SOFTWARE ENGINEERING

Module – 5 (FUNCTION-ORIENTED SOFTWARE DESIGN)

CONTENTS

Function-Oriented Design: Overview

🔹 Continued Relevance:

- Proposed **~40 years ago**, yet **still widely used** today.
- Particularly effective for many current software projects.

🔹 Key Idea:

- System is initially viewed as a **black box** that offers **high-level services** (functions) to users.

  Example (Library System):

  issue-book, search-book → considered **high-level functions**

🔹 Design Process:

➢ **Top-down decomposition**:
High-level functions are broken down into detailed sub-functions.

➢ **Module mapping**:
Identified functions are assigned to **modules**, forming a **module structure**.

Function-Oriented Design: Overview (cntd..)

- **Characteristics of a good design**:
  - High **cohesion**
  - Low **coupling**
  - **Layered structure**
  - **Functional independence**

✅ Structured Analysis / Structured Design (SA/SD)

- Instead of focusing on one specific design method, this text describes a **generic function-oriented methodology**, combining essential ideas from the most influential approaches.

🔹 Reason:

- Makes it easier to **adapt to any specific methodology** used in different software development companies.
- Function-oriented design techniques are **closely related (sister techniques)** with only **minor variations in steps and notations**.

Function-Oriented Design: Overview (cntd..)

🔹 **Influential Contributors to SA/SD:**

- **DeMarco & Yourdon** (1978)
- **Constantine & Yourdon** (1979)
- **Gane & Sarson** (1979)
- **Hatley & Pirbhai** (1987)

🔹 **Purpose of SA/SD:**

- Used primarily for **high-level design** of software systems.
- Helps structure the software system into well-defined **functions**, **modules**, and **data flows**.

Feature	Function-Oriented Design
View of System	Black-box offering high-level services
Approach	Top-down decomposition
Outcome	Module structure with good design properties
Method Used	Structured Analysis/Structured Design (SA/SD)
Based On	Techniques from DeMarco, Yourdon, Gane, Sarson, Hatley, Pirbhai
Use	Widely applicable in modern software engineering

5.1 Overview of SA/SD Methodology

SA/SD stands for:

- **Structured Analysis (SA)**
- **Structured Design (SD)**

- These are **two distinct but connected phases** in the **function-oriented design methodology**, forming a **systematic top-down approach** to software development.

The roles of structured analysis (SA) and structured design (SD) have been shown schematically **in Figure 5.1**. Observe the following from the figure

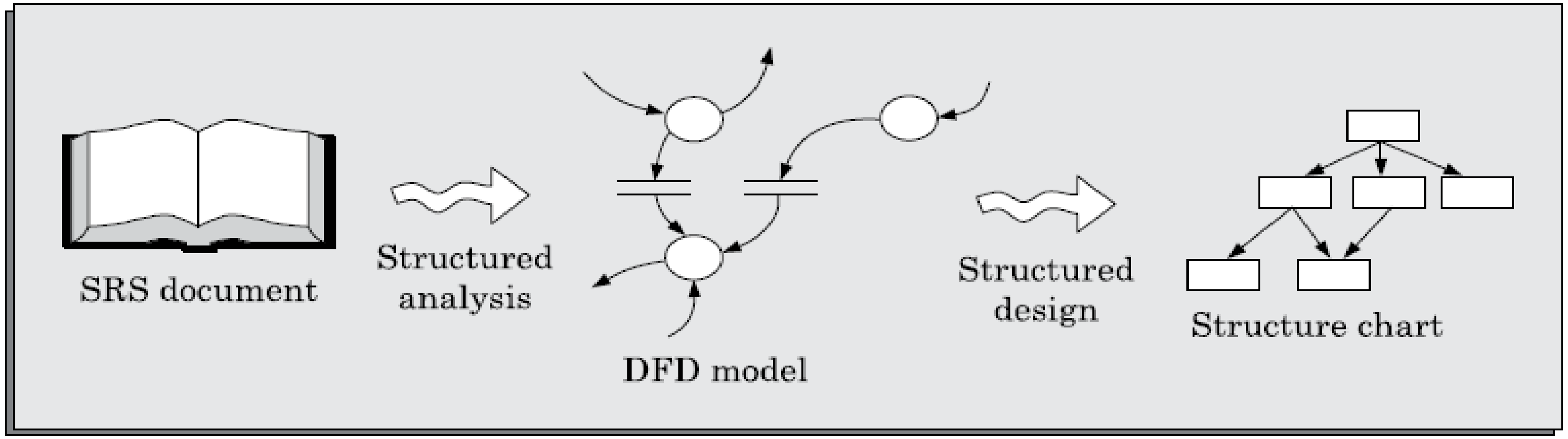


FIGURE 5.1 Structured analysis and structured design methodology.

🔄 Process Flow:

Structured Analysis and Structured Design follow a **step-by-step transformation**:

1. Structured Analysis (SA):

- **Input**: Software Requirements Specification (SRS) document
- **Output**: **Data Flow Diagram (DFD)** model
- **Purpose**: To analyze and decompose the system's required **functions** hierarchically into **sub functions**

- **Features**:
  - Uses **user-friendly terms** for functions and data
  - **Easily understandable** and reviewable by end-users

2. Structured Design (SD):

- **Input**: DFD model from SA
- **Output**: **Structure chart** (high-level design or software architecture)
- **Purpose**: To **map each function** from DFD to **software modules**
- **Result**: A hierarchical module structure ready for implementation

🧱 Next Step:

- After high-level design (structure chart):
- Perform **detailed design** of each module:
  - Design **algorithms** and **data structures**
  - This stage directly leads to **implementation** in a programming language

Concept	Description
Top-down decomposition	Gradual breaking down of high-level functions into more detailed ones
DFD (Data Flow Diagram)	Graphical model showing how data flows and is processed by the system
Structure chart	A tree-like diagram showing how modules are organized and interact
SA/SD Methodology	A stepwise approach from user requirements → graphical model → modular design

Phase	Activity	Input	Output
Structured Analysis (SA)	**Functional decomposition & modeling**	**SRS document**	**DFD model**
Structured Design (SD)	**Modular design**	**DFD model**	**Structure chart (high-level design)**
Detailed Design	**Design of algorithms & data**	**Structure chart**	**Code-level design for implementation**

5.2 Structured Analysis

Structured Analysis is the **first phase** of the SA/SD methodology. It involves identifying and representing the **major processing tasks (high-level functions)** and **data flow** in a system using a graphical model known as a **Data Flow Diagram (DFD)**.

🔑 Key Principles of Structured Analysis

- ✅ **Top-down decomposition**: Break high-level functions into more detailed ones.
- ✅ **Divide and conquer**: Analyze each function independently.
- ✅ **Graphical representation**: Use **DFDs** to visualize the flow of data and processes.

❖ **Note: DFDs focus only on data flow, not control flow (e.g., sequence of execution or conditional logic).**

5.2.1 Data Flow Diagrams (DFDs)

🧠 What is a DFD?

- A DFD is a **hierarchical graphical model** that shows:
- Inputs and outputs of the system
- Processing functions (called **processes or bubbles**)
- Data stores
- External entities

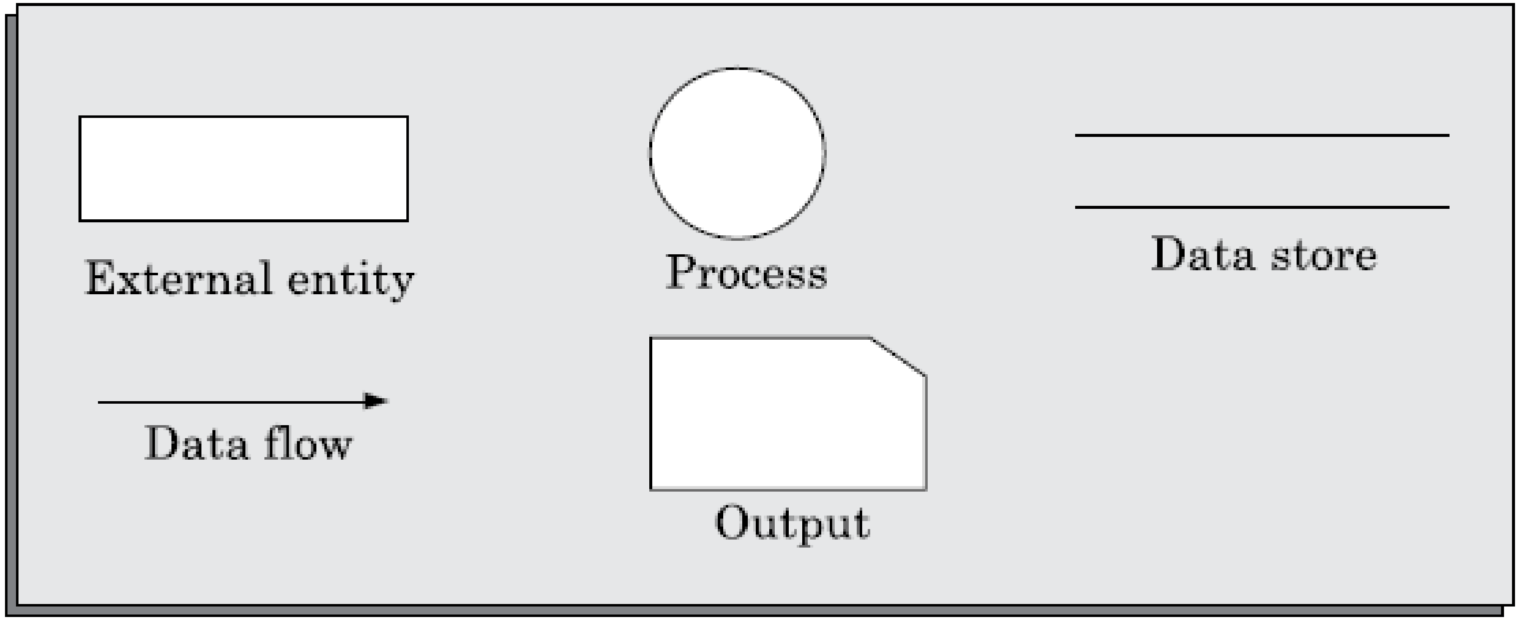


FIGURE 5.2 Symbols used for designing DFDs.

DFD Symbols and Their Meanings

Symbol	**Meaning**	**Description**
○ (Circle)	Process/Bubble	Represents a function (e.g., "Validate Input")
▭ (Rectangle)	External Entity	A user, hardware, or external software (e.g., "Librarian")
→ (Arrow)	Data Flow	Represents data movement between entities, processes, and stores
I| (Two parallel lines)	Data Store	Logical file or database (e.g., "Book Records")
📄	Output Symbol	Represents physical output (e.g., printed report)

Synchronous vs. Asynchronous Processing

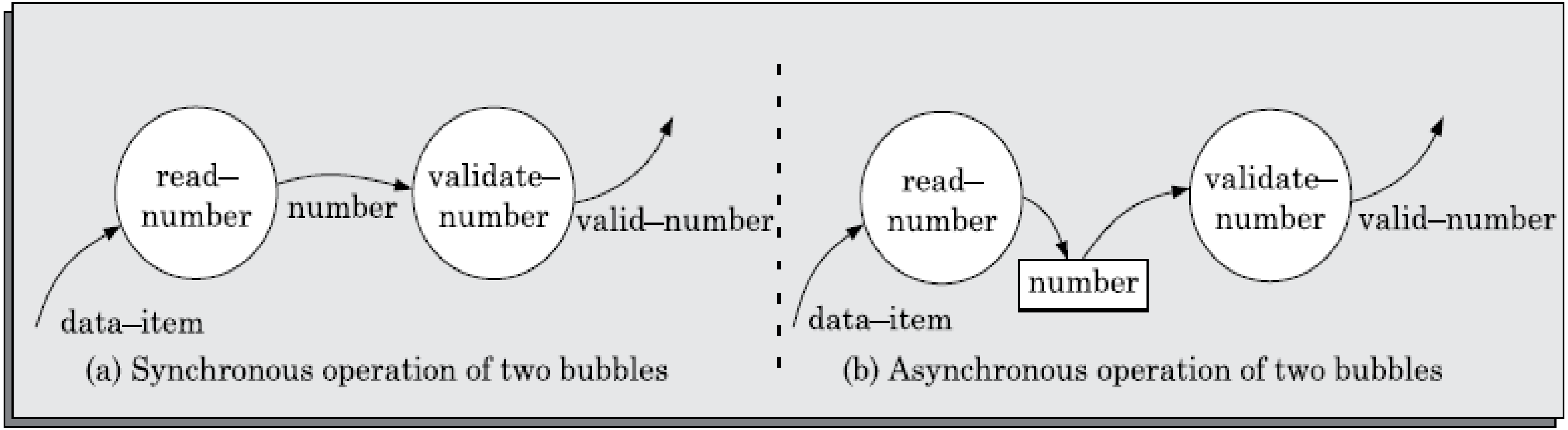


FIGURE 5.3 Synchronous and asynchronous data flow.

Type	Description
Synchronous	Two processes connected directly by a data flow arrow. They operate at the same speed.
Asynchronous	Two processes connected via a data store. Their operations are independent.

📌 **Example: A process writes to a file, and another reads it later — they're asynchronous.**

Data Dictionary

A **data dictionary** is a companion to the DFD and includes:

- A **list of all data items** (flows and stores)
- Definitions of **composite** and **primitive** data
- Purpose and usage of data items

🔁 Shared terminology ensures **consistency**, avoids confusion, and aids in **impact analysis** and **maintenance**.

🧰 Why Data Dictionary is Important

- Provides **standard definitions** across developers
- Helps design **data structures**
- Useful for **impact analysis** (e.g., what is affected if data changes)
- Is **auto-generated** by most CASE tools

🔡 Data Definition Operators

Operator	Meaning	Example
+	Composition	grossPay = basic + bonus
[a,b]	Selection (either/or)	Either a or b occurs
( )	Optional	a + (b) means a or a+b
{}	Iteration	{name}5 = 5 name values; {name}* = 0 or more names
=	Equivalence	x = y + z means x includes y and z
/* */	Comment	/* optional middle name */

Concept	Summary
Structured Analysis	Decomposes high-level functions into detailed ones
DFD	Shows system processing and data movement
Symbols	Process (○), Data Store (||), Data Flow (→), External Entity (▭)
Data Dictionary	Defines all data used in DFDs; helps in design and consistency
Operators	Define composite, optional, and iterative data relationships

5.3 Developing the DFD Model of a system

A **Data Flow Diagram (DFD) model** represents **how input data is transformed into output data** through a **hierarchy of diagrams**.

- The DFD model uses **multiple levels** to show increasing detail.
- It is developed **top-down**, starting with the most abstract level (Level 0) and gradually adding detail in lower levels.

Levels of DFD Hierarchy

- **Level 0 DFD** (also called the **Context Diagram**):
    - Represents the **entire system as a single bubble**.
    - It's the **most abstract** and **easiest to draw and understand**.
    - Shows only the **external entities**, the **data they send to the system**, and the **data they receive** from it.

Level 1 DFD:

- Decomposes the single process (bubble) of Level 0 into **sub-processes**.
- Shows **major internal processes**, **data stores**, and **data flows**.

