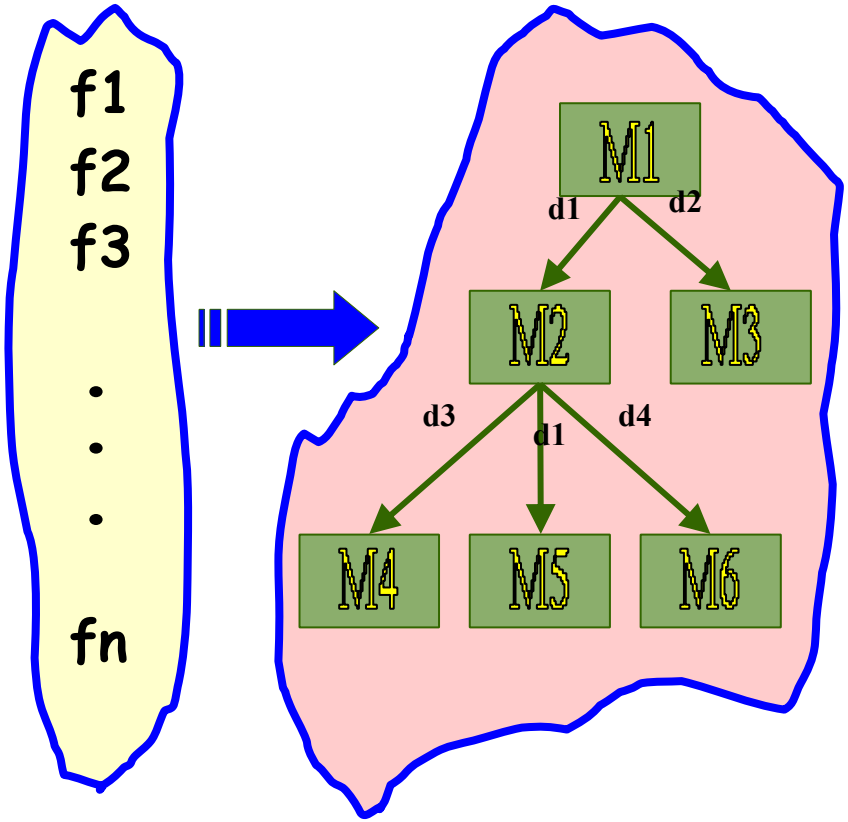
- Though outwardly a system may appear to have been developed in an object oriented fashion,
  - but inside each class there is a small hierarchy of functions designed in a top-down manner.

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#

Function-Oriented Design

- During Structured analysis:
  - High-level functions are successively decomposed:
    - Into more detailed functions.
- During Structured design:
  - The detailed functions are mapped to a module structure.

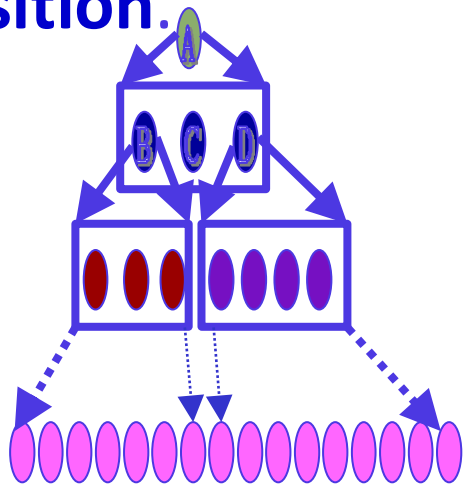


Structured Analysis/Structured Design (SA/SD)

- 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**
- 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.**



- 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.**

Structured Analysis

- 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 vs. Structured Design

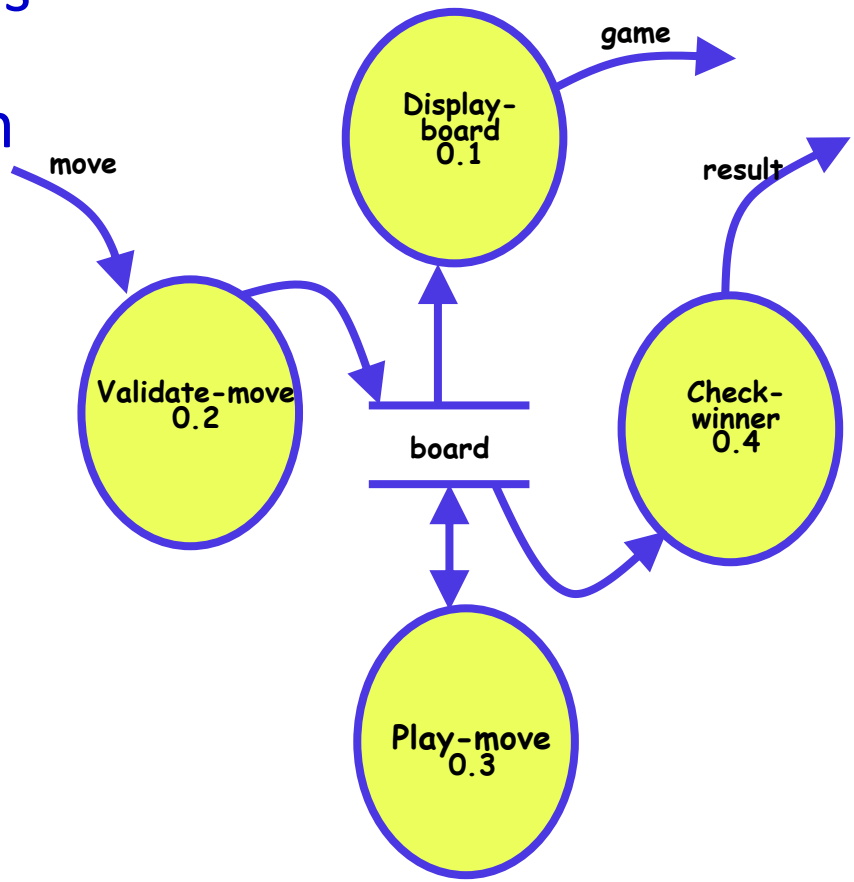
- Purpose of structured analysis:
  - Capture the detailed structure of the system as the user views it.
- Purpose of structured design:
  - Arrive at a form that is suitable for implementation in some programming language.

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).**

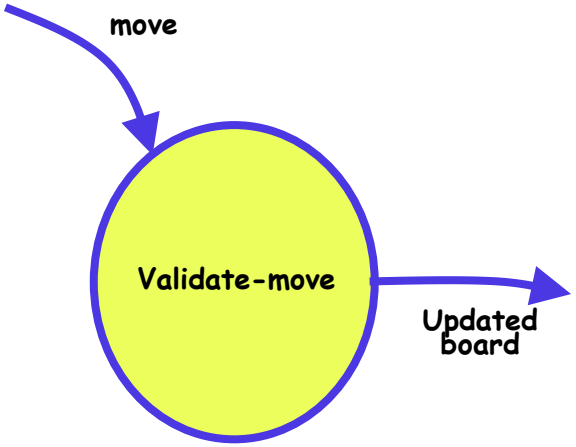
Data Flow Diagram

- 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

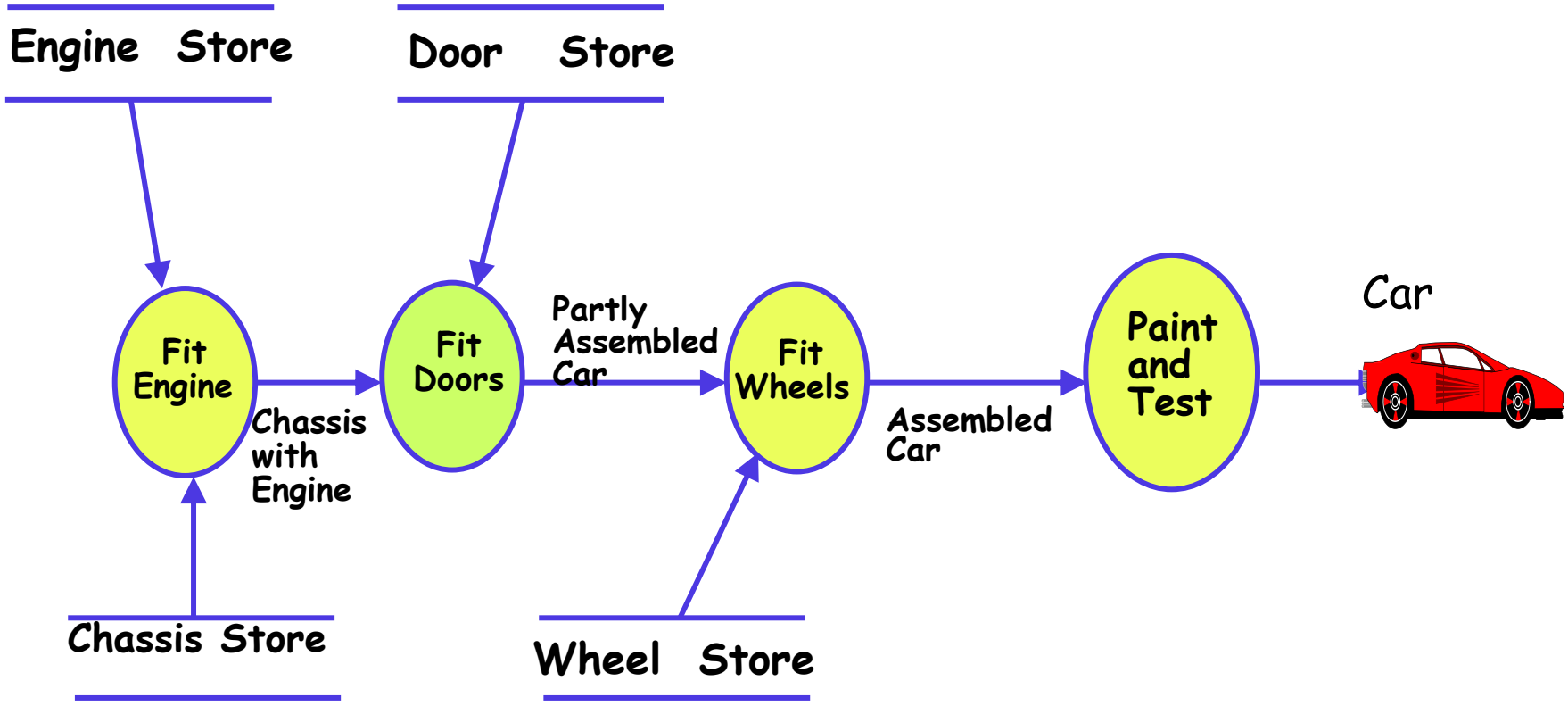


DFD Concepts

- 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



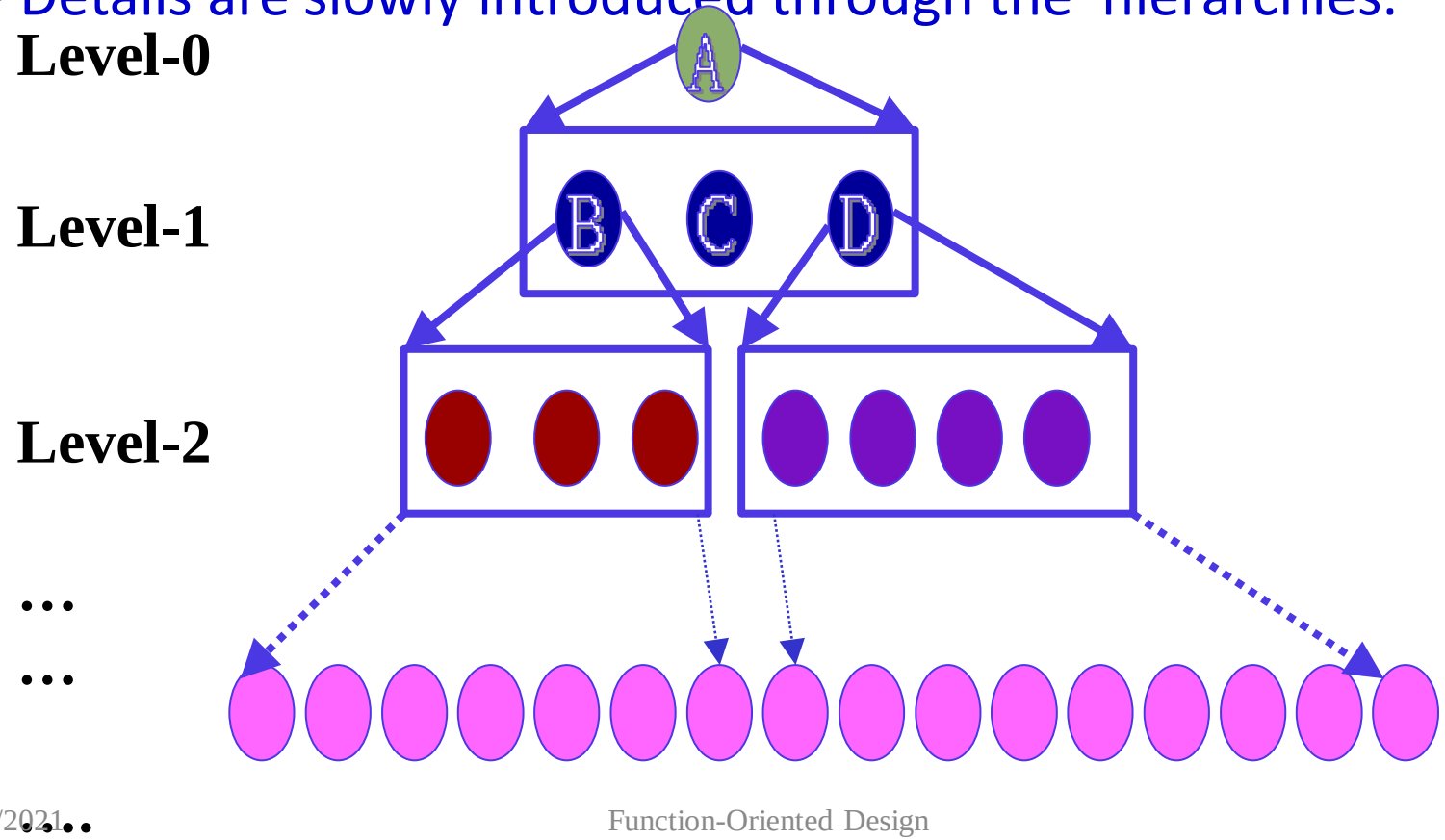
Pros of Data Flow Diagrams (DFDs)

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

Hierarchical Model

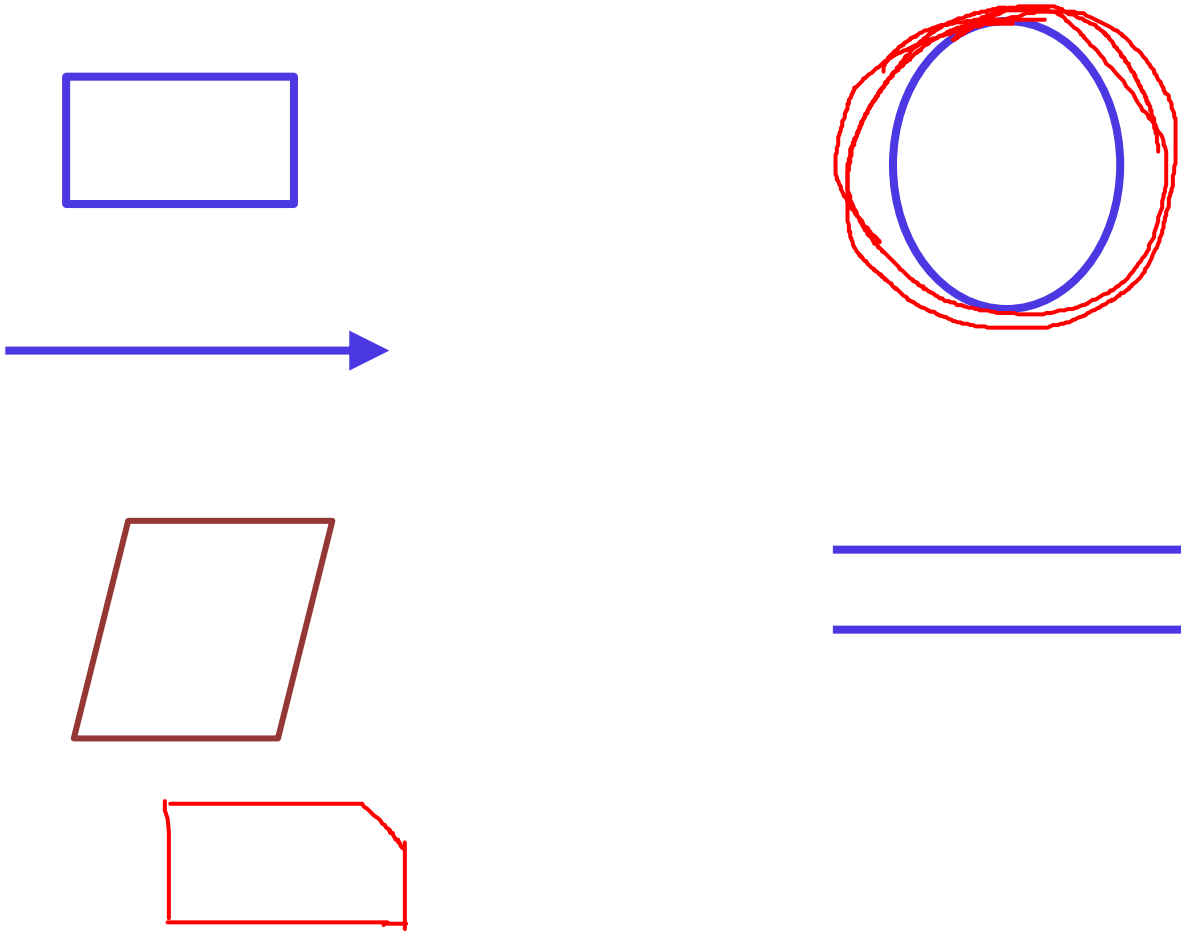
– In a hierarchical model:

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



Data Flow Diagrams (DFDs)

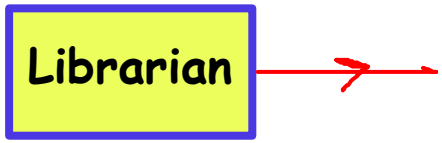
- Basic Symbols Used for Constructing DFDs:



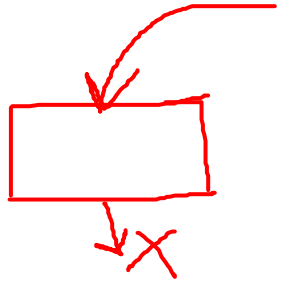
DFD symbol: rectangle

- Rectangle: external Entity Symbol

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

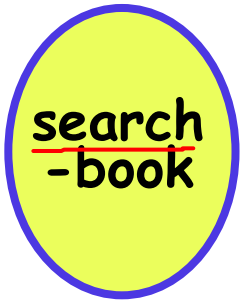


- External entities are either users or external systems:
  - Produces (input) data to the system or
  - consume data produced by the system.
  - Sometimes external entities are called **terminator, source, or sink.**



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.**

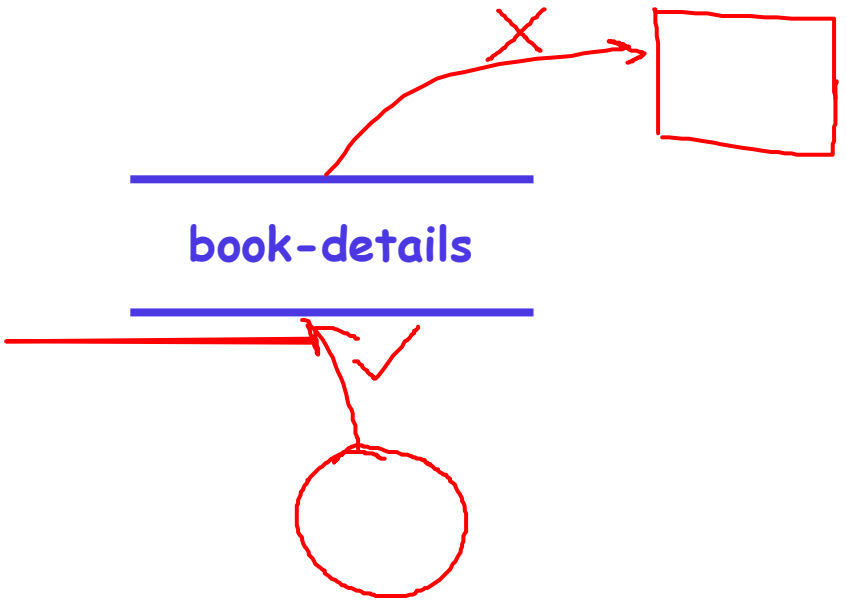
Data Flow Symbol

- 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

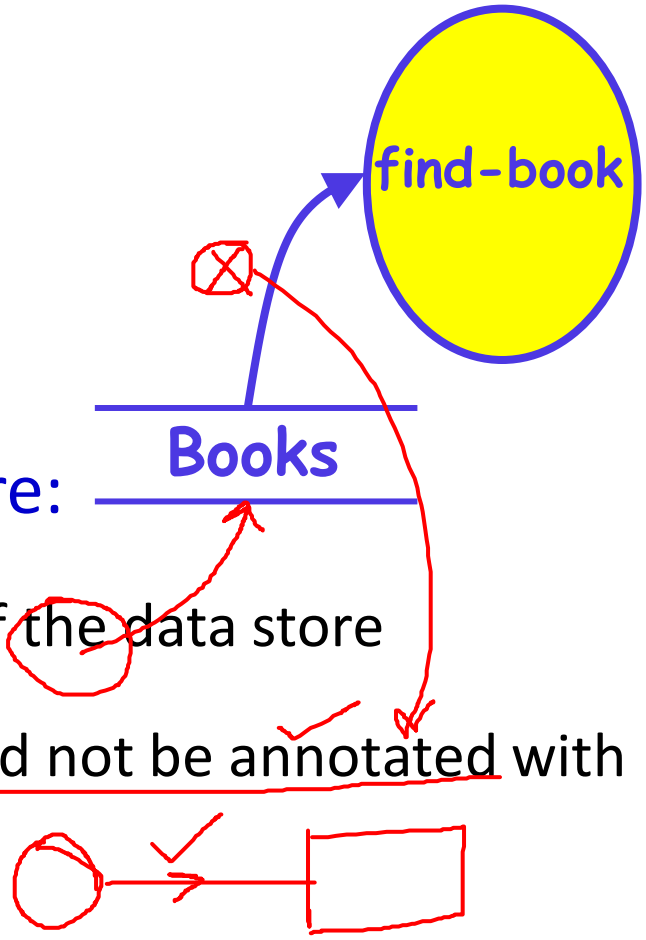
- Represents a logical file:
  - A logical file can be:
    - **a data structure**
    - **a physical file on disk.**



  - Each data store is connected to a process (not to a external user):
    - By means of a data flow symbol.