Level 2 and below:

- Each process in Level 1 can be further decomposed into **more detailed sub-processes**.
- Up to:
  - **7 DFDs in Level 2**
  - **49 DFDs in Level 3**
  - And so on (based on 7±2 rule of cognitive load)

- **Note: Even though there are many DFDs at different levels, there is only one Data Dictionary for the entire DFD model.**

- **It defines all data items used across all levels of DFDs.**

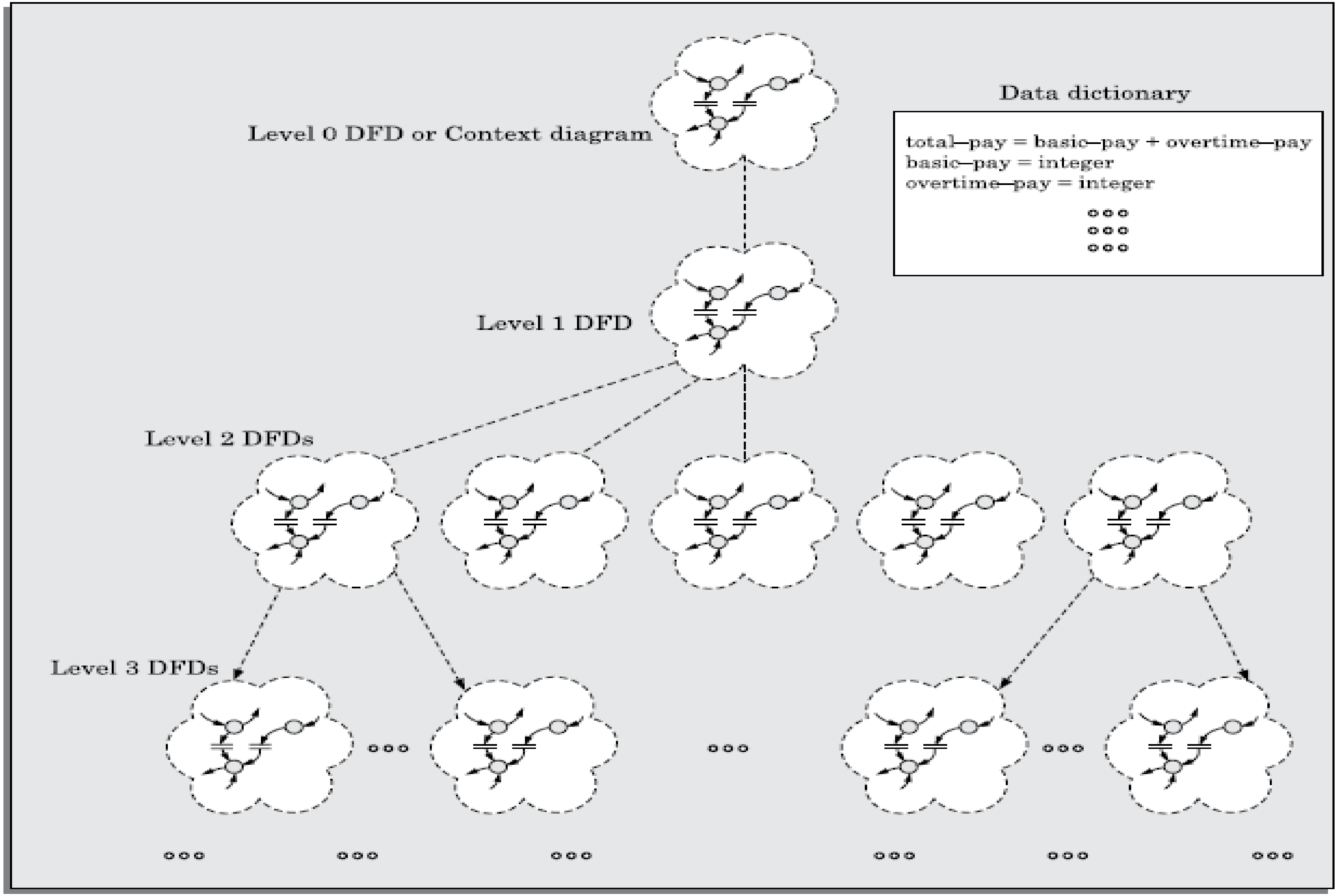


FIGURE 5.4 DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

5.3.1 Context Diagram (Level 0 DFD)

- It shows the **complete system as one process** (bubble) and is **named using a noun** (e.g., “Library System” or “Supermarket Software”).
- **Only in this diagram** is a noun used to name the bubble; in lower levels, **verbs** are used to describe functions.
- **Key features:**
- Captures the **context** in which the system operates.
- Shows:
  - **External entities** (users or other systems)
  - **Input data** to the system (incoming arrows)
  - **Output data** from the system (outgoing arrows)
- Helps identify **who interacts** with the system, **what data** they provide, and **what data** they receive.

To create a context diagram:

- **Read the SRS (Software Requirements Specification)** to:
  - Identify all **types of users**
  - Determine the **data they input**
  - Determine the **data they expect as output**
  - Include **external systems** as entities if they interact with the system

- A **DFD model** includes **multiple hierarchical diagrams** starting from the **context diagram**.
- Each level **adds more detail** by decomposing processes from the previous level.
- A **single data dictionary** defines all data used across all DFD levels.
- The **context diagram** is the top-level view that shows **who uses the system and how they interact** with it.

5.3.2 Level 1 DFD

A **Level 1 DFD** is a more detailed version of the **context diagram (Level 0 DFD)**. It shows how the main system process (from the context diagram) can be **decomposed into 3–7 sub-processes** (bubbles), each representing a major function of the system.

Key Points of Level 1 DFD

- **Ideal Number of Bubbles**:
  - Typically includes **3 to 7 bubbles**, each for a high-level function.
  - Based on **SRS (Software Requirements Specification)** document:
    - If exactly 3–7 major functions exist → each becomes a bubble.
    - If **>7 functions** → combine related ones into broader bubbles.
    - If **<3 functions** → split them into subfunctions to maintain clarity.

Input/output Analysis:

- Identify:
  - Input data to each function
  - Output data from each function
  - Interactions (data flow) among the functions
- All must be documented and shown clearly.

Decomposition (Factoring or Exploding)

- Each bubble (function) in Level 1 can be further **decomposed into sub functions**.
- Ideal decomposition also follows the **3–7 bubble rule**.
- A bubble should be decomposed **until its task can be implemented using a simple algorithm**.
- Avoid:
  - **Too few bubbles**: Makes levels redundant.
  - **Too many bubbles**: Hard to understand.

Steps to Develop the DFD Model

Context Diagram:

- From the SRS:
  - Identify high-level functions.
  - Find their input/output data.
  - Identify interactions.
- Represent the system as a **single bubble**, interacting with external entities.

Level 1 DFD:

- Create **3–7 bubbles**, each showing a high-level function.
- Combine/split functions if needed to follow the 3–7 rule.

Lower-Level DFDs (Level 2, 3...):

- Decompose each bubble from the previous level:
  - Identify sub functions.
  - Show input/output data for each.
  - Show interactions between sub functions.
- Repeat until all functions are **simple enough to code directly**.

Bubble Numbering

- Helps in identifying and referencing bubbles.
- Numbering follows hierarchy:
  - Context Diagram: 0
  - Level 1 bubbles: 0.1, 0.2, ...
  - If 0.1 is decomposed: 0.1.1, 0.1.2, ...
- By reading the number, you can tell a bubble's level and parent.

Balancing DFDs

- **Balanced DFD**: Data flow **into/out of a bubble at one level** must **match** the data flow **into/out of the same bubble in its decomposed level**.
- Example:
  - If 0.1 has d1, d2, d3 as input/output, the DFD showing 0.1.1, 0.1.2 etc., must also reflect d1, d2, d3.

How Far to Decompose?

- Stop decomposing when:
  - The function can be **described using a simple algorithm**.
- Simple systems: Level 1 is usually enough.
- Complex systems: May need Level 2, 3, or 4.
- Rarely required: Beyond Level 4.

Common Errors in DFD Construction

Avoid these frequent mistakes:

1. **Multiple bubbles in context diagram** (only one allowed).
2. **External entities shown in lower levels** (should only be in context diagram).
3. **Too few or too many bubbles per level** (stick to 3–7).
4. **Unbalanced DFDs** (mismatched input/output between levels).

5. Trying to show control logic or sequencing, e.g.:

- Arrows for "if-else" decisions.
- Representing execution order.
- Showing invocation conditions.

  (DFDs show **data flow only**, not control flow).

6. Connecting data stores directly to each other or to external entities (data must flow through processes).

7. Omitting system functions described in the SRS.

8. Including functionality not mentioned in the SRS.

9. Incomplete or incorrect data dictionary.

10. Using non-intuitive names (like a, b, c) for data/functions.

11. Data flow clutter: Too many arrows going in/out of a bubble.

Solution: Combine multiple data into a **single high-level data item**.

Illustrative Example (Example 5.1: RMS Calculator)

Problem:

- Input: Three integers (–1000 to +1000)
- Output: RMS (Root Mean Square) of the numbers

Context Diagram:

- A single bubble: RMS Calculator
- Data:
  - Input from user: three integers
  - Output to user: rms result

Level 1 DFD:

- Four main functions:
  1. Accept numbers
  2. Validate numbers
  3. Calculate RMS
  4. Display result

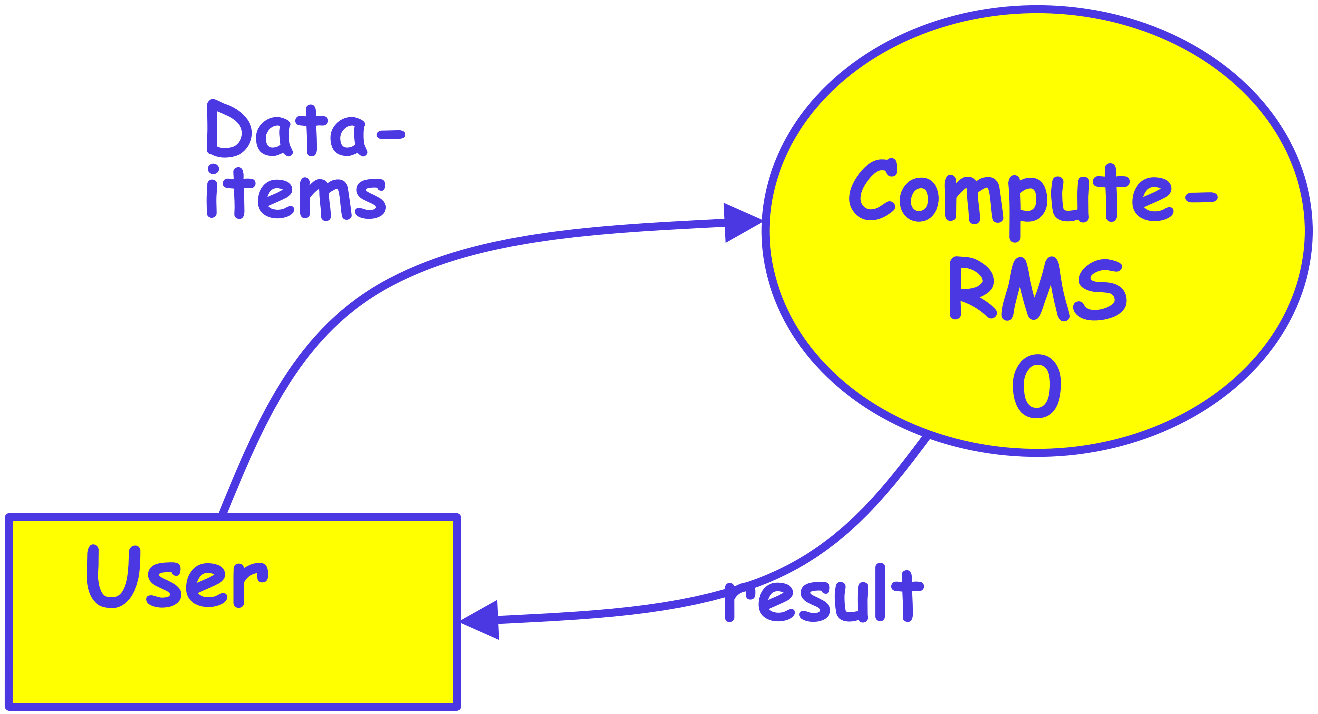
Level 2 DFD (for "Calculate RMS"):

- Decomposed into:
  1. Square inputs
  2. Compute mean
  3. Compute root

Data Dictionary for Example

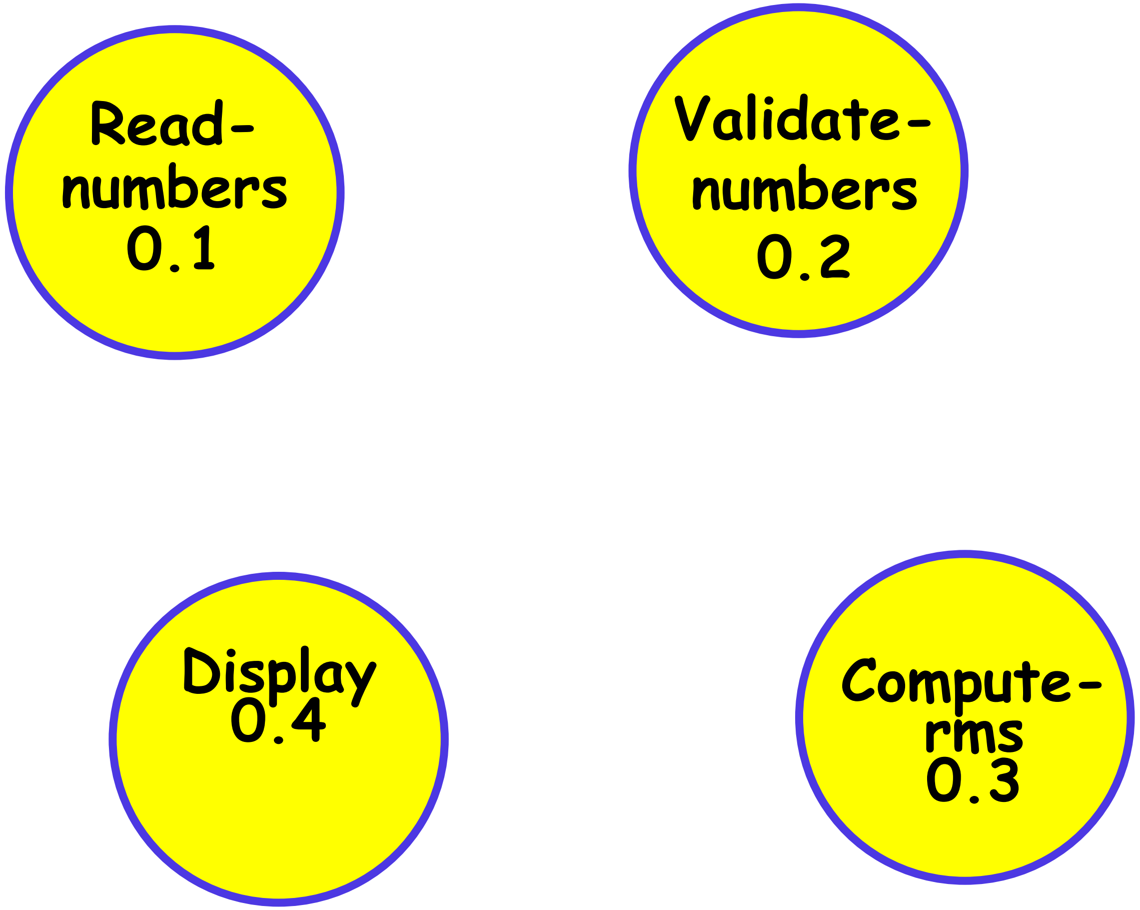
Data Item	Description
data-items	3 integers
rms	Floating-point RMS value
valid-data	Same as data-items if valid
a, b, c	Individual integers
asq, bsq, csq	Squares of inputs
msq	Mean of squares