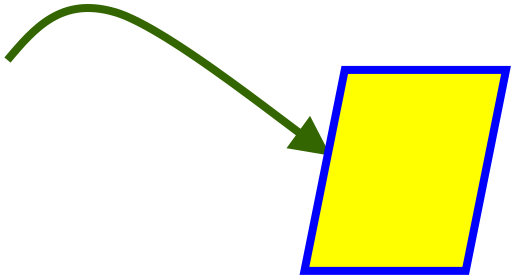
Data Store Symbol

- 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



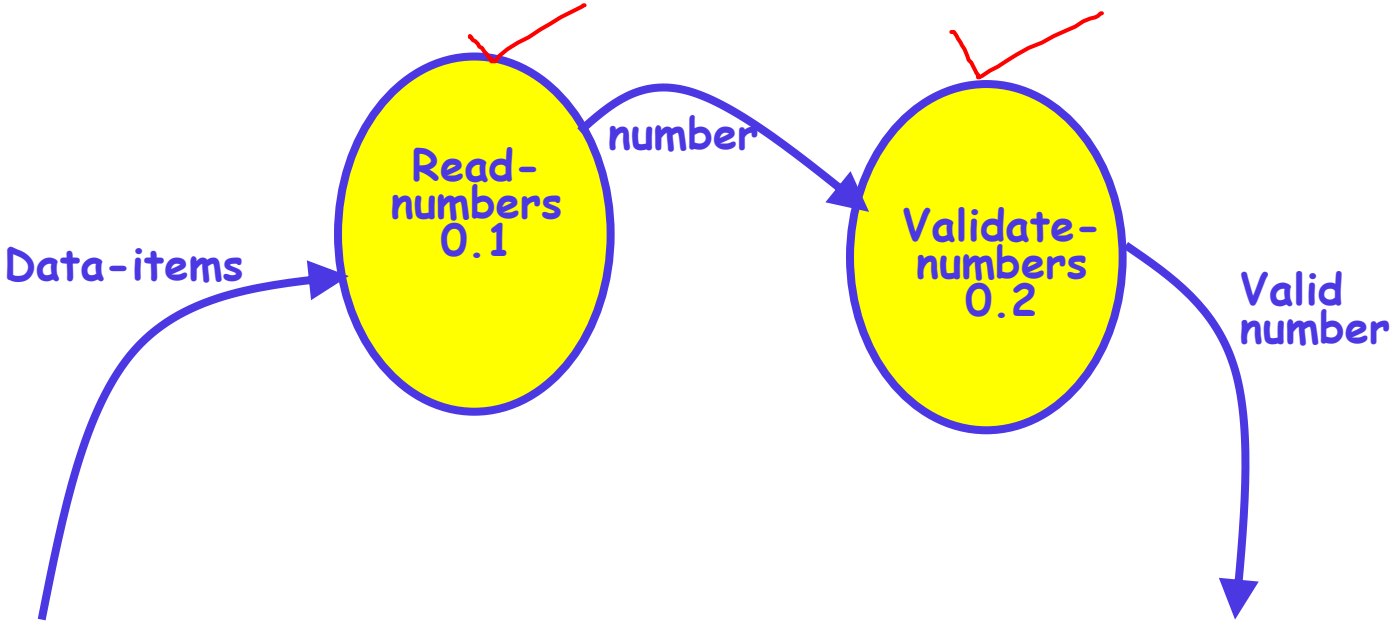
Output Symbol: Parallelogram

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



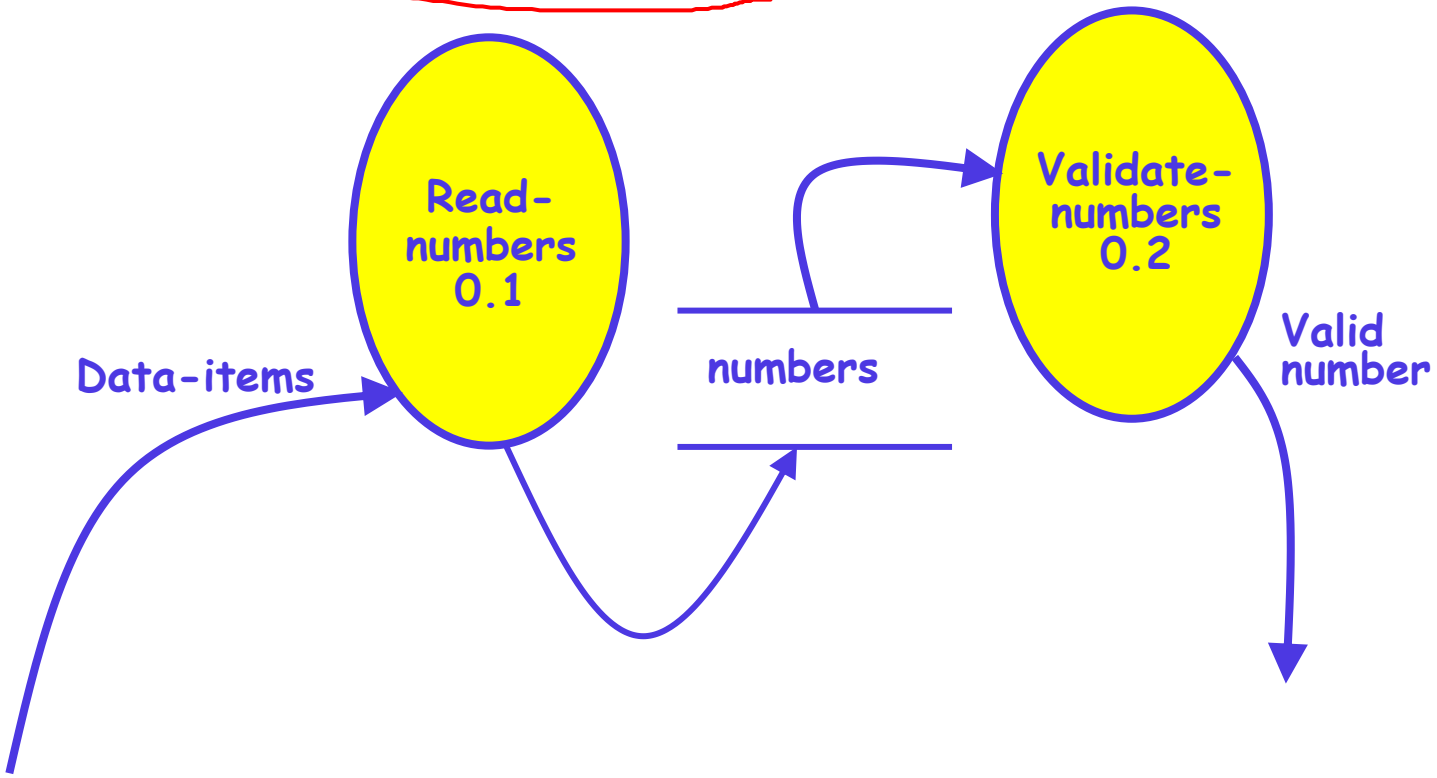
Synchronous Operation

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



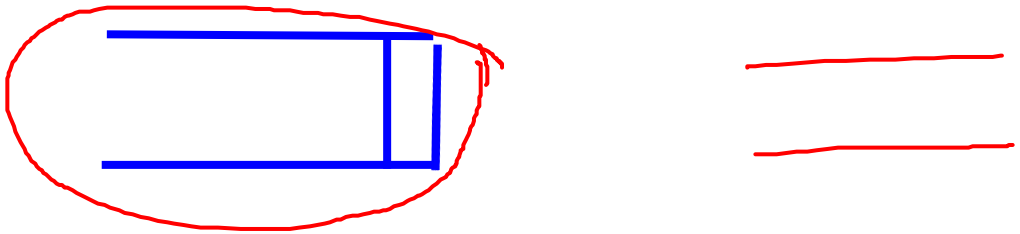
Asynchronous Operation

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



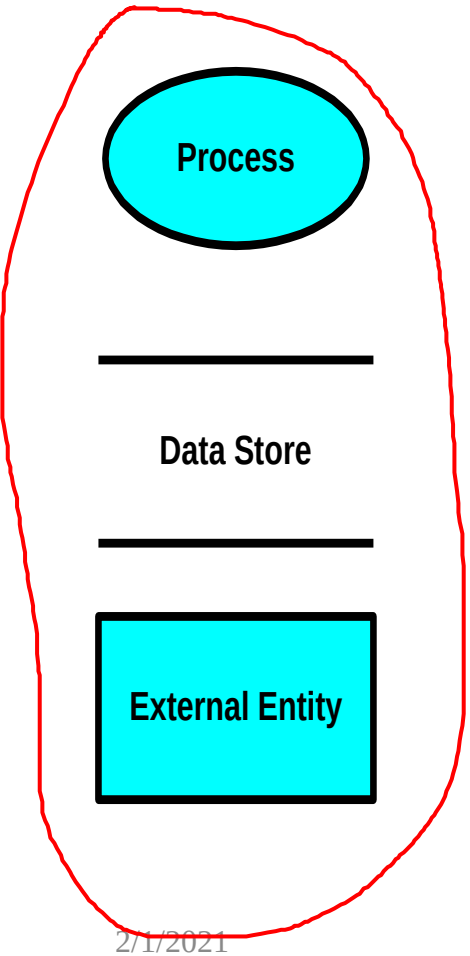
Yourdon's vs. Gane Sarson Notations

- 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

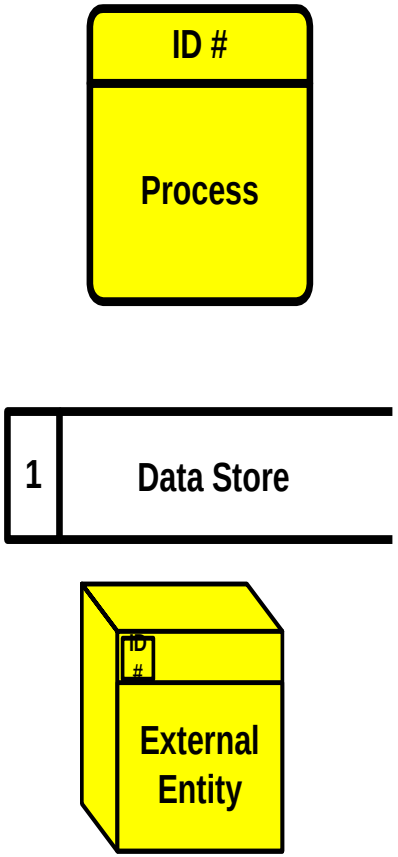


Visio 5.x

From Flow Chart / Data Flow Diagram



From Software Diagram / Gane-Sarson DFD



Visio 2000

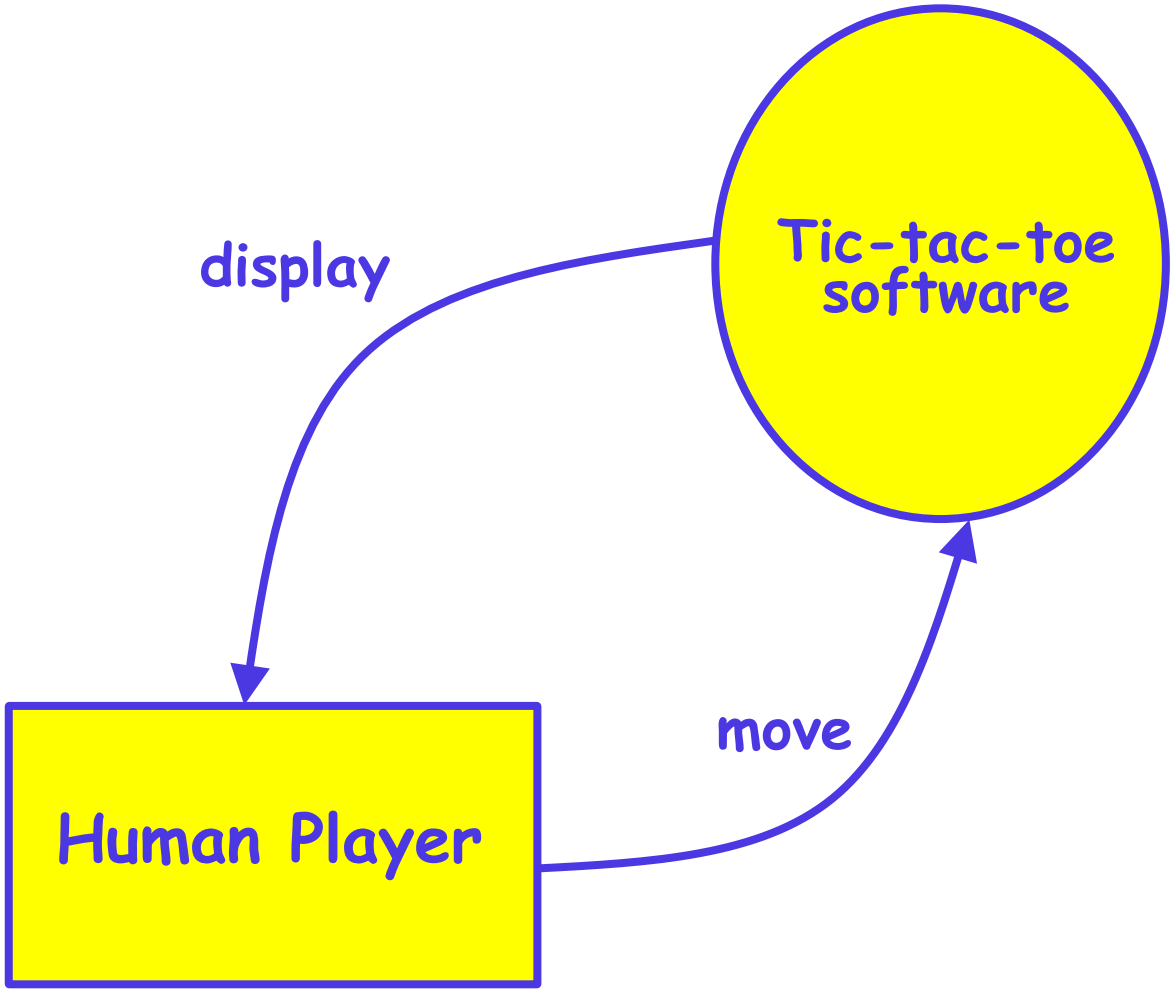
Data Flow Diagram

DFD Shapes from Visio

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.**

Tic-tac-toe: Context Diagram

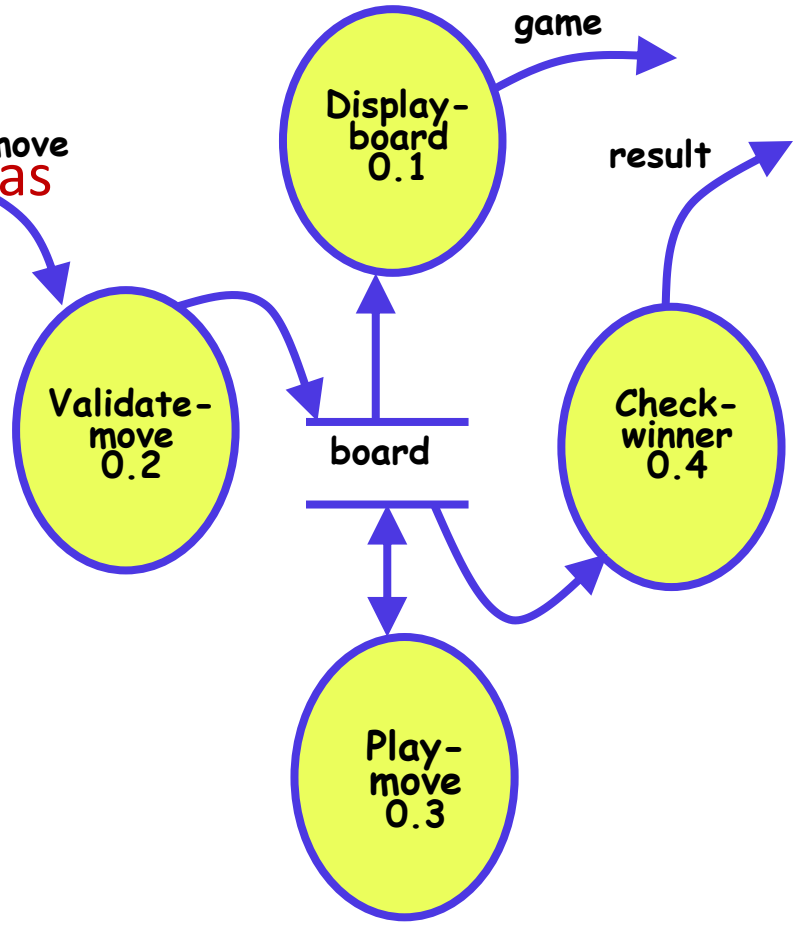


Context Diagram

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

Level 1 DFD Construction

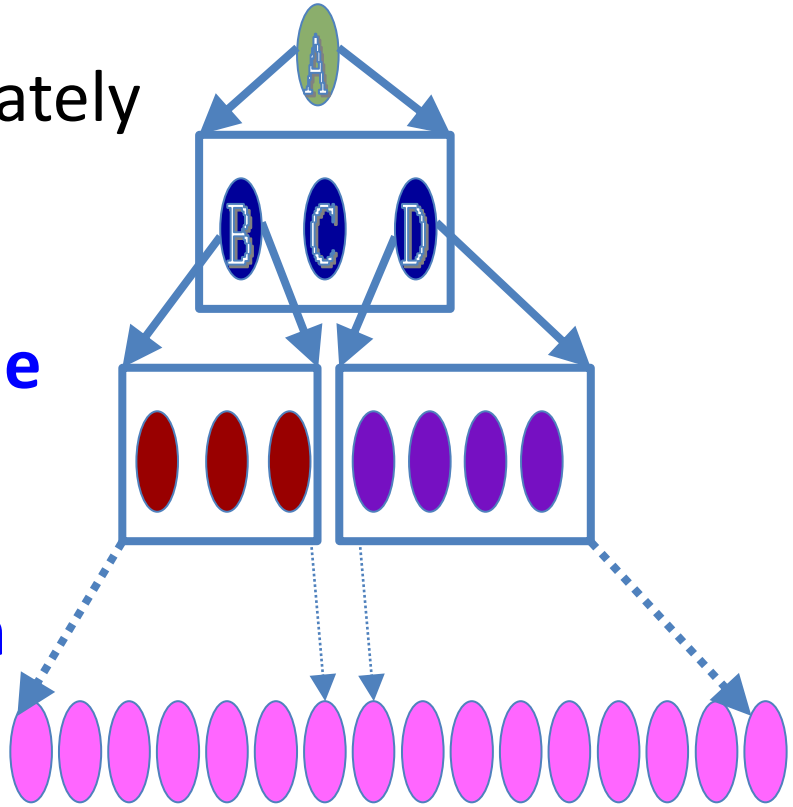
- Examine the SRS document:
  - Represent each high-level function as a bubble.
  - Represent data input to every high-level function.