Example 1: RMS Calculating Software

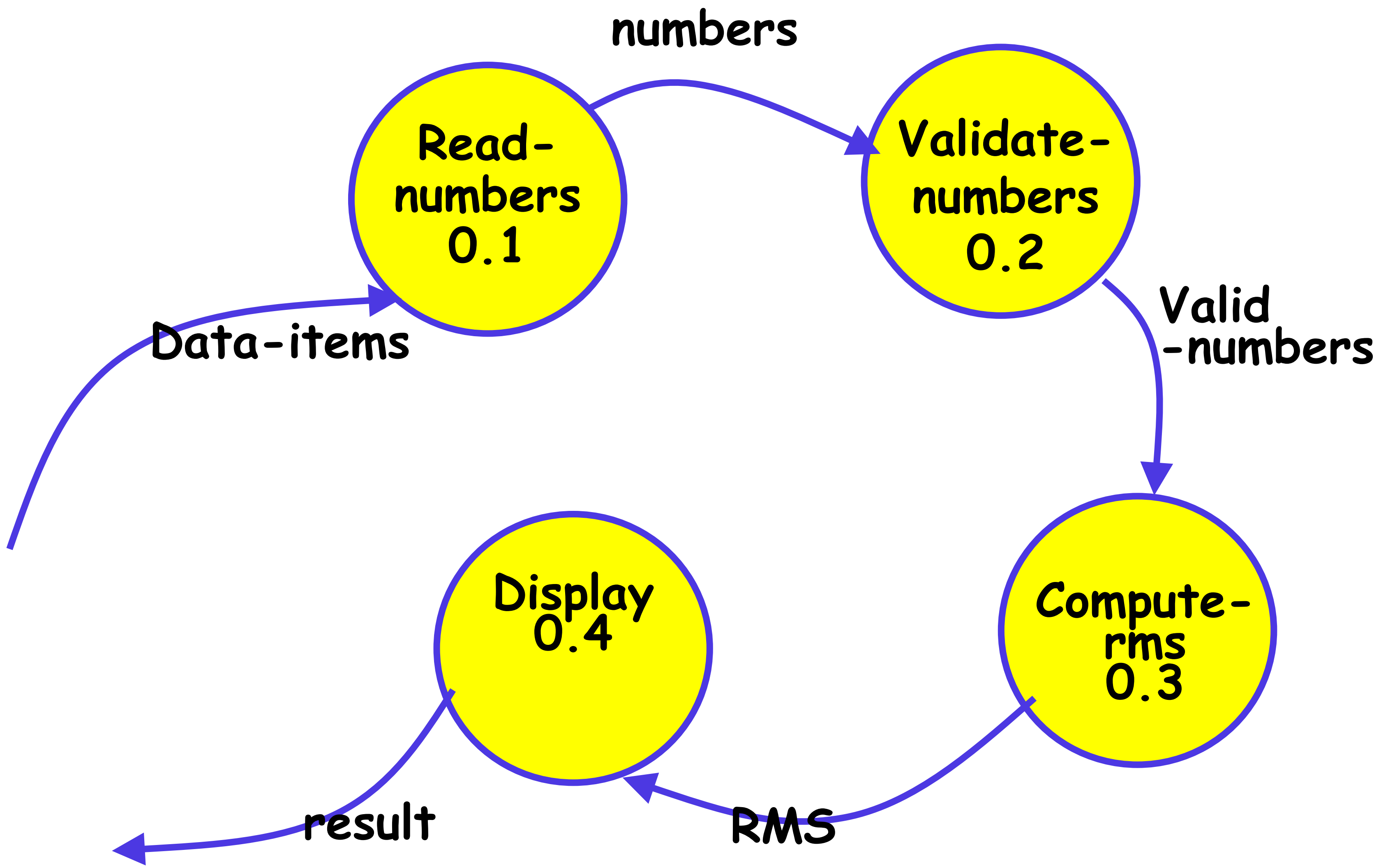


Context Diagram

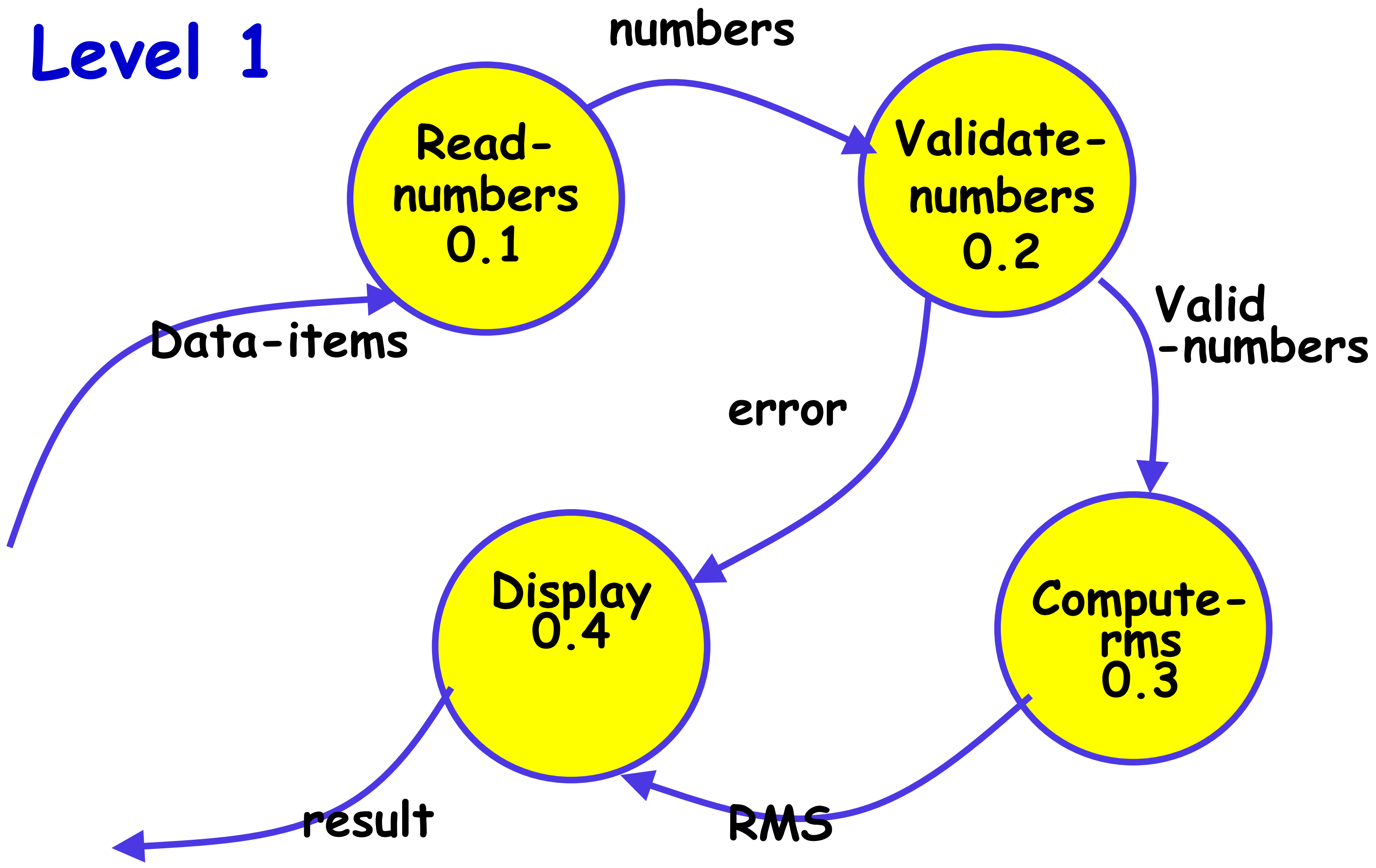
Example 1: RMS Calculating Software



Example 1: RMS Calculating Software

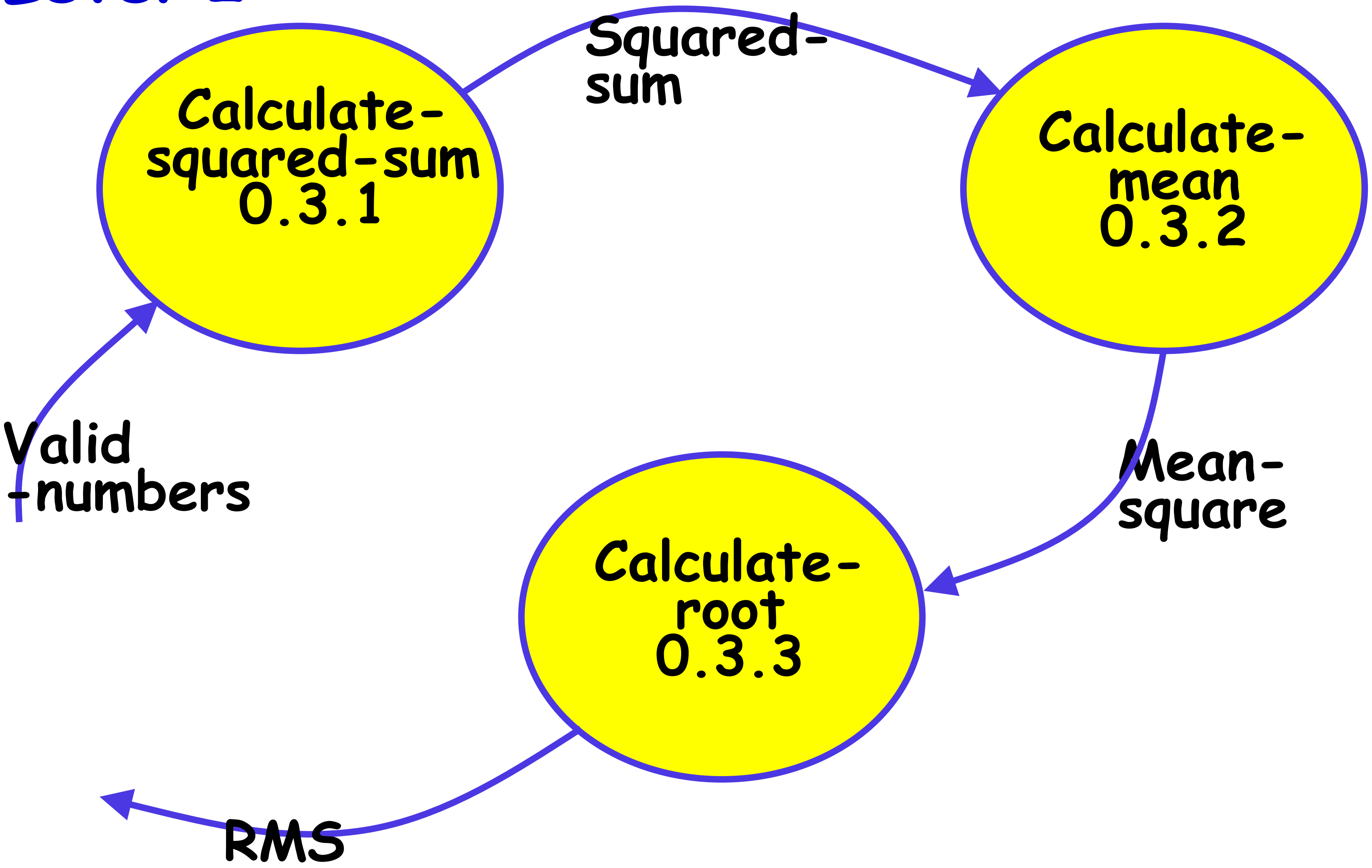


Example 1: RMS Calculating Software

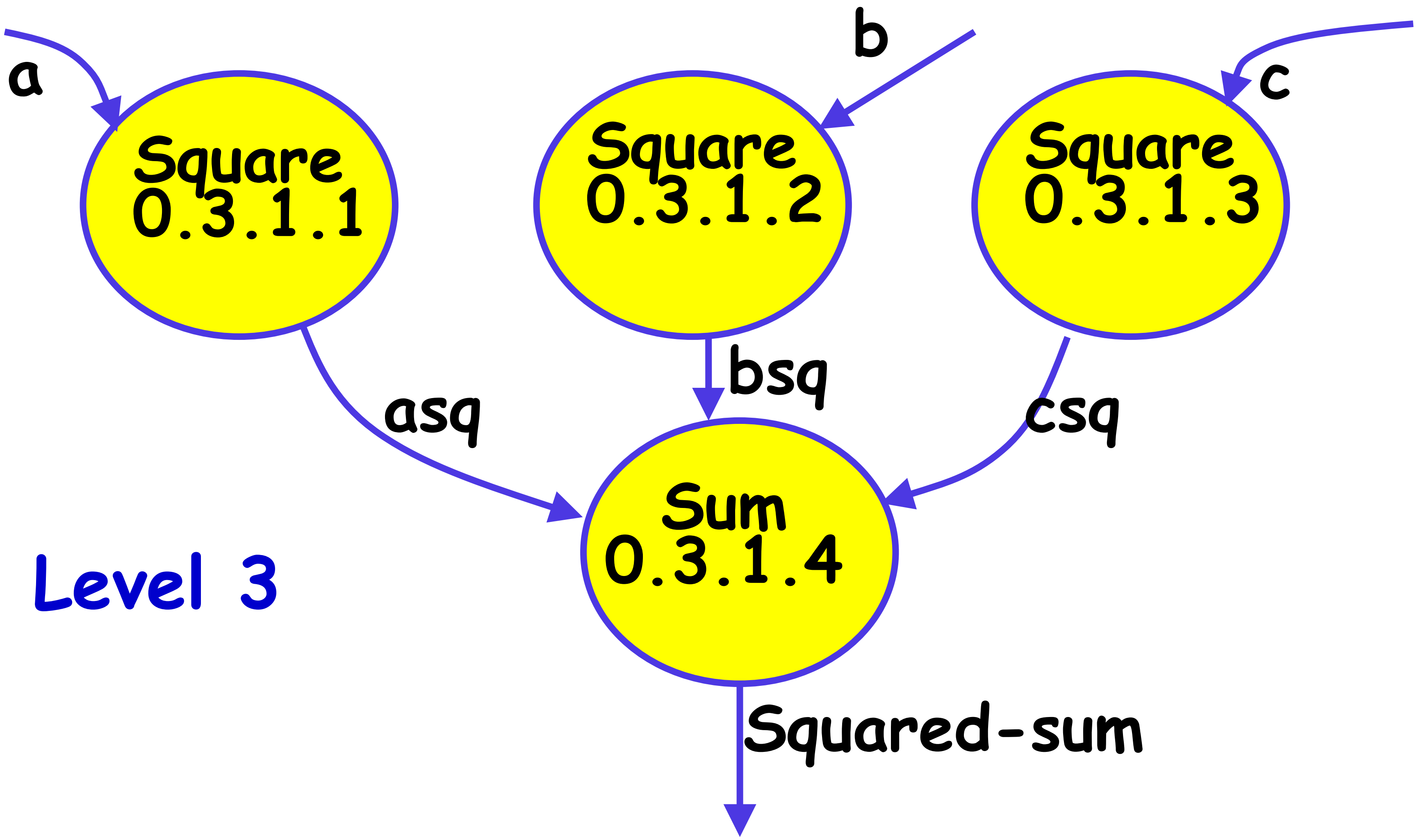


Example 1: RMS Calculating Software

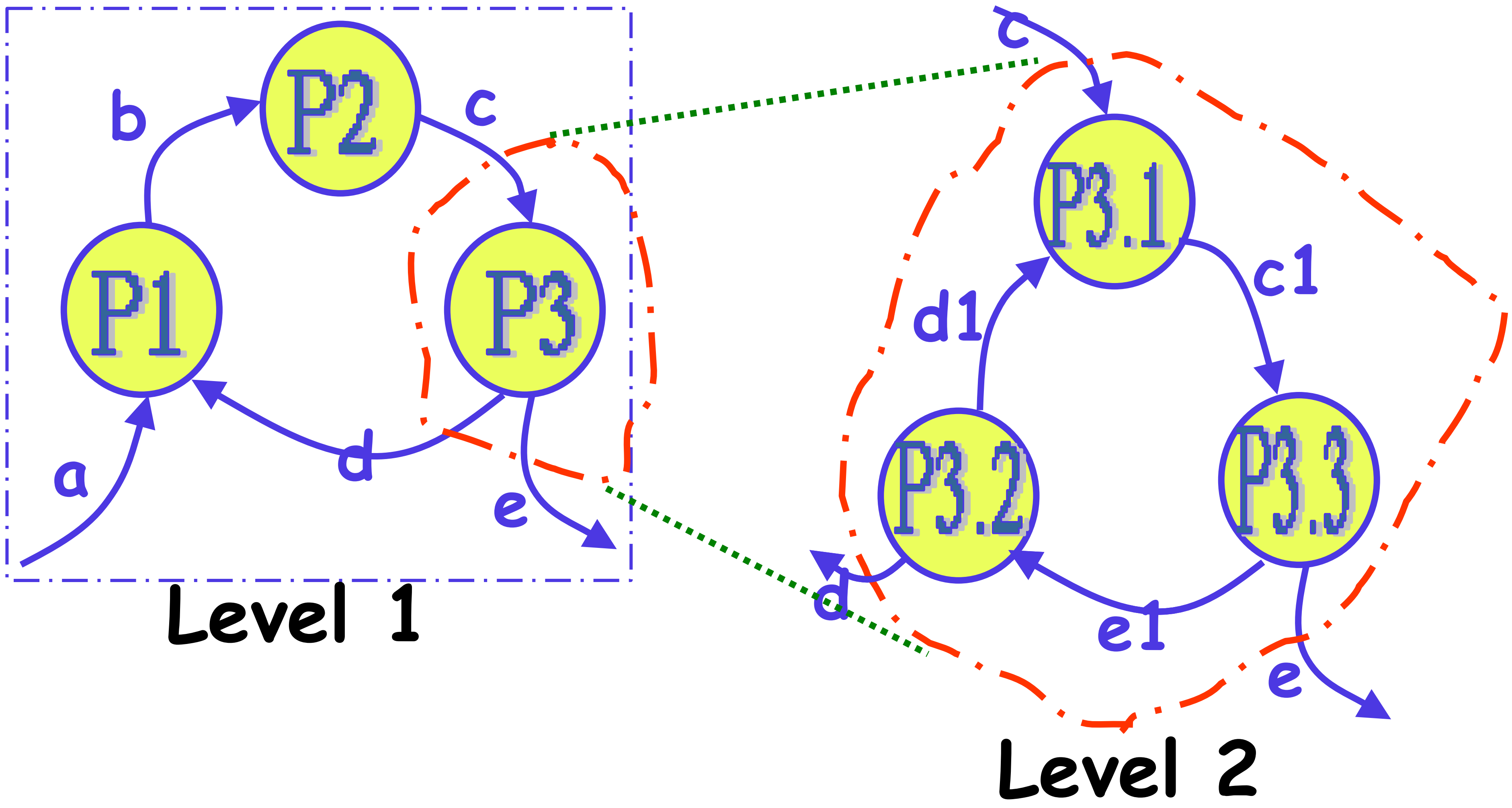
Level 2



Example: RMS Calculating Software



Balancing a DFD



5.3.3 Extending DFD Technique to Real-Time Systems

✅ Why Extension is Needed:

- **Real-time systems** are different from regular systems because:
  - They must **produce correct results**.
  - They must do so **within strict time deadlines**.
- Therefore, in real-time systems, **timing and control flows** are **critical** in the design.

✅ Problem with Traditional DFD:

- DFDs focus only on **data flow** and **functional decomposition**.
- They **do not represent**:
  - **Control flow**
  - **Event handling**
  - **Time constraints**
- Hence, DFDs need to be **extended** to suit real-time systems.

✅ Solution: Extensions by Ward & Mellor and Hatley & Pirbhai

🔹 Ward and Mellor Technique (1985):

- Adds **new symbols** to DFDs:
  - **Dashed bubbles** → represent **control processes**.
  - **Dashed arrows/lines** → represent **control flows** (like events or triggers).
- This allows both **data processing** and **control processing** to be shown in **the same diagram**.

🔹 Hatley and Pirbhai Technique (1987):

- They further simplify the model by **separating**:
  - **Data processing** (shown in traditional DFD).
  - **Control processing** (shown in a new diagram called **CFD** – Control Flow Diagram).
- They use a **solid vertical bar (notational reference)** to **link** the two diagrams.

✅ New Components Introduced:

📌 CFD (Control Flow Diagram):

- Represents **control-related processing**.
- Makes diagrams **less complex** by separating control and data.

📌 CSPEC (Control Specification):

- Linked to the CFD.
- Describes:
  - How the system **reacts to external events/control signals**.
  - **Which processes are invoked** when an event occurs.

✅ CSPEC Includes Two Parts:

STD (State Transition Diagram):

- Shows how the system **changes state** in response to events.
- Describes system behavior **sequentially**.

PAT (Program Activation Table):

- Describes **which processes** are **activated under what conditions**.
- Describes behavior **combinatorially**.
- Helps understand **which bubbles in DFD** are triggered by which events.

Concept	Description
Ward & Mellor	Use dashed bubbles/arrows in DFD to show control
Hatley & Pirbhai	Separate data flow (DFD) and control flow (CFD)
CSPEC	Links control diagram to behavior logic
STD	Sequential behavior (state changes)
PAT	Combinatorial behavior (which function activates when)

<u>5.4 STRUCTURED DESIGN</u>

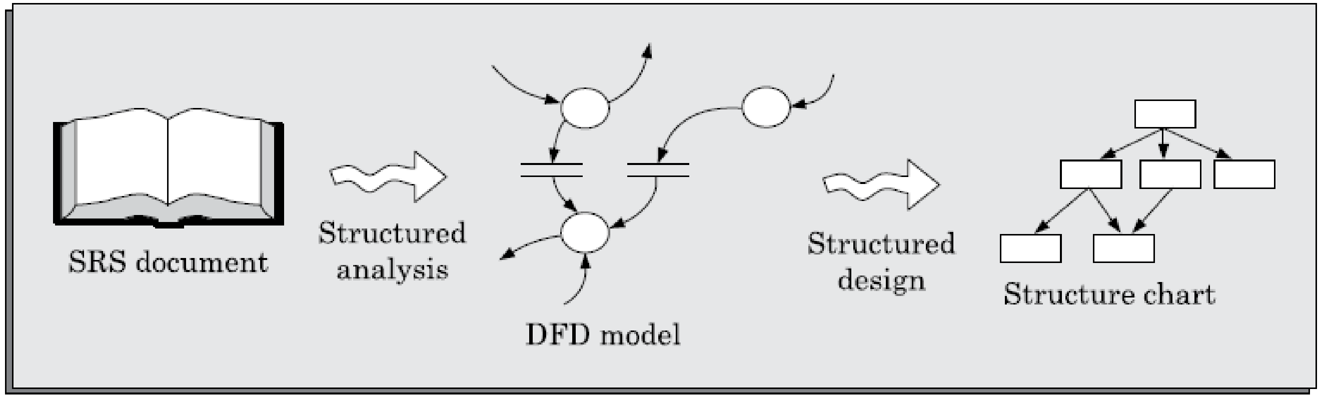
Structured Design transforms the results of **Structured Analysis** (i.e., the **DFD model**) into a **Structure Chart**, which defines the **software architecture**.

✅ <u>What is a Structure Chart?</u>

A **Structure Chart** is a **graphical representation** of:

- Software **modules**
- **Hierarchy** of module calls (i.e., who calls whom)
- **Data passed** between modules

- It **does NOT show** how the functionality is achieved (procedural logic is not included). Instead, it shows the **module structure and interactions**.



✅ Basic Components of a Structure Chart:

Symbol	Description
📦 Rectangle	Represents a module
➡️ Arrow (between rectangles)	Shows module invocation (caller → callee)
🔁 Loop	Indicates repetition of module calls
🔷 Diamond	Indicates selection/decision (only one of many modules is invoked)
➡️ Small arrow near control line	Shows data flow between modules
📦 Double-edged rectangle	Represents a library module (called frequently)

✅ Rules:

- Only **one root module** (top-level).
- **No cyclic calls**: If module A calls B, B cannot call A back.
- Modules are **arranged in layers**: Lower-level modules should **not know about** higher-level ones.
- **Different higher-level modules** can call the **same lower-level module**.

However, it is possible for two higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design are shown in Figure 6.18.

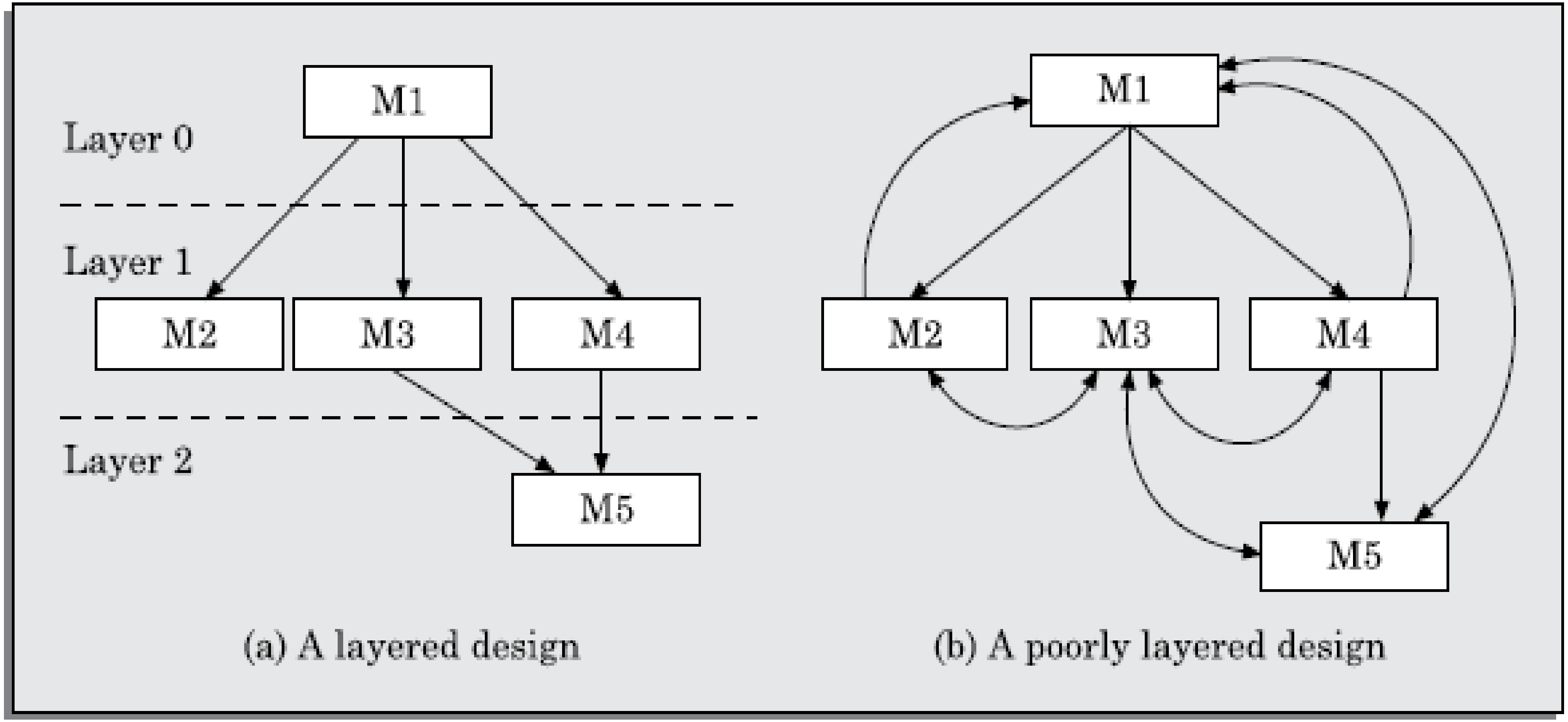


FIGURE 5.5 Examples of properly and poorly layered designs.

📌 Difference Between Flowchart and Structure Chart:

Flowchart	**Structure Chart**
Shows control flow	Shows module hierarchy
Difficult to identify modules	Clear module representation
No data flow shown	Shows data flow between modules
Sequential in nature	Suppresses sequencing

6.4.1 Transformation of DFD to Structure Chart

To convert a DFD to a structure chart, two techniques are used:

- 🔸 **1. Transform Analysis**
- 🔸 **2. Transaction Analysis**

You choose the technique **based on the DFD's input structure**:

🔹 When to Use Which?

Criteria	Use...
All inputs go to a single bubble	✅ Transform Analysis
Inputs go to different bubbles (i.e., multiple entry points)	✅ Transaction Analysis

✅ Transform Analysis (For Simple Processes)

- **Goal**: Convert the DFD into 3 parts:
- **Input part**: Converts data from physical → logical (called **afferent branch**).
- **Processing part**: Main logic (called **central transform**).
- **Output part**: Converts data from logical → physical (called **efferent branch**).

Steps:

Identify input, processing, and output parts from the DFD.

- Create modules for:
    - Input
    - Output
    - Central processing
- Place them under a **root module**.
- Refine structure chart:
    - Break high-level modules into **submodules** (called **factoring**).
    - Add: initialization modules, error handlers, read/write modules, etc.
- **Goal**: Continue factoring until **every bubble in the DFD** is represented.

🧠 **Tip:**

- Processes that just **validate or receive input** → not part of the central transform.
- Processes that **filter, sort, or manipulate** data → are part of the **central transform**.

✅ Transaction Analysis (Not covered in detail in 6.4, but hinted)

- Applied when **different input types trigger different processes**.
- Used in **interactive or menu-based systems**.

Criteria	Use...
All inputs go to a single bubble	✅ Transform Analysis
Inputs go to different bubbles (i.e., multiple entry points)	✅ Transaction Analysis

Concept	Explanation
Structured Design	Converts DFD into implementable structure chart
Structure Chart	Shows modules and their interaction
Transform Analysis	Divides system into input → process → output
Factoring	Refining each module into smaller submodules
Module Hierarchy	Follows top-down and no back-invocations
Flowchart ≠ Structure Chart	Flowcharts focus on steps; structure charts focus on module interactions

PROBLEM 5.1 Draw the structure chart for the RMS software of Example 5.1.