  - Represent data output from every high-level function.



Tic-tac-toe example

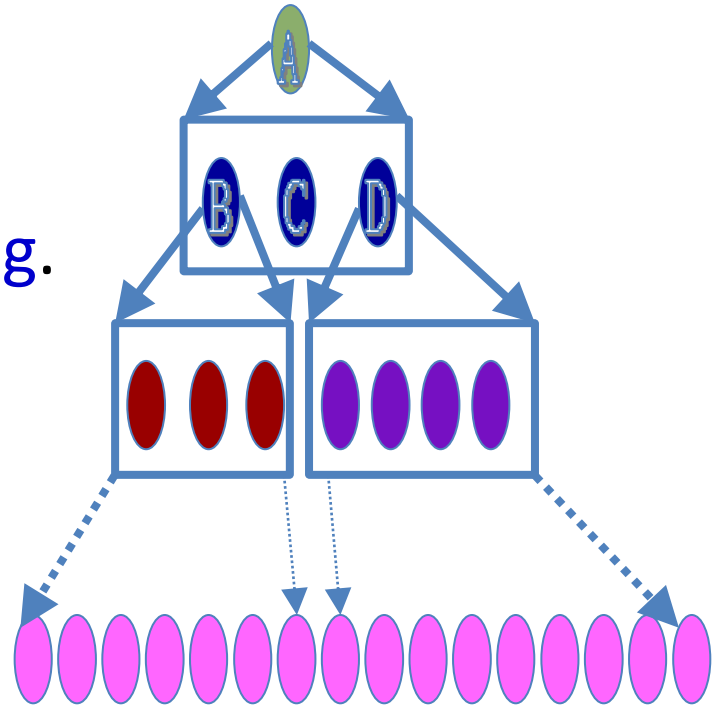
Higher Level DFDs

- 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.

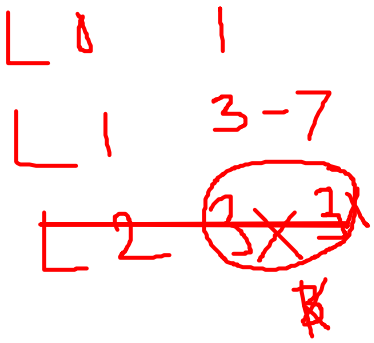


Decomposition

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

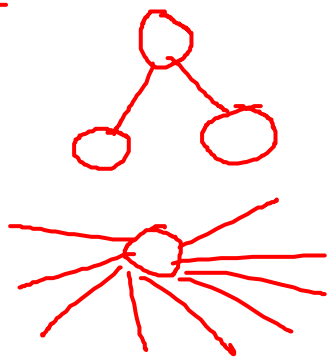


Decomposition Pitfall



- 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.

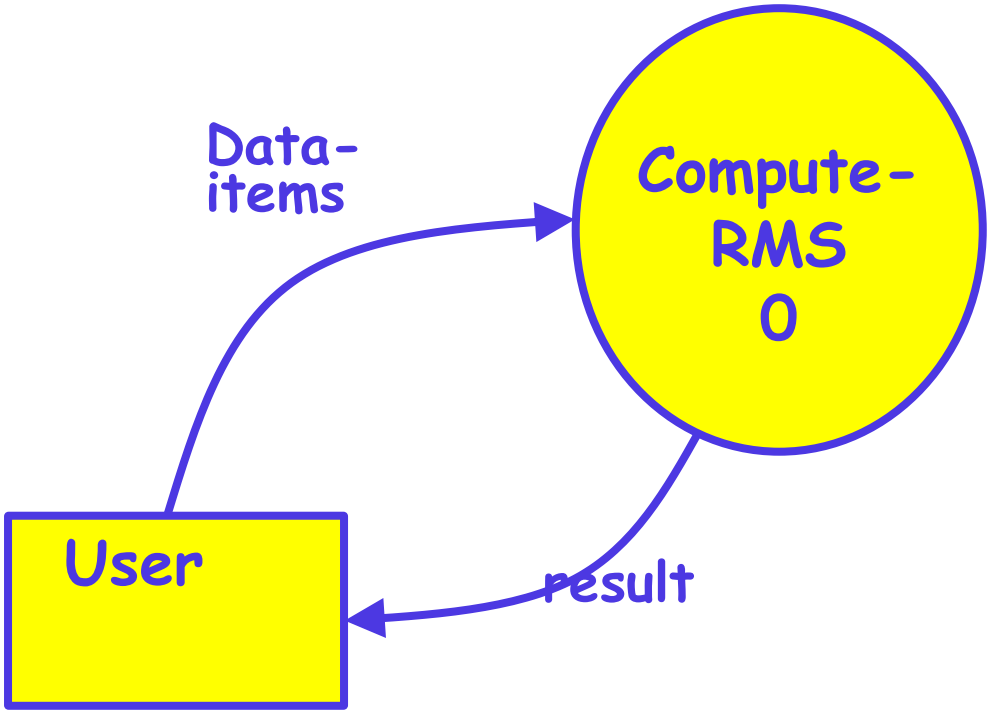
Example 1: RMS Calculating Software

- 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 1: RMS Calculating Software

- The context diagram is simple to develop:
  - The system accepts 3 integers from the user
  - Returns the result to him.

Example 1: RMS Calculating Software



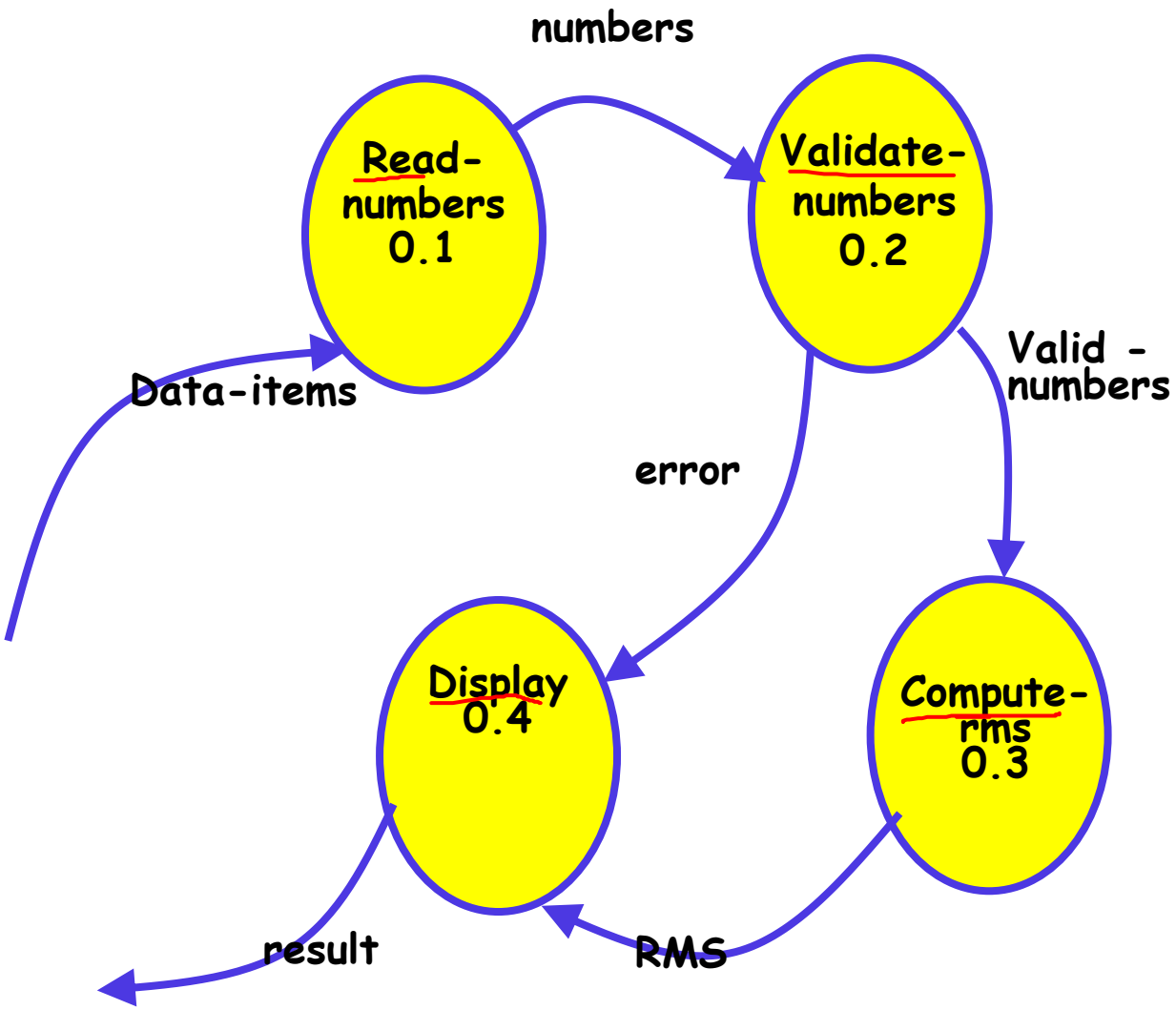
Context Diagram (Level 0 DFD)

Example 1: RMS Calculating Software

- From a cursory analysis of the problem description:
  - We can see that the system needs to perform several things.

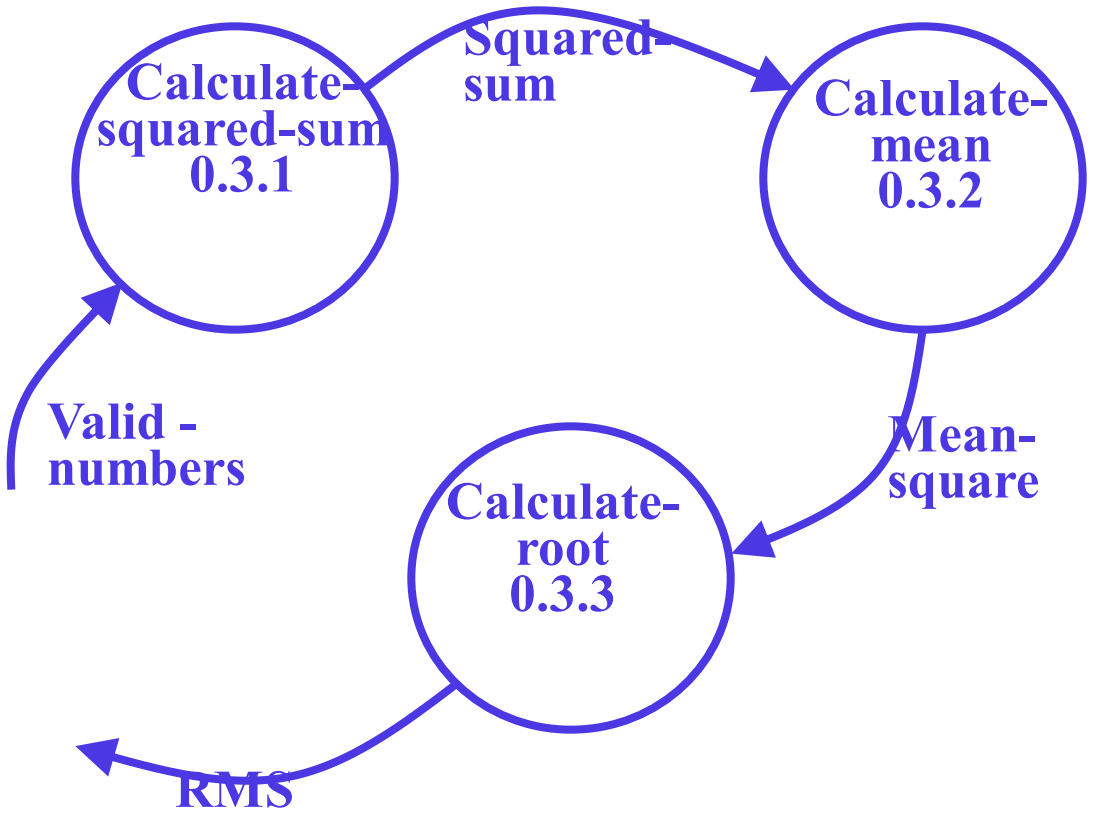
Example 1: RMS Calculating Software

- Accept input numbers from the user:
  - Validate the numbers,
  - Calculate the root mean square of the input numbers
  - Display the result.