- ***Solution:*** By observing the level 1 DFD, we can identify validate-input as the afferent branch and write-output as the efferent branch. The remaining (i.e., computerms) as the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in Figure 5.6.

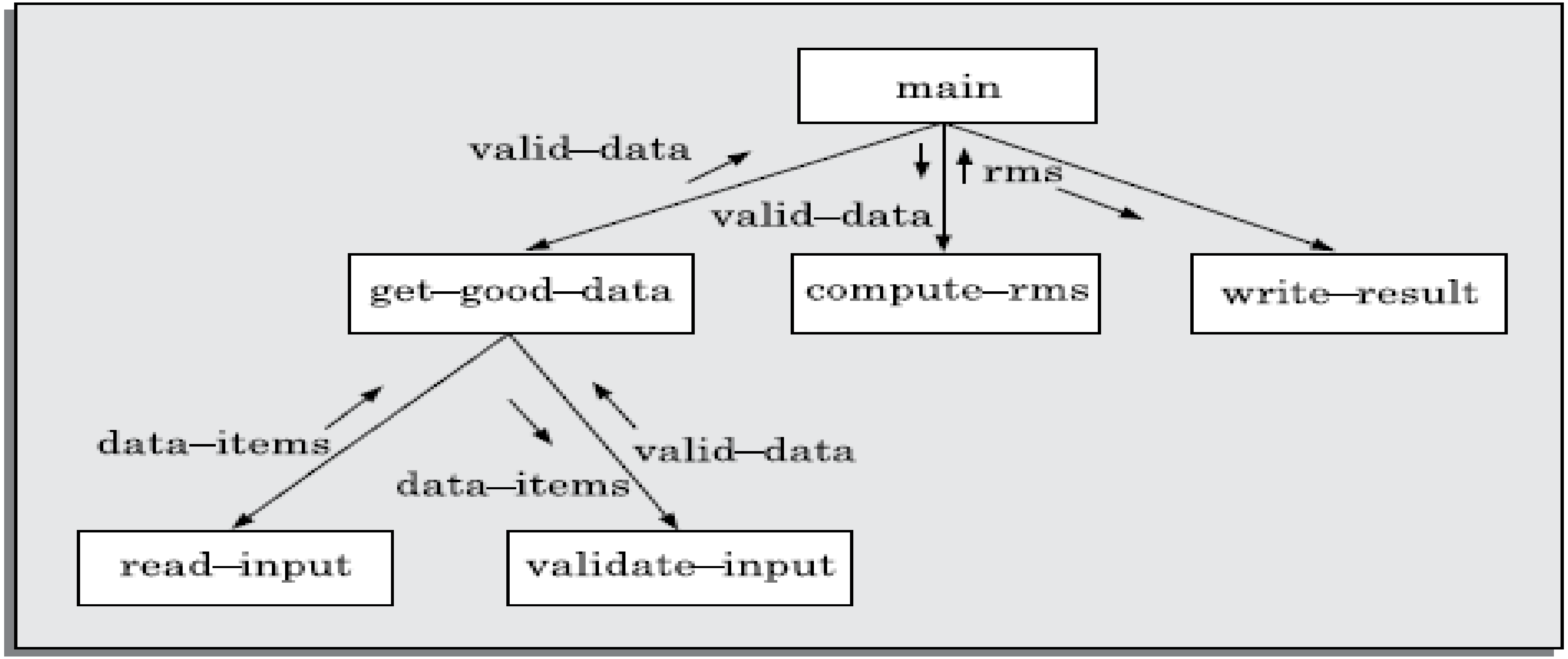
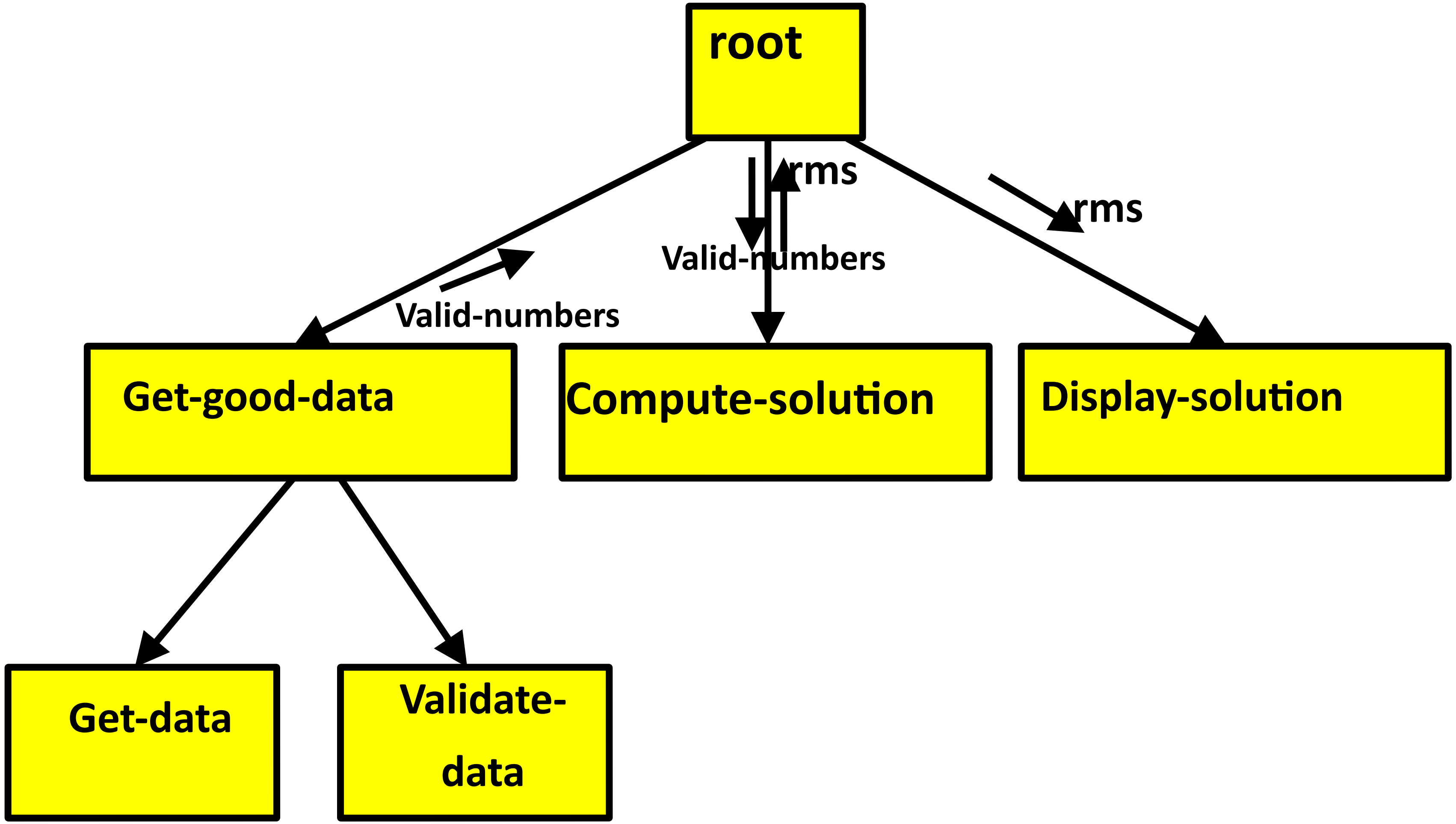


FIGURE 5.6 Structure chart for Problem 5.1.

Example 1: RMS Calculating Software



✅ Structure Chart Explanation (for RMS Software)

This structure chart is derived by applying **Transform Analysis** on the Level 1 DFD.

🔹 Root Module:

- **main**: Top-level module that controls the program execution.
  - It calls three major submodules:
    - get-good-data
    - compute-rms
    - write-result

🔹 Afferent Branch (Input Handling):

- **get-good-data**: Responsible for acquiring and validating user input.
  - It further breaks down into:
    - **read-input**: Reads 3 integer values from the user.
    - **validate-input**: Checks if values are within the valid range.

→Data passed: data-items → valid-data

🔹 **Central Transform:**

- **compute-rms**: Performs the RMS calculation using the validated input.
  - Takes valid-data as input.
  - Returns rms value.

🔹 **Efferent Branch (Output Handling):**

- **write-result**: Displays the RMS result to the user.
  - Uses the computed rms data.

🔄 **Data Flow:**

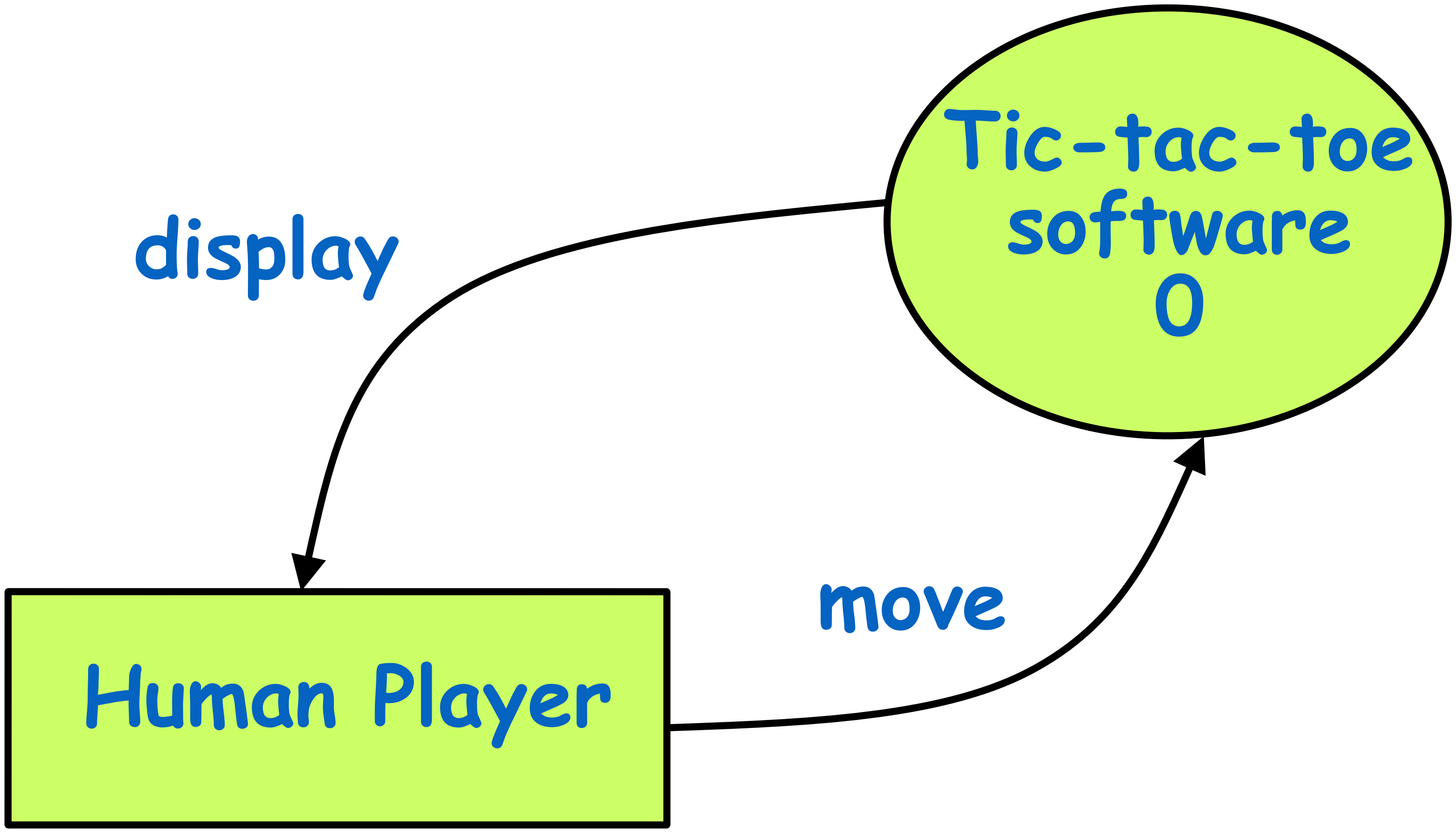
- Arrows represent data flow between modules.
  - For example:
    - valid-data is passed from get-good-data → compute-rms
    - rms is returned to main and passed to write-result

Component	**Role**
main	Root module managing the flow
get-good-data	Handles reading and validating input
compute-rms	Calculates root mean square
write-result	Displays result to user
read-input & validate-input	Submodules of input handling