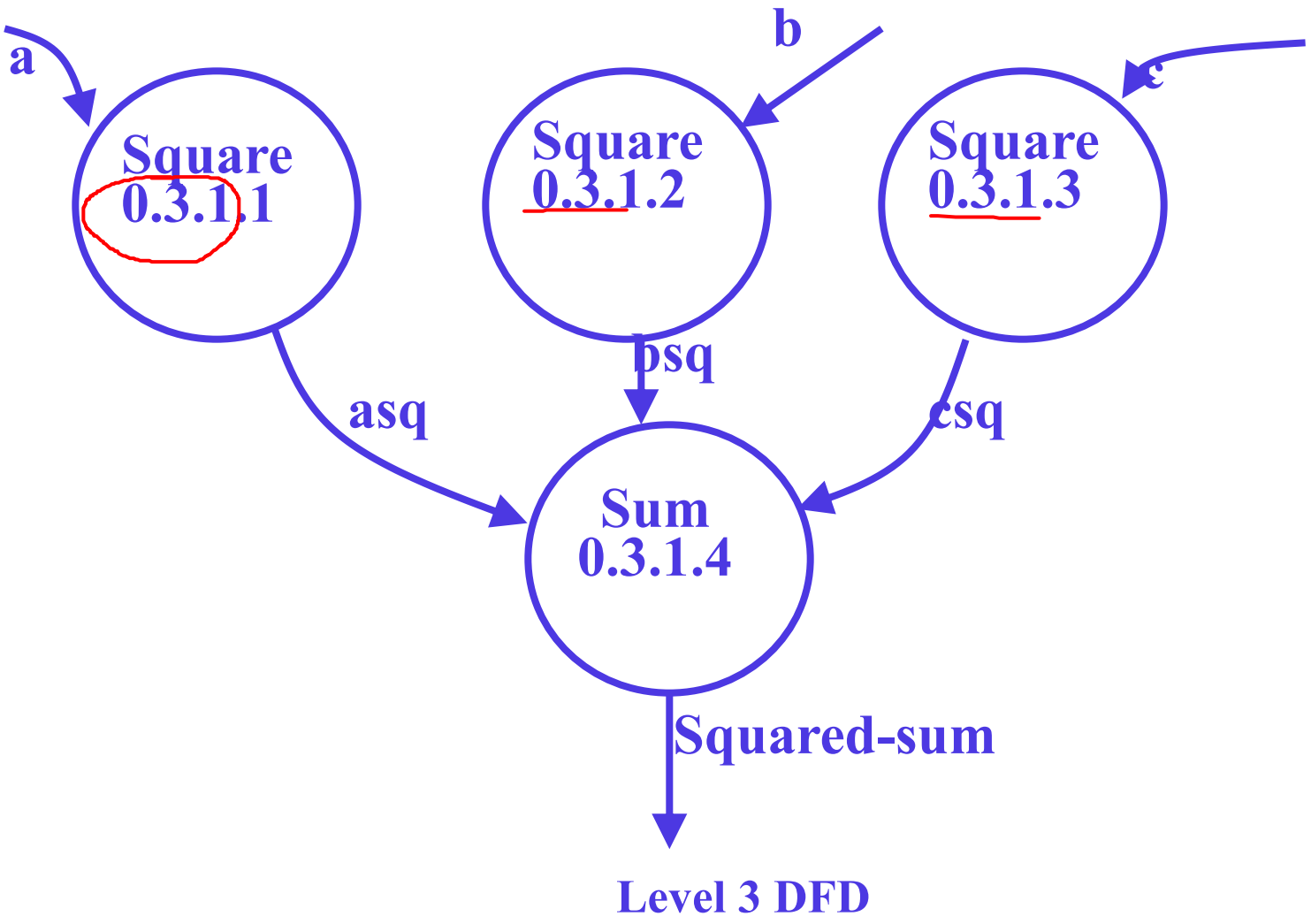
Example 1: Level 1 DFD RMS Calculating Software

Example 1: RMS Calculating Software



Level 2 DFD

Example 1: RMS Calculating Software



Example: 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

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.

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).

- 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

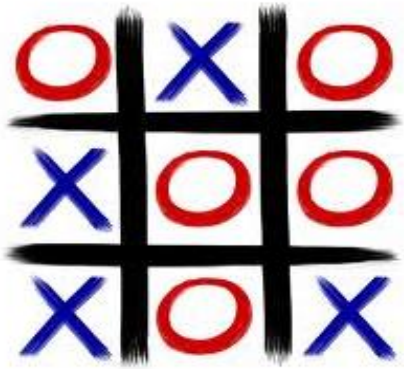
- {name}* represents
  - zero or more instances of name data.
- = represents equivalence,
  - e.g. **a=b+c** means that a represents b and c.
- * *: Anything appearing within * * is considered as comment.

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***

Example 2: Tic-Tac-Toe Computer Game

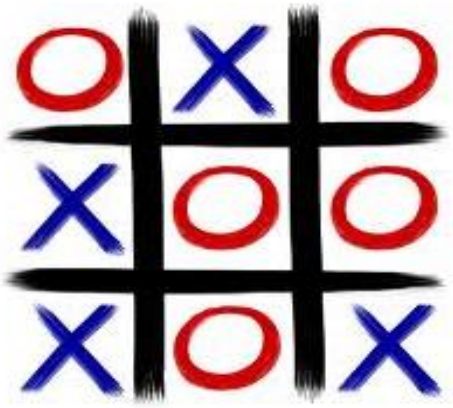
- A human player and the computer make alternate moves on a 3 X 3 square.



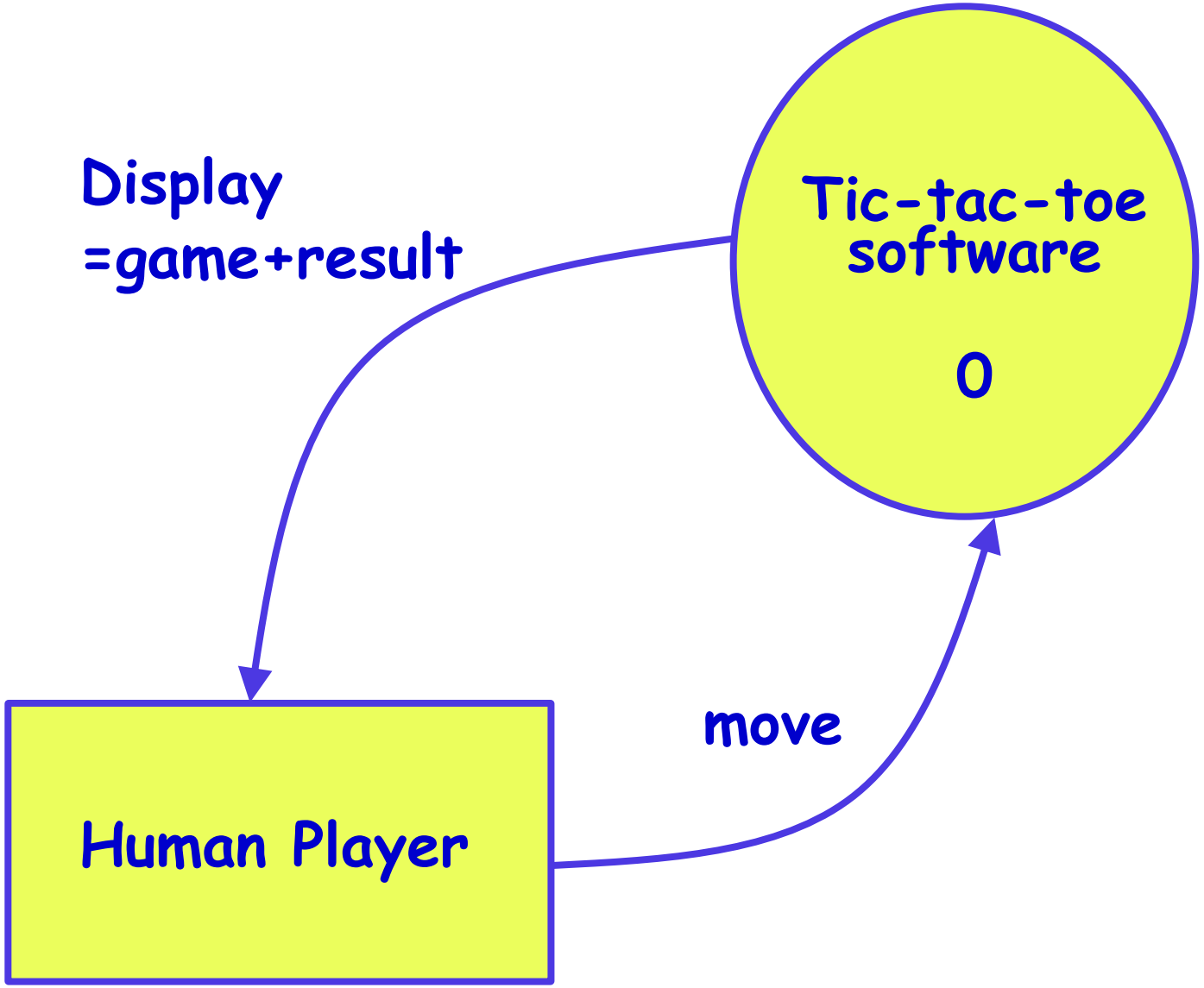
- A move consists of marking a previously unmarked square.
- The user inputs a number between 1 and 9 to mark a square
- Whoever is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins.

Example: Tic-Tac-Toe Computer Game

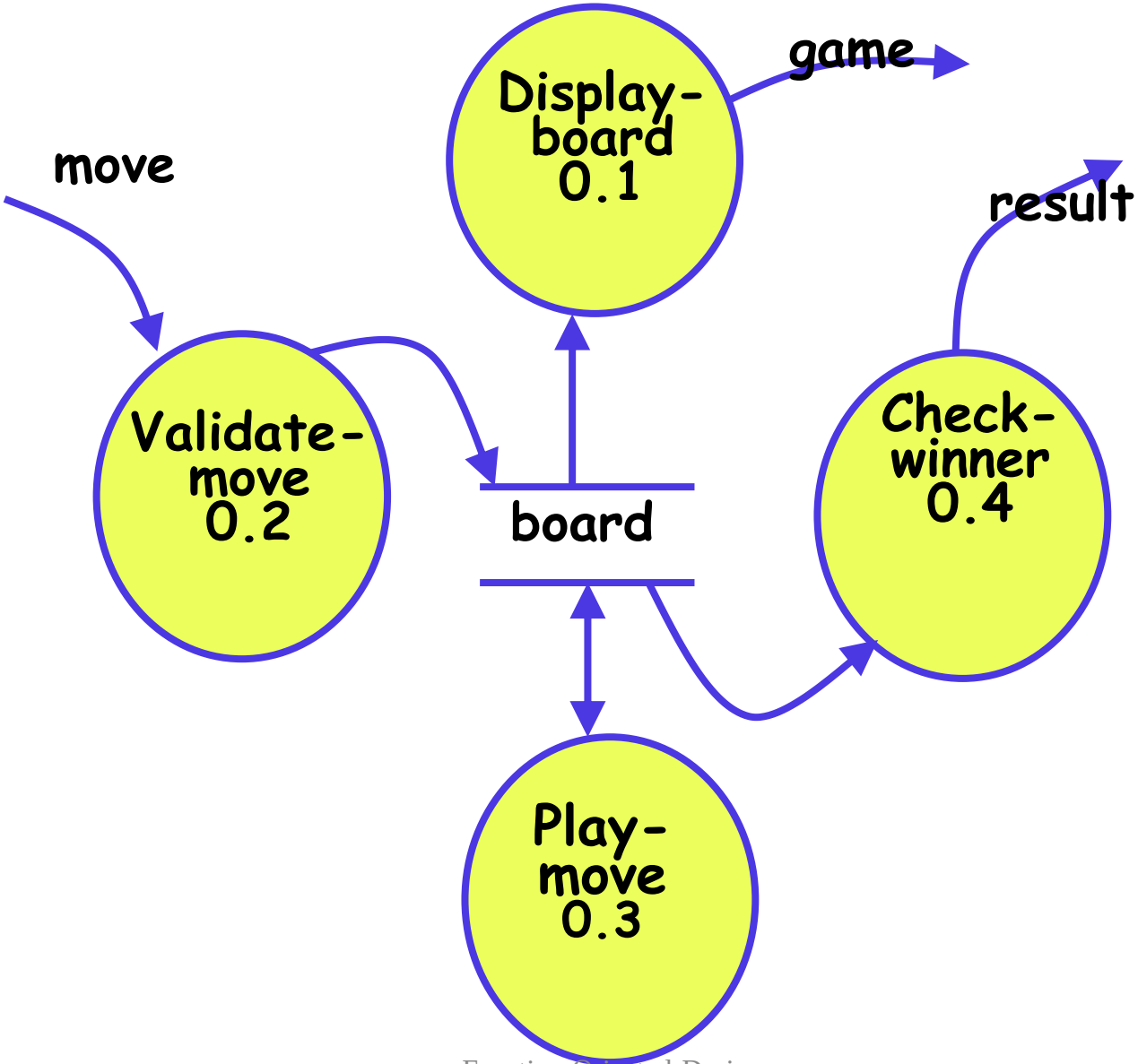
- As soon as either of the human player or the computer wins,
  - A message announcing the winner should be displayed.
- If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up,
  - Then the game is drawn.
- The computer always tries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



Context Diagram: Tic-tac-toe



Level 1 DFD


move
Display-board 0.1
game
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

Observation

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

Observation

- 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.**

Guidelines For Constructing DFDs

- 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.