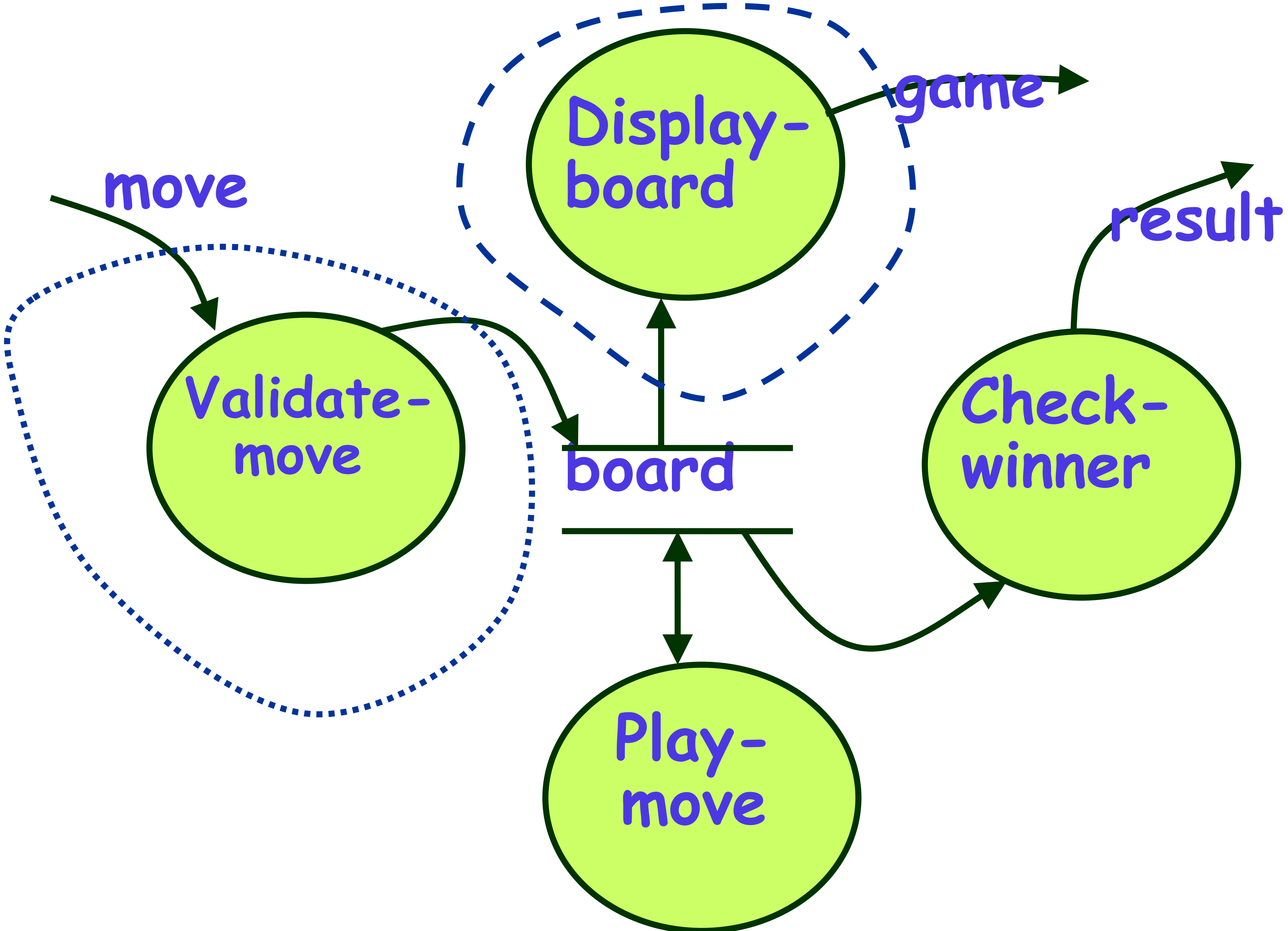
Example 2: Tic-Tac-Toe Computer Game

- 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

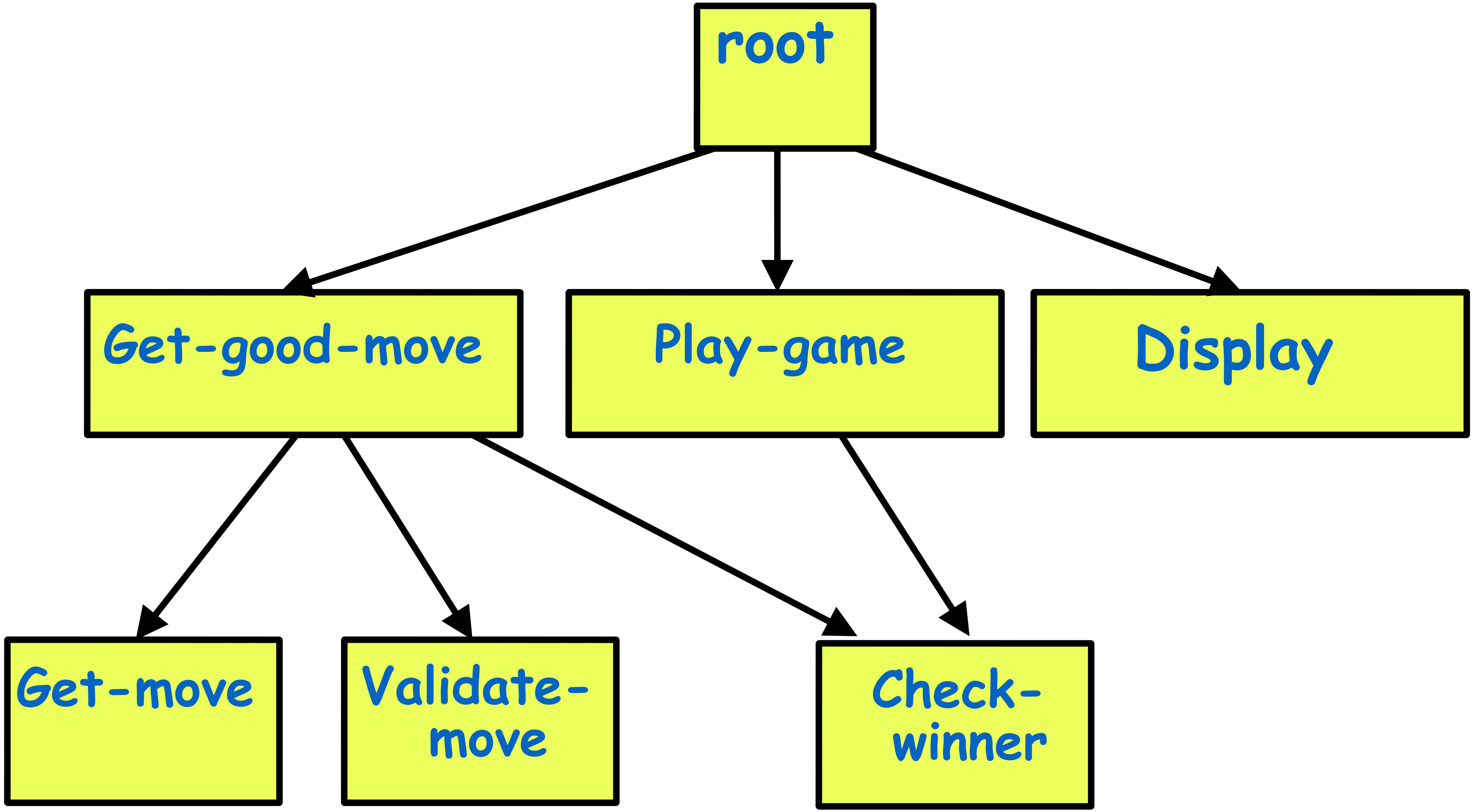


Level 1 DFD



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

Structure Chart



Transaction Analysis

Transaction analysis is a technique used in **structured design** to convert a Data Flow Diagram (DFD) into a **structure chart**, especially for **transaction-driven systems**. It's an **alternative** to **transform analysis**.

🔸 What is a Transaction?

A **transaction** is a specific task a user performs using the system.

- 📌 **Examples**:
  - Issue book
  - Return book
  - Query book availability
- Each type of transaction has a **distinct processing path** through the system.

🔸 Key Characteristics of Transaction-Driven Systems

- **Input data** may follow **different paths** through the DFD depending on the transaction type.
- Contrasts with **transform-centered systems**, where **all input data follow the same path**.

◆ Steps for Transaction Analysis

1. Identify input data items:

- Look at the **dangling arrows** in the DFD — these represent inputs.

2. Identify transactions:

- Count how many **bubbles (processes)** the input data are directed to.
- Each distinct process indicates a **separate transaction**.
- Some transactions may not need input data and are identified based on prior **experience**.

3. Trace each transaction path:

- Follow the data flow from **input** to **output** for each transaction.
- All the **bubbles** traversed form the logic of that transaction.

4. Map each transaction to structure chart modules:

- Create a **root module**.
- Under the root, draw **one module per transaction**.
- Each transaction module includes all the processing for that specific transaction.

5.Use tags for transaction types:

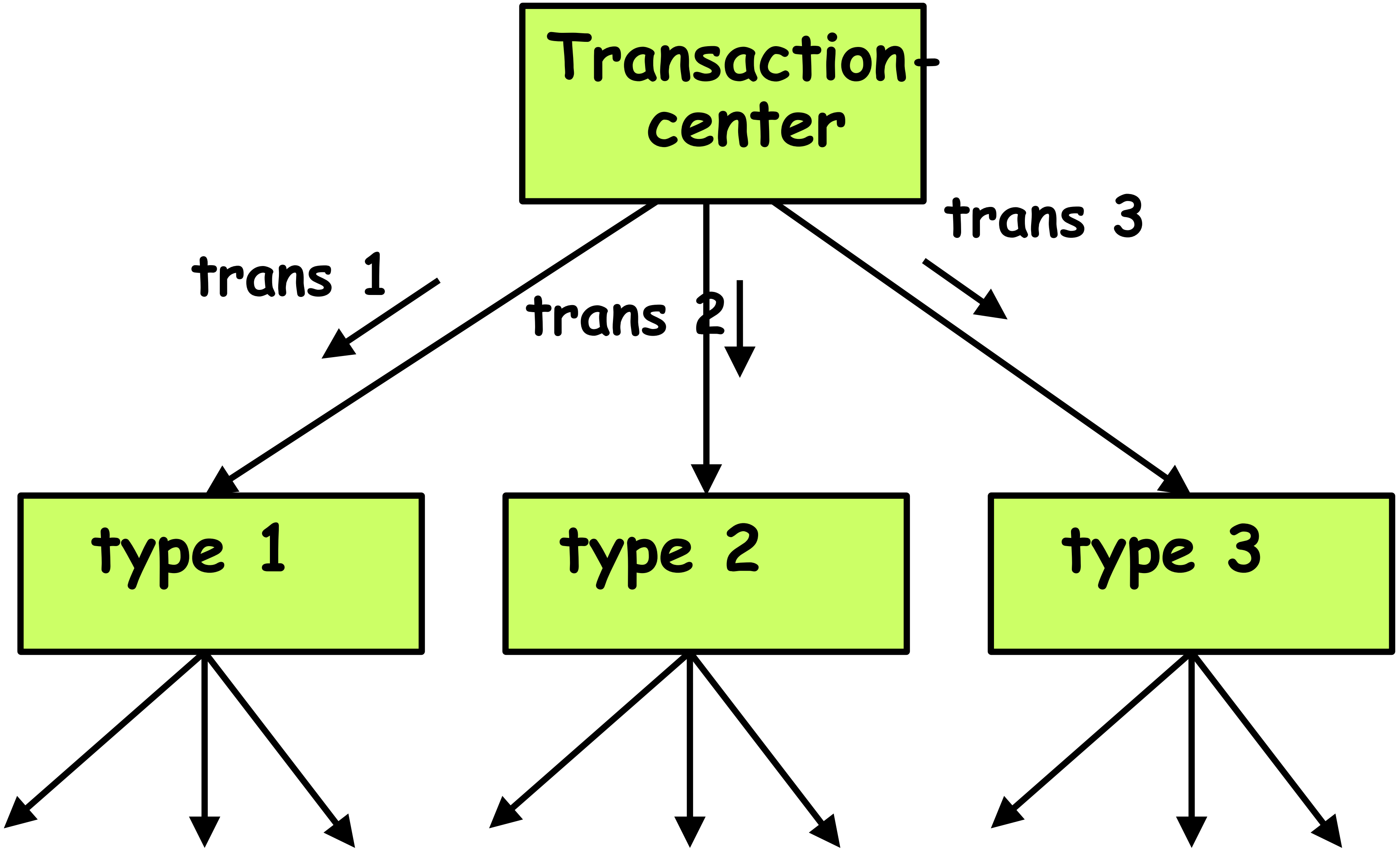
- These help the system know **which transaction logic to execute**.

🔹 Structure Chart Characteristics (Transaction-Based)

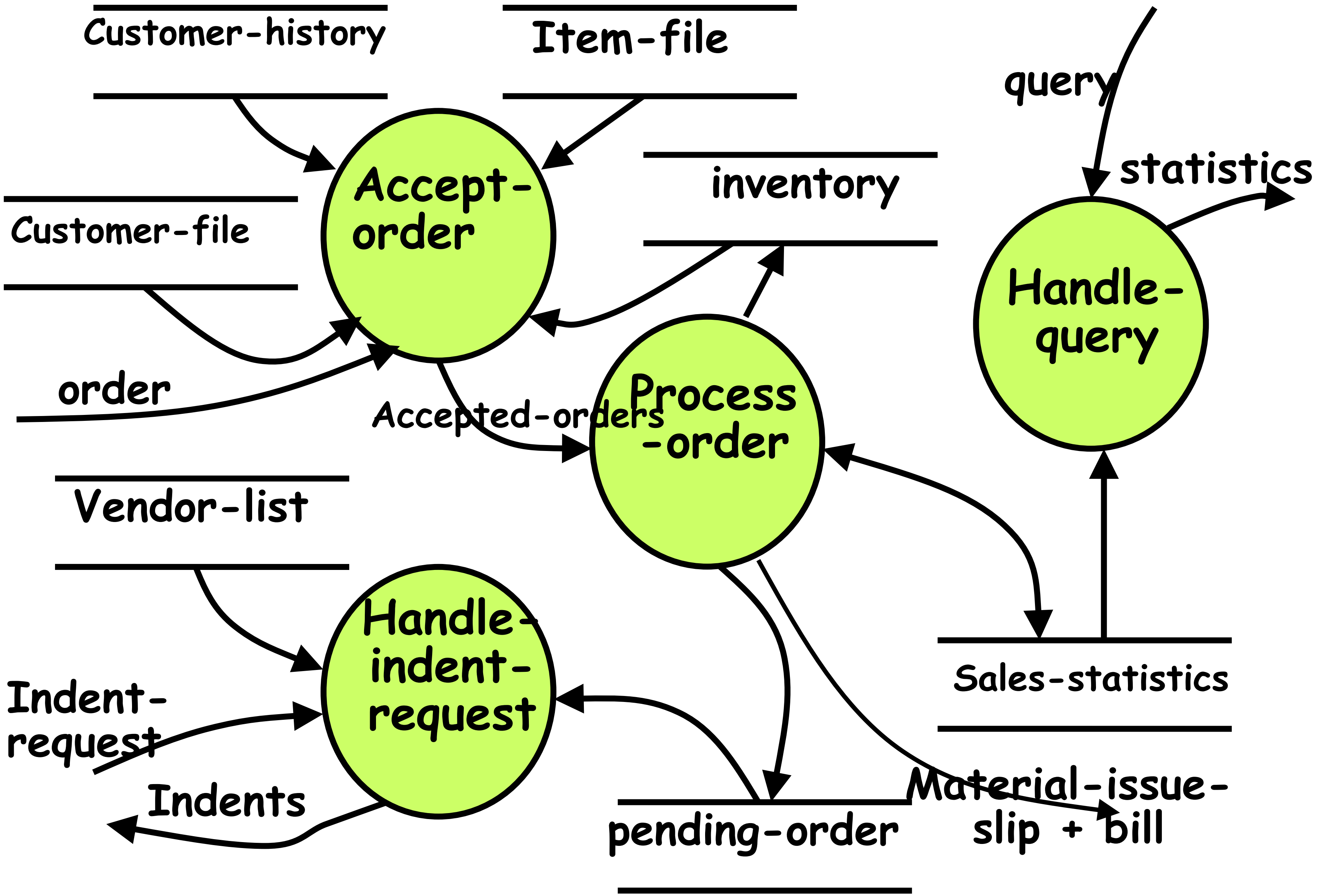
- **Root module**: Controls the flow.
- **Child modules**: Each handles a specific transaction.
- **No duplication**: Shared functionality may be placed in reusable modules.
- **Flexible**: Easy to add new transactions later.