Guidelines For Constructing DFDs

- 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.**

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.

Shortcomings of the DFD Model

- 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.

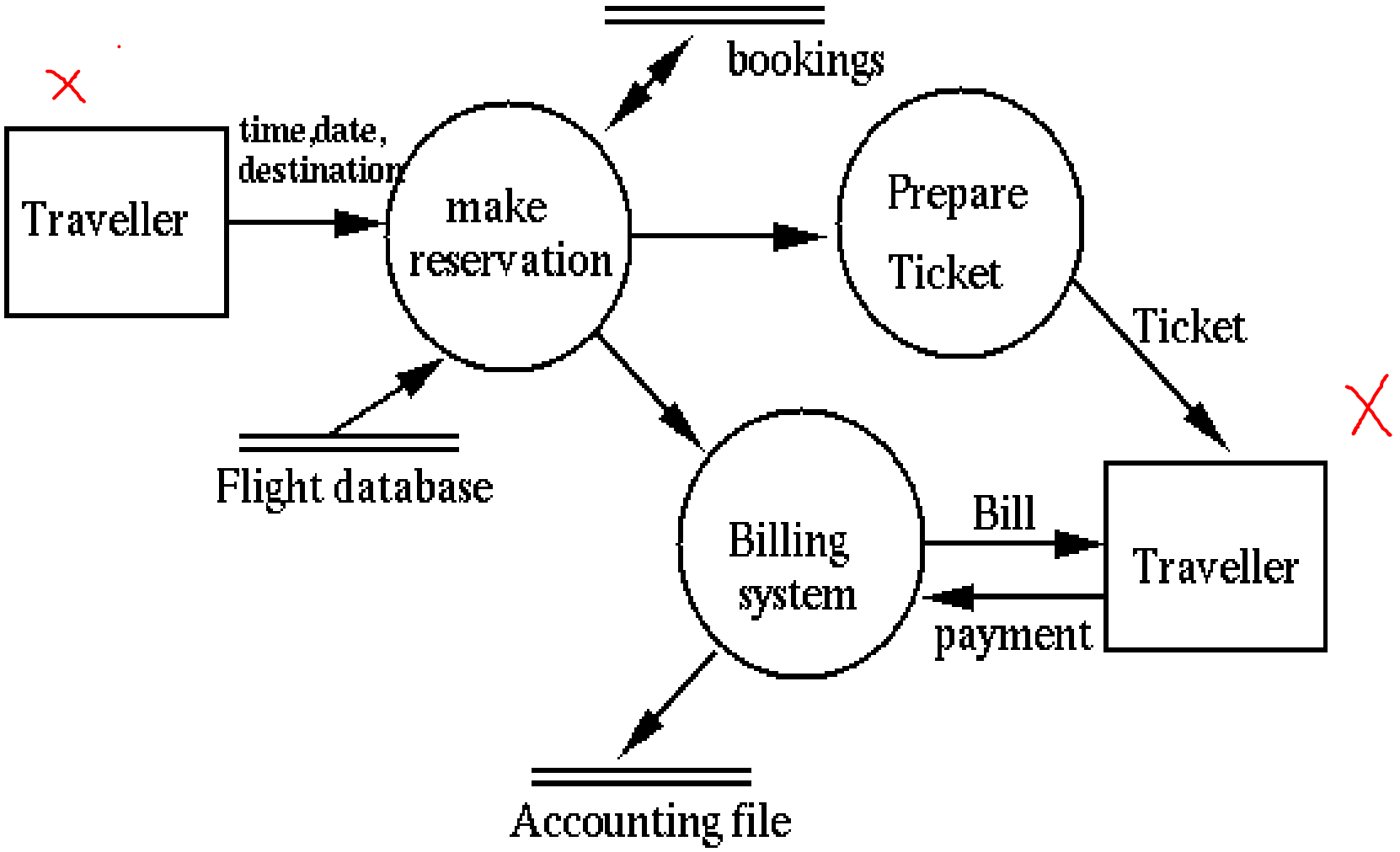
Shortcomings of the DFD Model

- 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.**

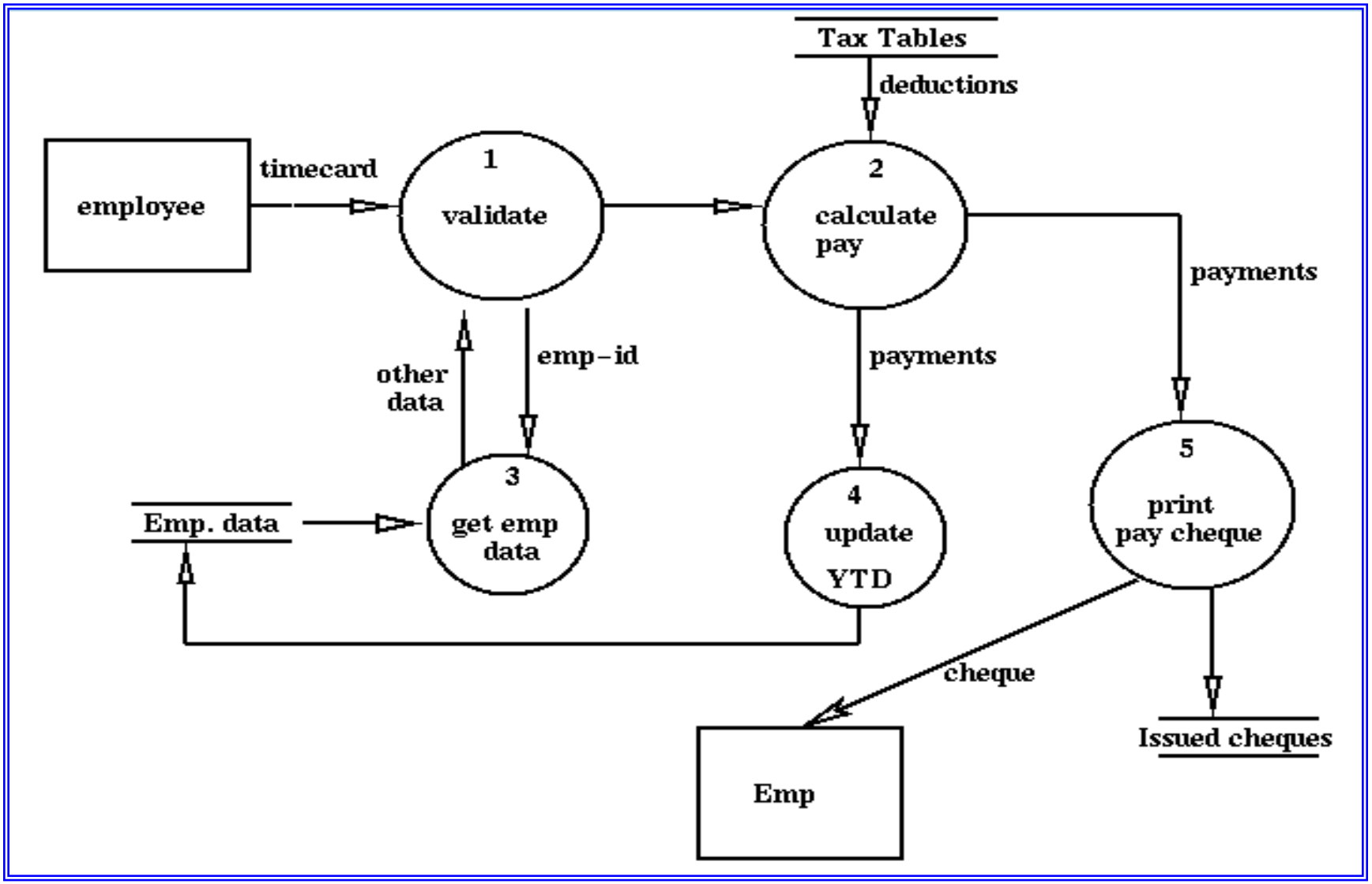
Shortcomings of the DFD Model

- 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.

Example: Air line reservation

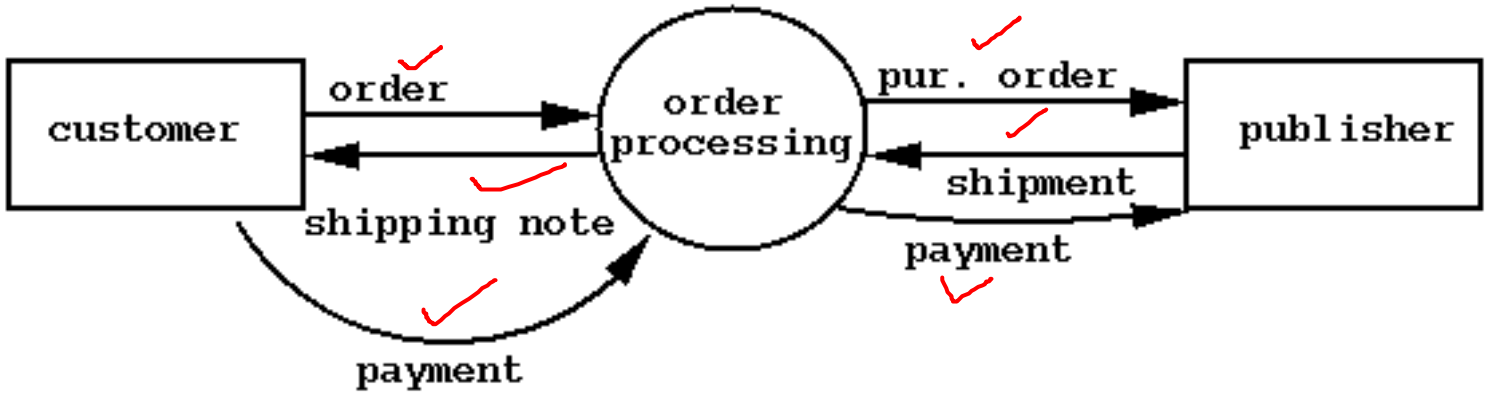


DFD Example : Payroll

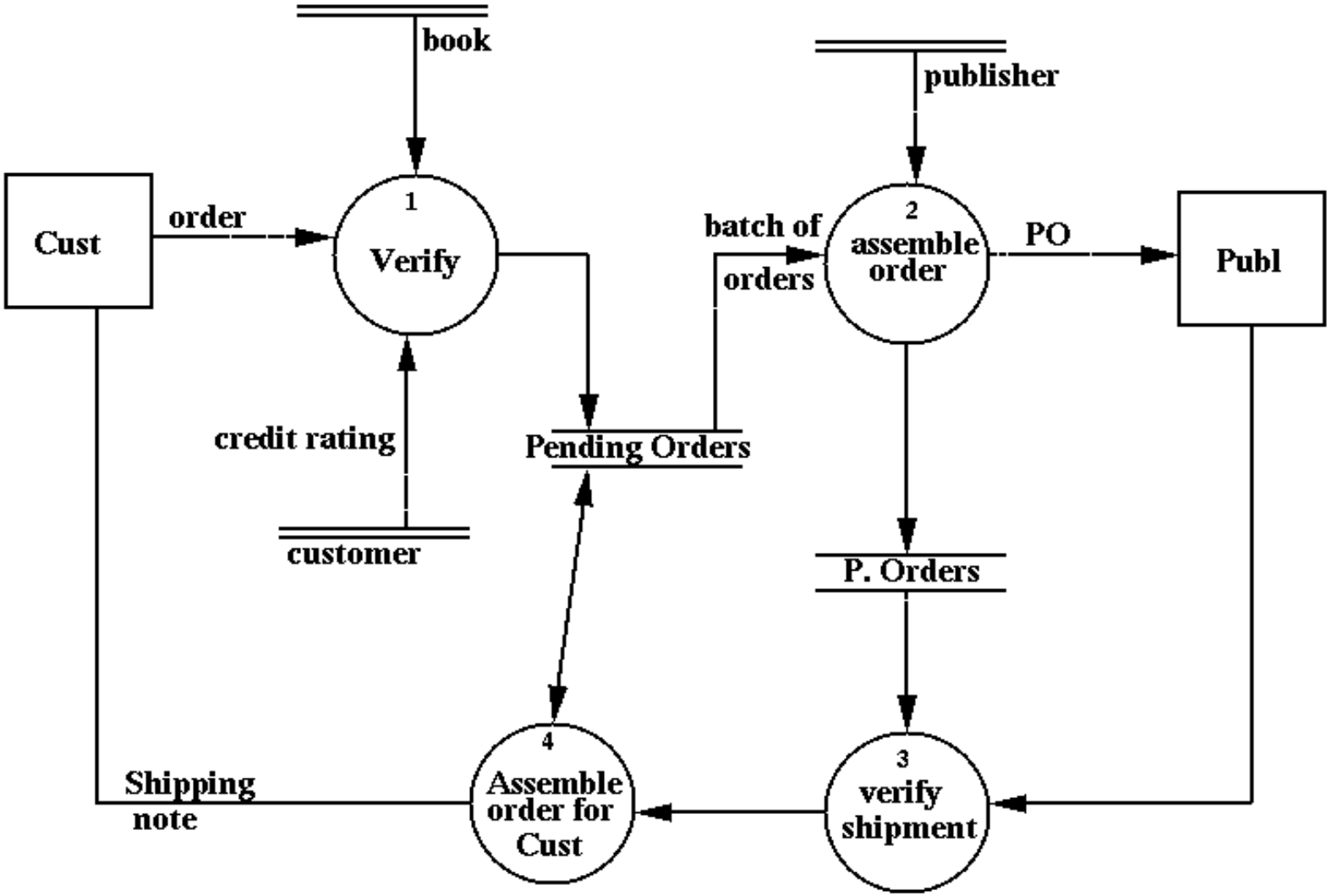


Example: Book Supplier

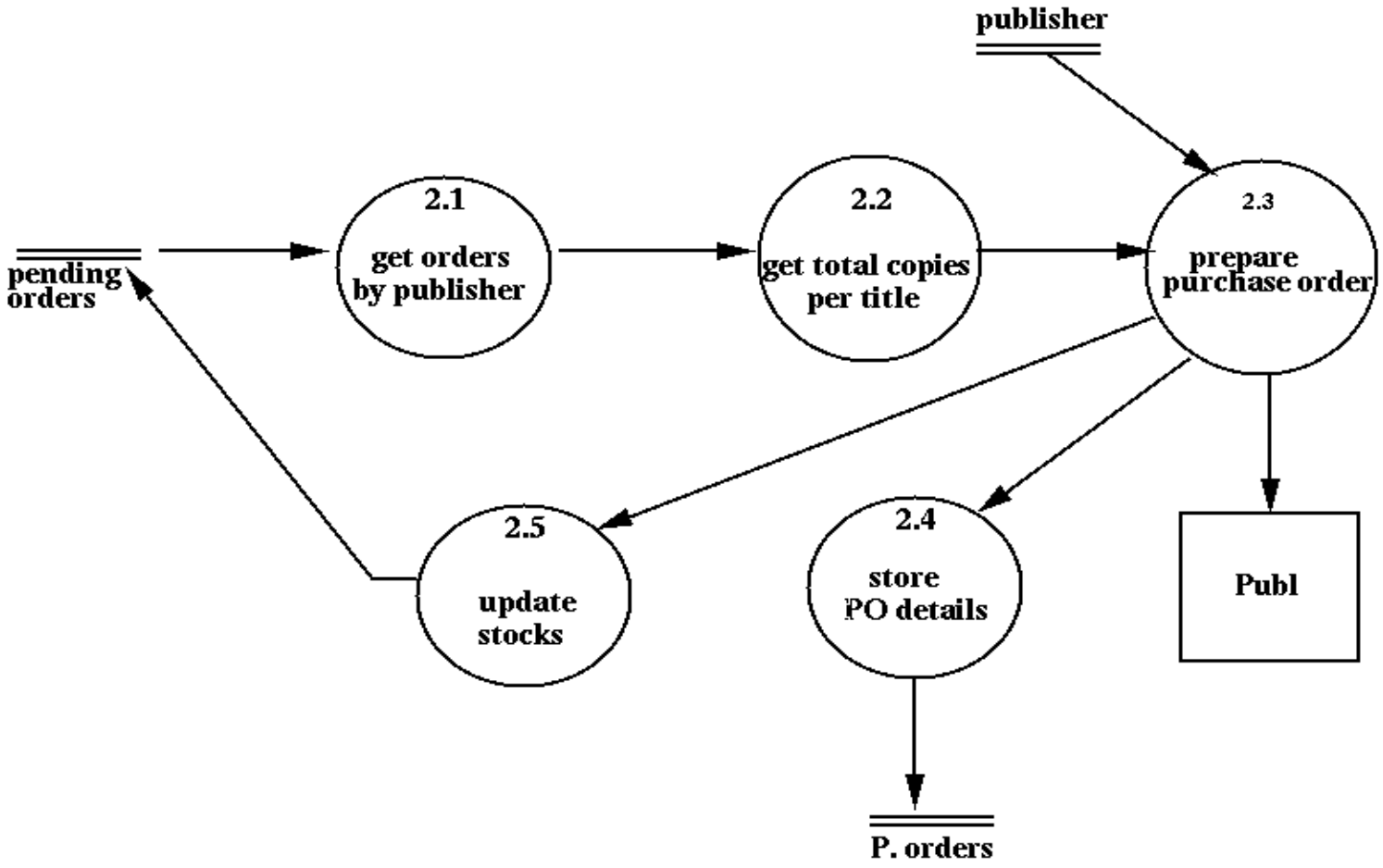
- Supplies books to customers; no stocks maintained; books sourced directly from publishers
- Prepare context diagrams



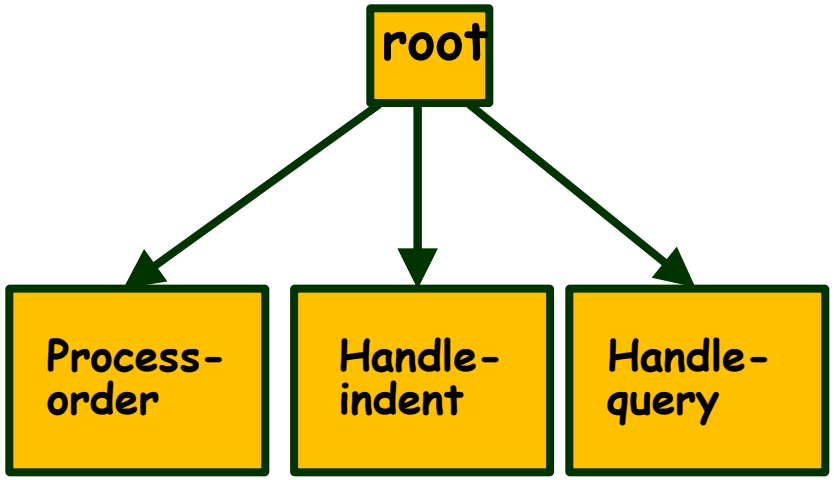
Book-Supplier : Refinement 1



Book Supplier: Exploding Process 2

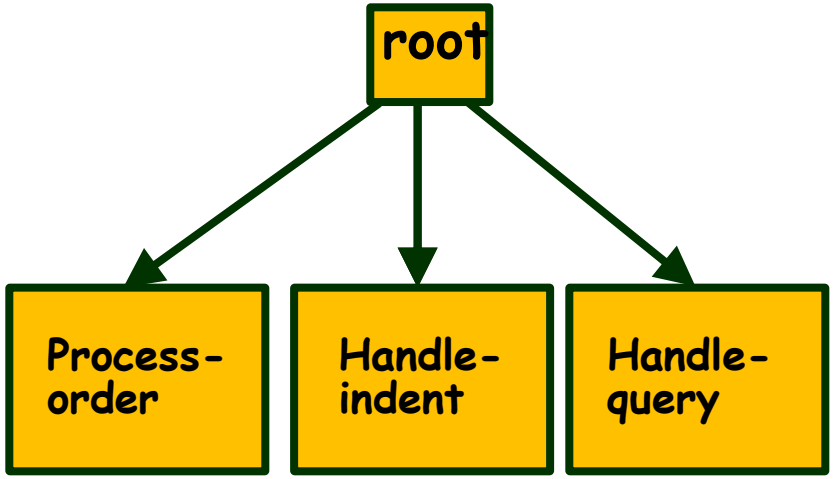


- The aim of structured design
  - **Transform the results of structured analysis (DFD representation) into a structure chart.**



Structure Chart

- Structure chart representation
  - Easily implementable using programming languages.



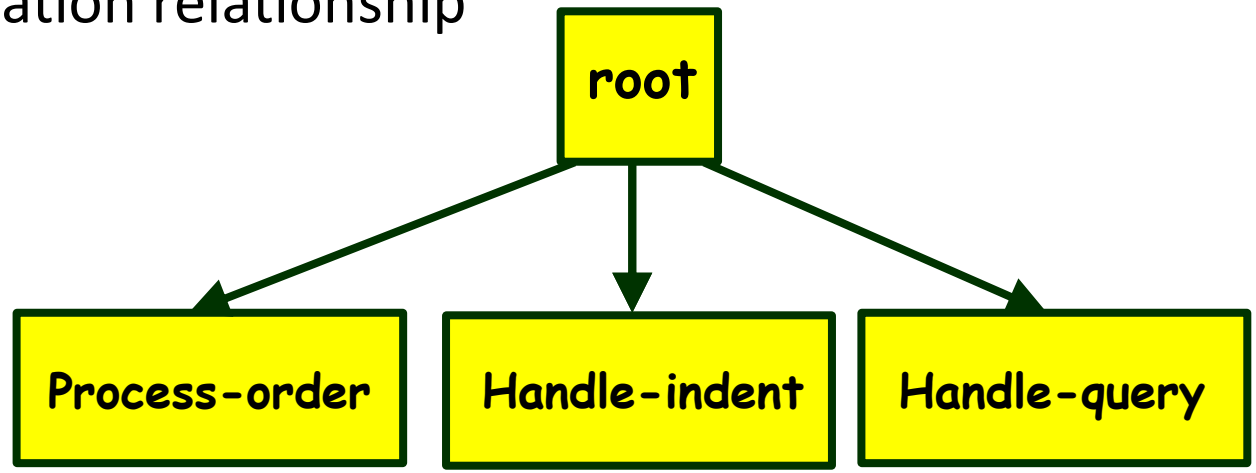
- Main focus of a structure chart:
  - Define the module structure of a software,
  - Interaction among different modules, (call relationship)
  - **Procedural aspects (e.g, how a particular functionality is achieved) are not represented.**