Feature	Transaction Analysis
Best for	Transaction-based systems
Input path	Different for each transaction
Output	Depends on input type (transaction tag)
Modules	One per transaction under a root module
Example	Library system: issue, return, query book

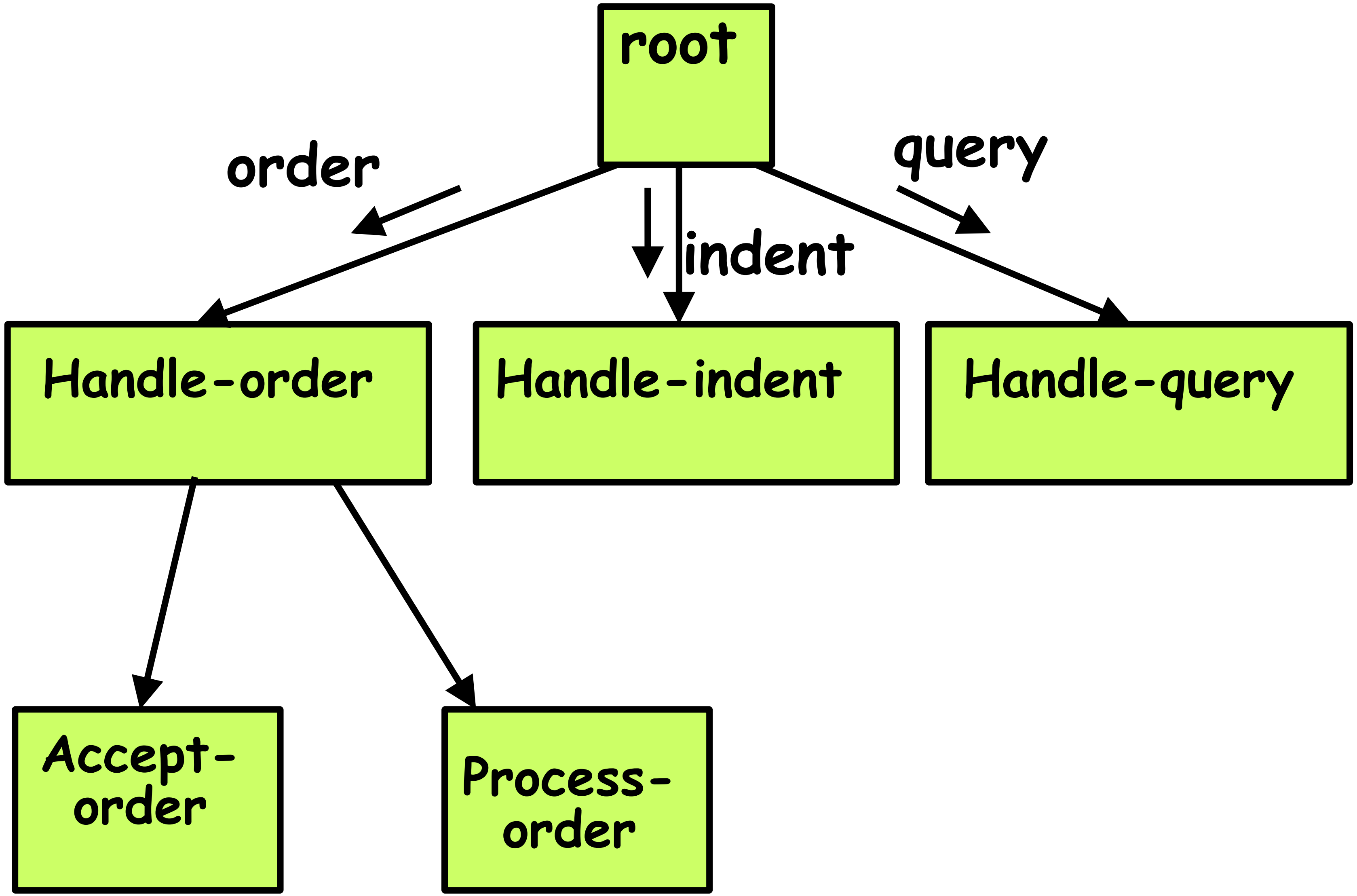
Transaction analysis



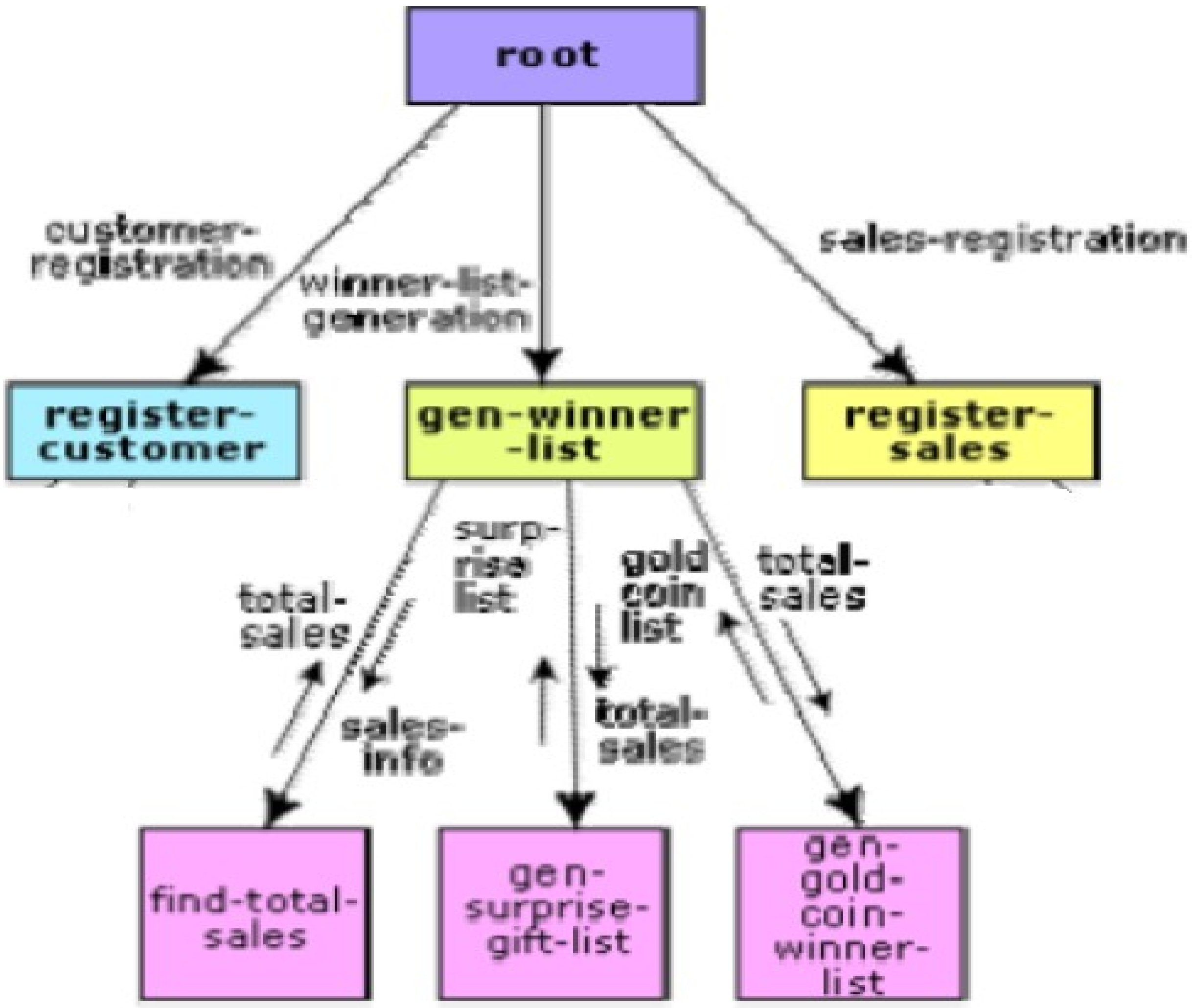
Level 1 DFD for TAS



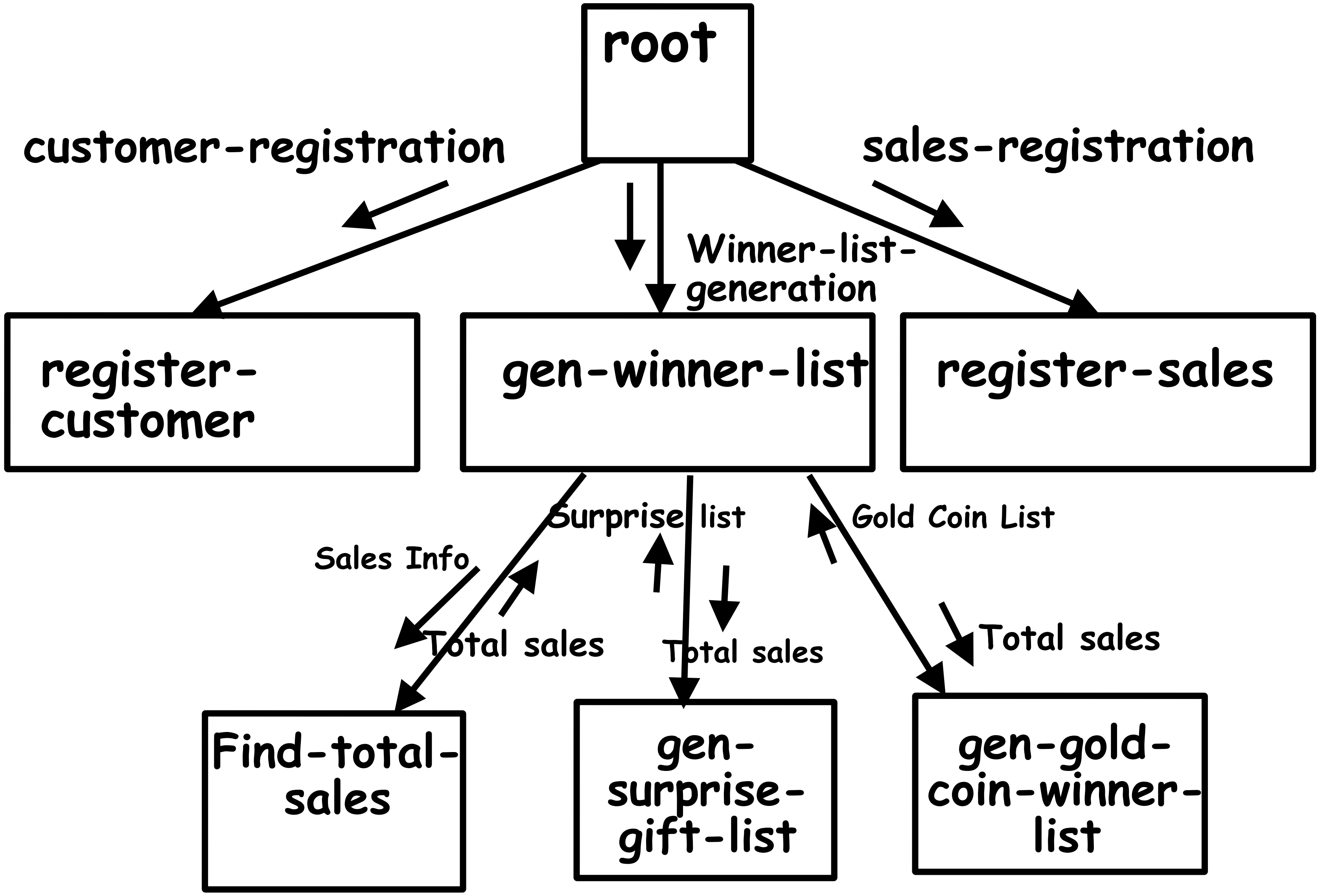
Structure Chart



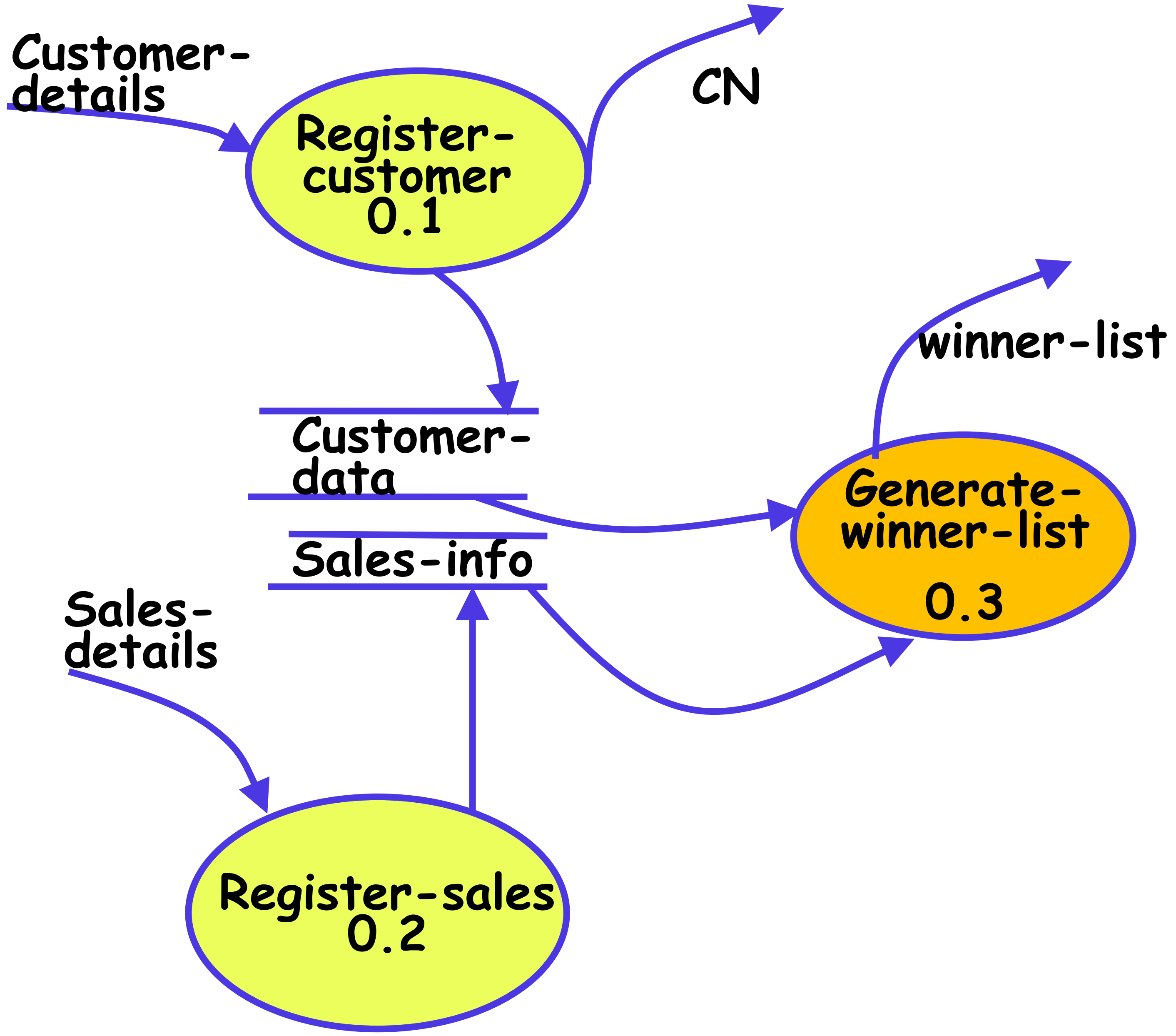
Structure Chart: Supermarket Prize Scheme



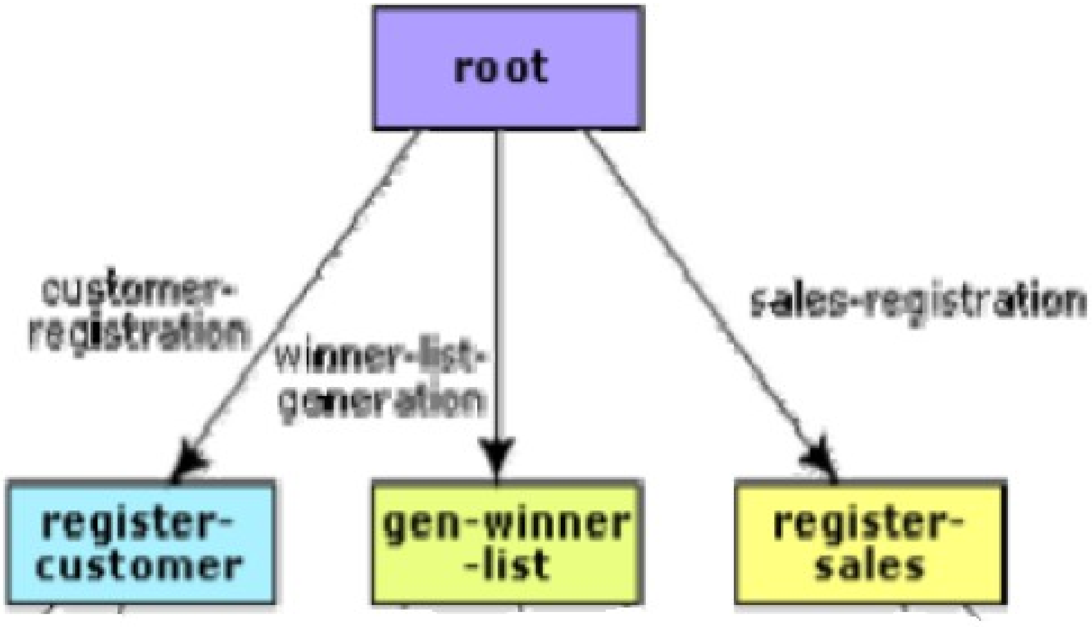
Structure Chart



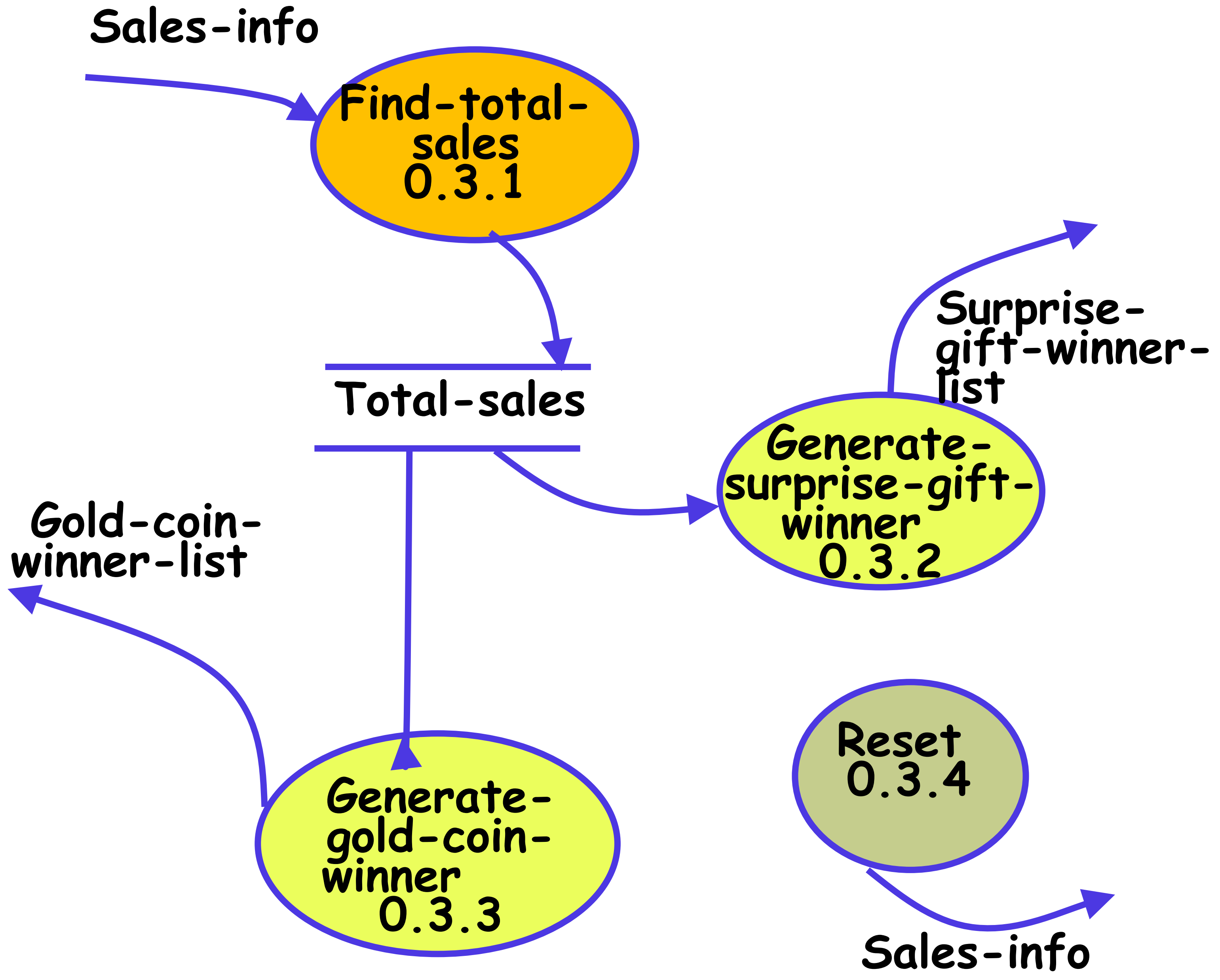
Level 1 DFD: Supermarket Prize Scheme



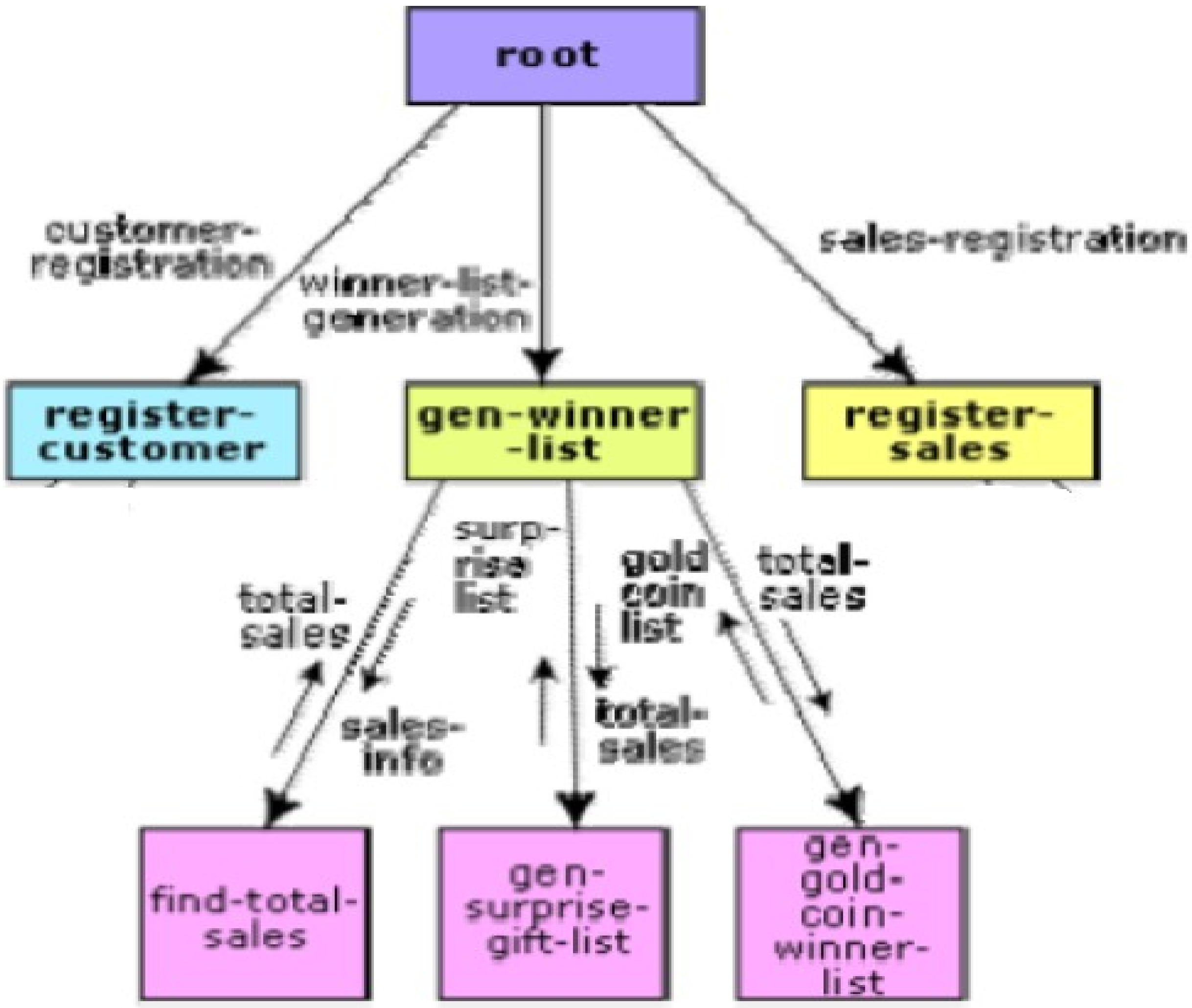
Structure Chart: Supermarket Prize Scheme



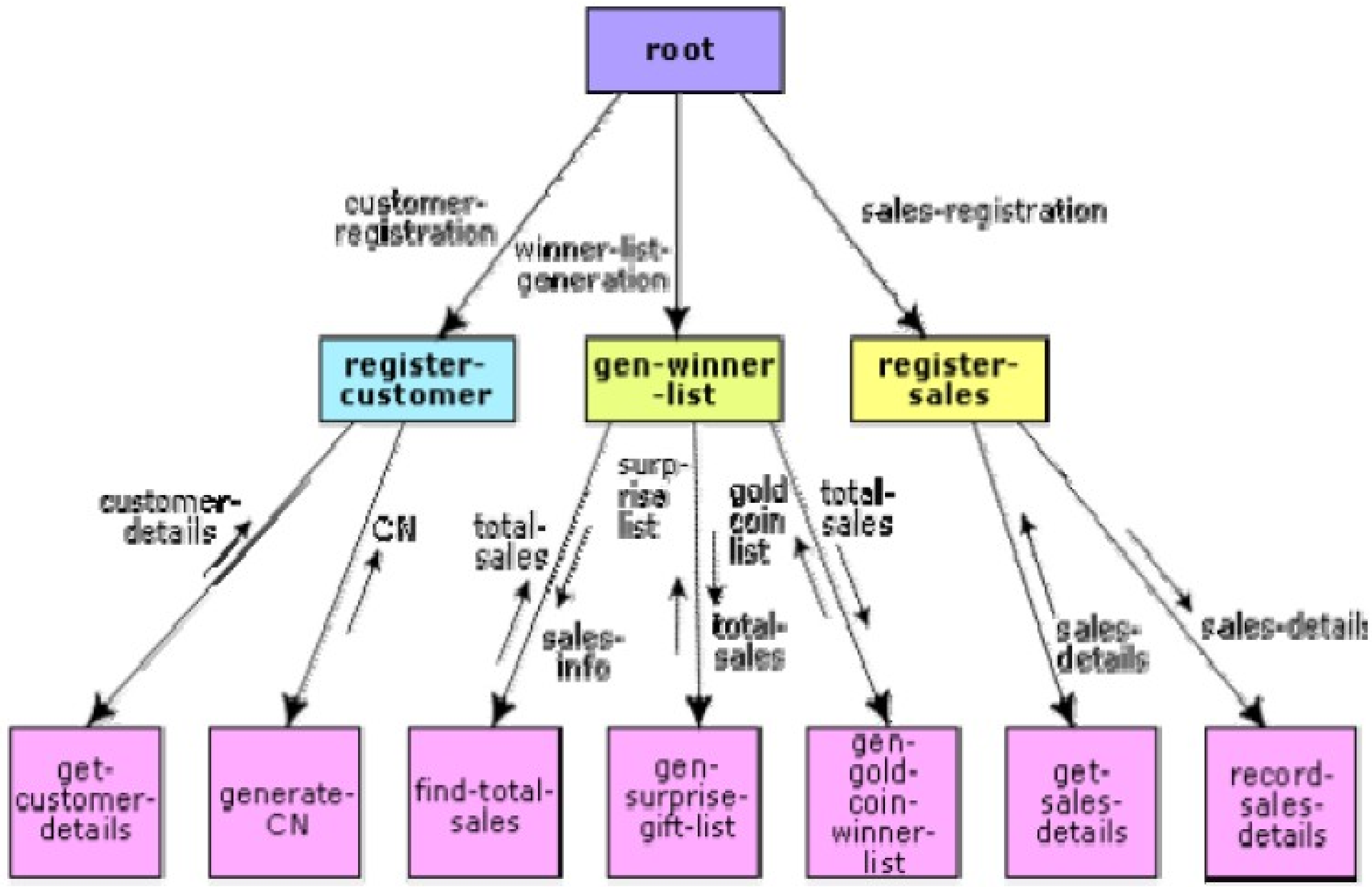
Level 2 DFD: Supermarket Prize Scheme



Structure Chart: Supermarket Prize Scheme



Structure Chart: Supermarket Prize Scheme



5.5 DETAILED DESIGN

Once the structure chart has been created during structured design, the **next step is detailed design**.

◆ Purpose of Detailed Design

- To specify **how** each module in the structure chart **actually works** — in terms of logic and data structures.

◆ Main Outputs of Detailed Design

- **Module Specifications (MSPEC)**:
  Describes what **each module** does in more detail.
- **Data Structures**:
  Defines the **data** used or manipulated in the module.

◆ What is MSPEC? (Module Specification)

- Written in **structured English** or **pseudo-code**.
- Helps bridge the gap between design and coding.

Two types of MSPEC:

Module Type	**Description Style**	**Contents**
Non-leaf modules	Control logic	Describe conditions under which it calls lower-level (child) modules.
Leaf modules	Algorithmic logic	Specify step-by-step logic of what the module does internally.

🔸 How to Develop MSPECs

To write the **MSPEC** for any module, refer to:

- ✅ The **DFD model** (for understanding data flow)
- ✅ The **SRS document** (for knowing the required functionality)

Component	**Purpose**
Detailed Design	Converts structure chart modules into implementable module logic
MSPEC	Describes logic of each module in pseudo code or structured English
Leaf Module MSPEC	Describes algorithmic steps
Non-leaf MSPEC	Describes control flow and delegation to child modules

<u>5.6 DESIGN REVIEW</u>

After completing the software design, it **must be reviewed** by a qualified team to ensure **quality, correctness, and feasibility**.

🔸 Who Participates in the Review?

- The review team usually includes people from different roles:
  - ✅ **Designers**
  - ✅ **Developers (coders)**
  - ✅ **Testers**
  - ✅ **Analysts**
  - ✅ **Maintainers**

❖ **These members may or may not be part of the original design team.**

🔶 Key Focus Areas of the Review

The review team evaluates the design based on these **important criteria**:

1. Traceability

- ➢Can every **DFD bubble** be matched to a module in the structure chart?
- ➢Can each **SRS functional requirement** be traced to the DFD and structure chart?

2. Correctness

- ➢Are the **algorithms and data structures** used in detailed design **logically correct**?

3. Maintainability

- ➢Is the design **simple, modular, and easy to modify** in the future?

4. Implementability

- ➢Can the design be **efficiently implemented** in code?

🔸 What Happens After the Review?

- The designers must **address all concerns** and **suggestions** raised by the review team.
- Once everything is resolved, the **design document is approved** and becomes ready for **coding (implementation)**.

Review Aspect	Purpose
Traceability	Ensures connection from SRS → DFD → Structure Chart
Correctness	Checks logic and structure of algorithms and data used
Maintainability	Confirms design is adaptable for future changes
Implementability	Verifies design can be coded and executed efficiently