Basic Building Blocks of Structure Chart

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

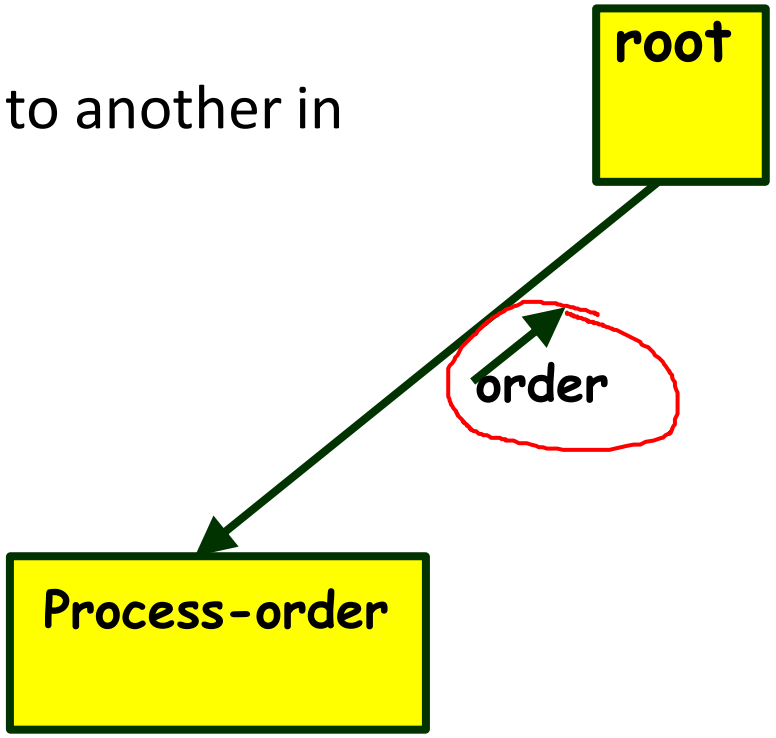
Arrows

- 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



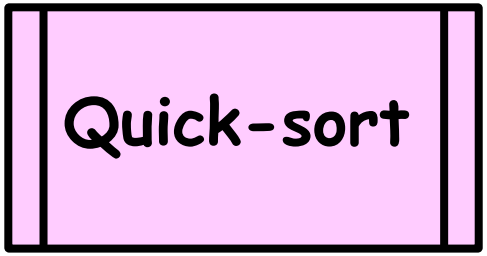
Data Flow Arrows

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



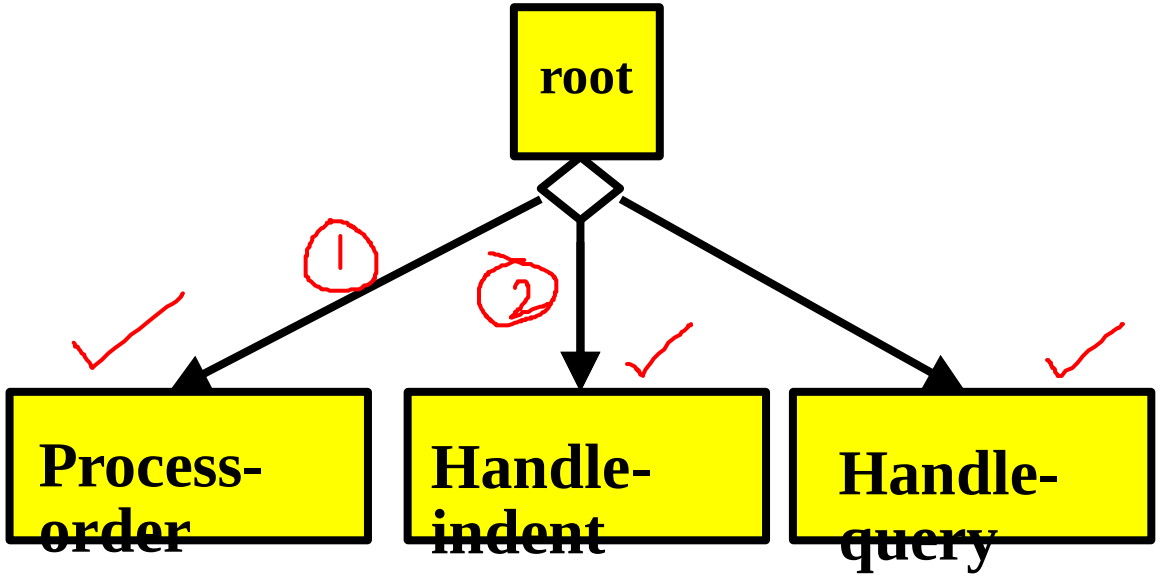
Library Modules

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



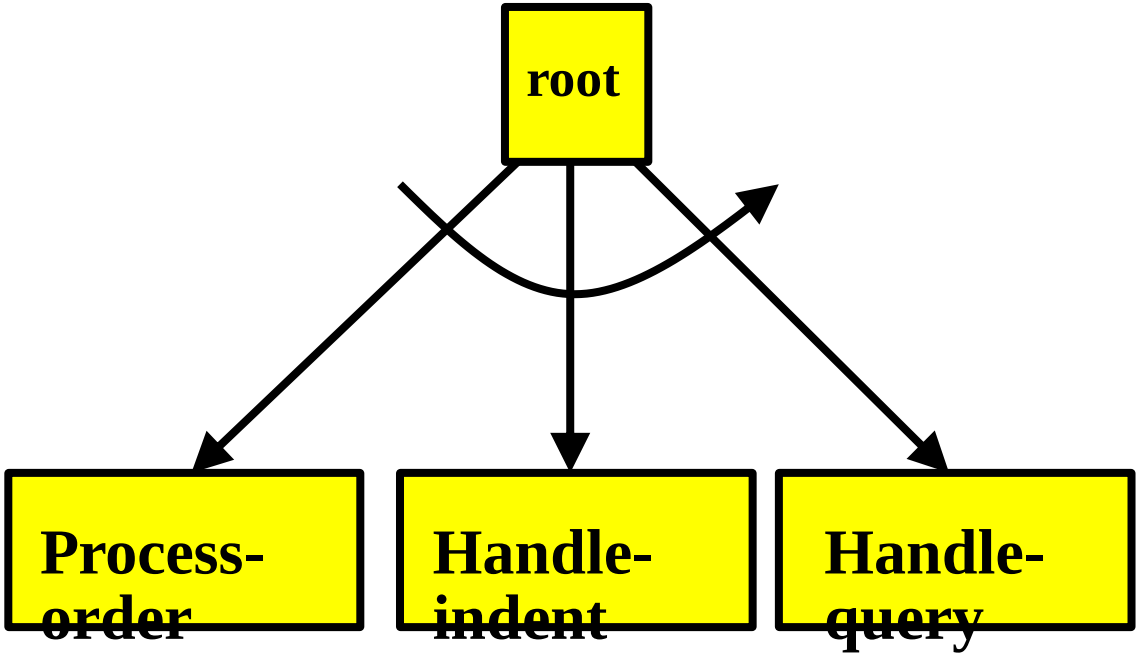
Selection

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

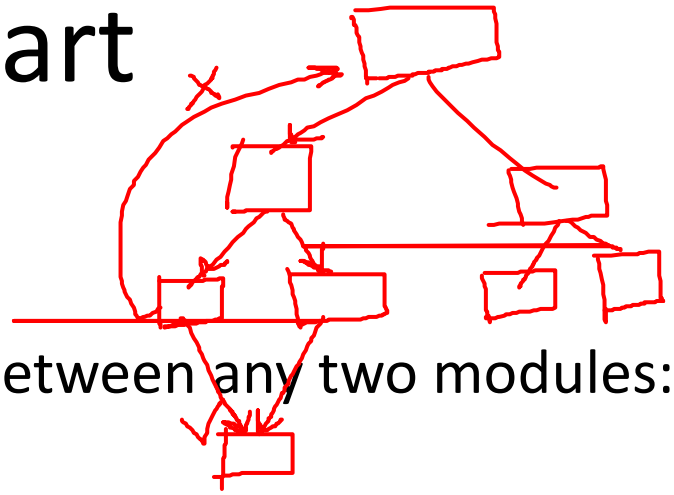


Repetition

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

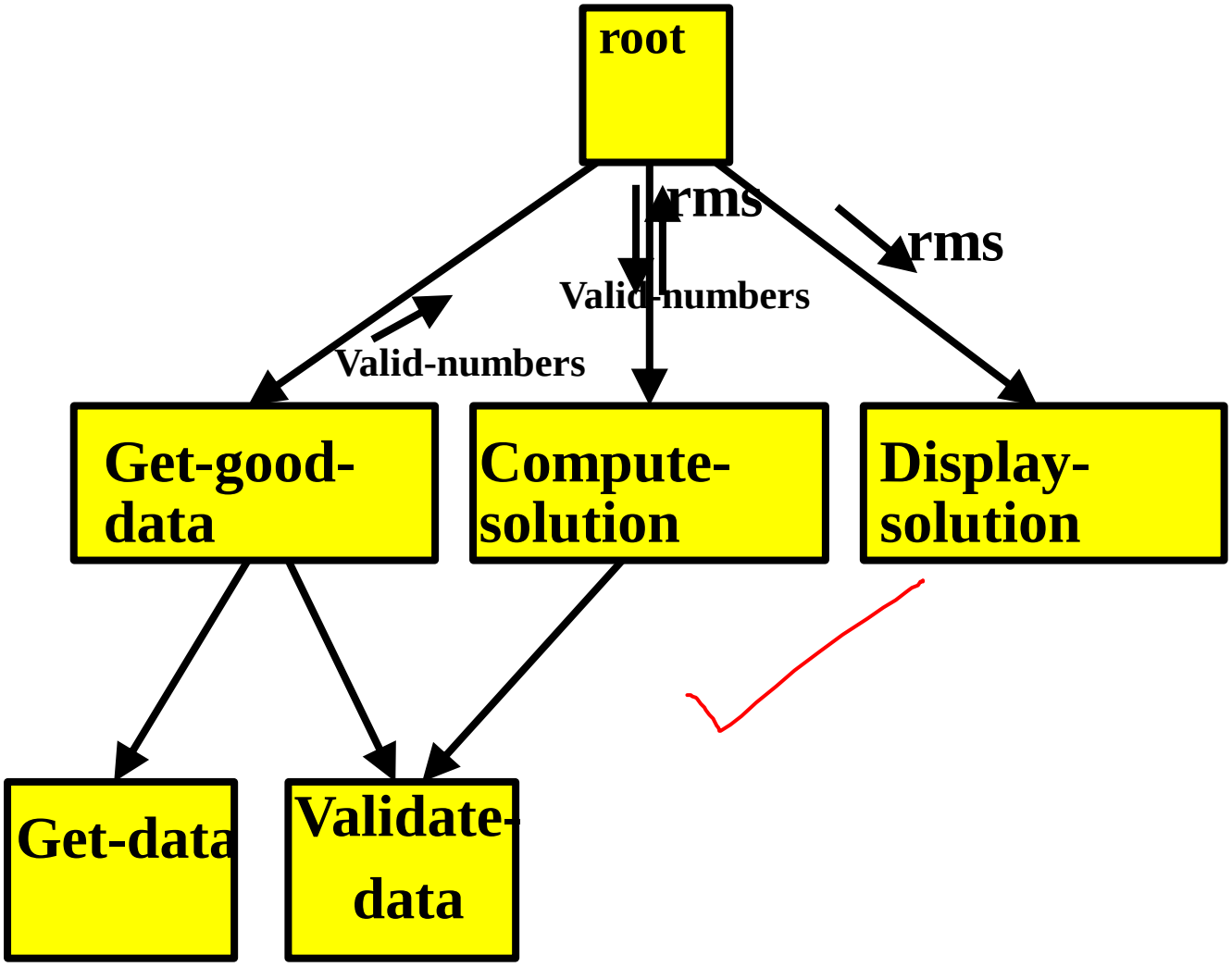


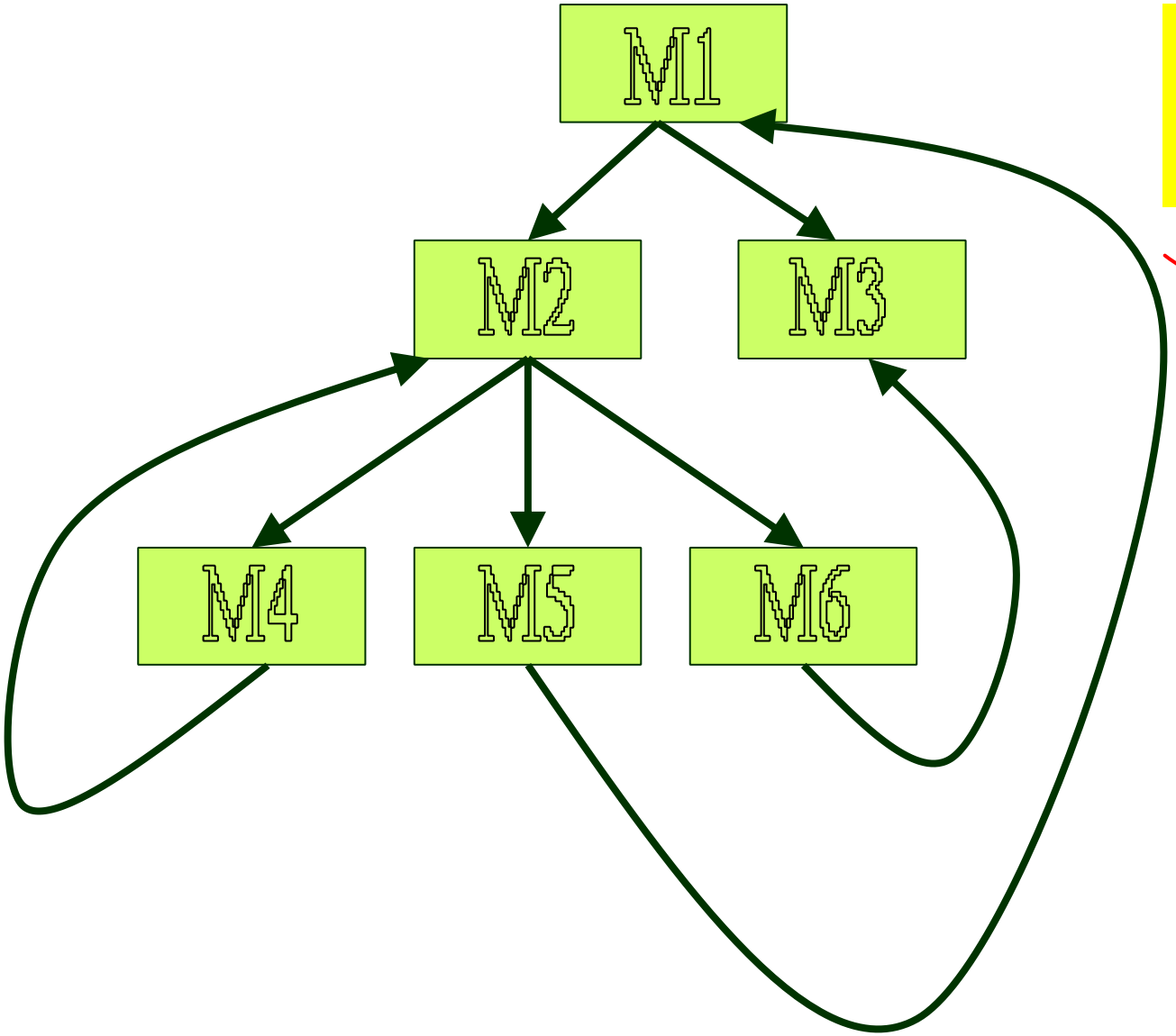
Structure Chart



- 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

References

1. Rajib Mall, “Fundamentals of Software Engineering”, 3rd edition, PHI, 2009
2. R.S. Pressman, “Software Engineering: A Practitioner's Approach”, 7th Edition, McGraw
3. Sommerville, “ Introduction to Software Engineering”, 8th Edition, Addison-Wesley, 2007
4. JAMES RUMBAUGH, IVAR JACOBSON, GRADY BOOCH, “The Unified Modeling Language Reference Manual”, Second Edition, Addison-Wesley, 2004.
5. PPT available for the respective books

Thank You