Software Engineering (Code: CSE3122)

B. Tech CSE 5th Sem.

(School of Computer Engineering)

Dr. Kranthi Kumar Lella
Associate Professor,
School of Computer Engineering,
Manipal Institute of Technology,
MAHE, Manipal – 576104.

SOFTWARE ENGINEERING

Module – 1 (INTRODUCTION)

CONTENTS

Rapid Growth in Computer Technology

- ➢ Over the past **60 years**, computers have evolved from slow and simple machines to highly powerful and sophisticated systems.
- ➢ Their **performance has increased rapidly** while **costs have dropped dramatically**.
- ➢ Analogy: If airplanes had progressed like computers, you'd have **mini-airplanes costing as little as bicycles**, flying at **1000x supersonic jet speed**.
- ➢ This **extraordinary pace** of growth in computing power has **no parallel** in any other human field.

Impact on Software

- ➢ As hardware got more powerful, software needed to handle **more complex and larger problems**.
- ➢ Software engineers had to **improve their practices** continuously to write efficient, scalable, and reliable software.
- ➢ They did this by building on **past experience and innovations**, leading to the development of **software engineering** as a field.

What is Software Engineering?

- It is defined as:
  - “A **systematic collection** of good program development practices and techniques.”
  - Or: “An **engineering approach** to develop software.”

  IEEE Definition: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
- These practices evolved through **research and real-world experience**, making development more efficient and reliable.

Analogy: Petty Contractor vs. Professional Engineer

- A petty contractor might successfully build a **small house** using intuition but would fail in building a **50-storey complex** due to lack of engineering knowledge.
- Likewise, **small software projects** can be built without formal methods, but **large-scale systems** need **planning, analysis, design, and testing**—just like in civil engineering.
- Therefore, **software engineering** is essential for developing complex, real-world systems.

Is Software Engineering an Art or Engineering?

- **Not just art**: Unlike art, software engineering uses **well-documented, structured methods**, not just creativity.
- **Not pure science**: Unlike science, it deals with **trade-offs**, not always one perfect solution.
- Like other engineering fields, it combines:
  - **Experience-based rules**
  - **Theoretical knowledge**
  - **Systematic decision-making**

❖Software engineering emerged due to the **growing demands on software**, driven by rapid advances in hardware. It evolved from the **Natural Art** into a **structured engineering discipline**, much like civil or mechanical engineering—making it essential for building complex, high-quality software systems.

1.1 Evolution from an art form to an engineering discipline

Software engineering didn’t start as a formal field. Over the past 60 years, it has evolved in **three main stages**:

1.1.1 Evolution of an Art into an Engineering Discipline

- In the **early days**, software development was an *art*, done in an **ad hoc** way using the **exploratory (code-and-fix)** method.
- Programmers relied on **intuition, experience, and trial-and-error**—there were no standard rules or processes.
- Good programmers were seen as *artists*, using their own “tricks,” while others struggled to replicate their work.
- As software systems grew more complex, this informal style **led to poor quality, high cost, and unmaintainable code**.

- Over time, with the contribution of many researchers and developers, this approach evolved into a **craft**—and eventually into a formal **engineering discipline** using tested principles, planning, documentation, and teamwork.

1.1.2 Evolution Pattern for Engineering Disciplines

Pattern of evolution is not very different from that seen in other engineering disciplines. Irrespective of whether it is iron making, paper making, software development, or building construction; evolution of technology has followed strikingly similar patterns. This pattern of technology development has schematically been shown in Figure 1.1.

- Like **iron-making or construction**, software development followed a common path:
  - **Art**: Secret knowledge, based on personal skill
  - **Craft**: Shared techniques, taught to apprentices
  - **Engineering**: Scientific, documented methods
- In software:
  - Early programming was an art.
  - Then, programmers shared knowledge, making it a craft.
  - Eventually, systematic research and structured practices formed **software engineering** as a professional discipline.

1.1 Evolution from an art form to an engineering discipline (cntd..)

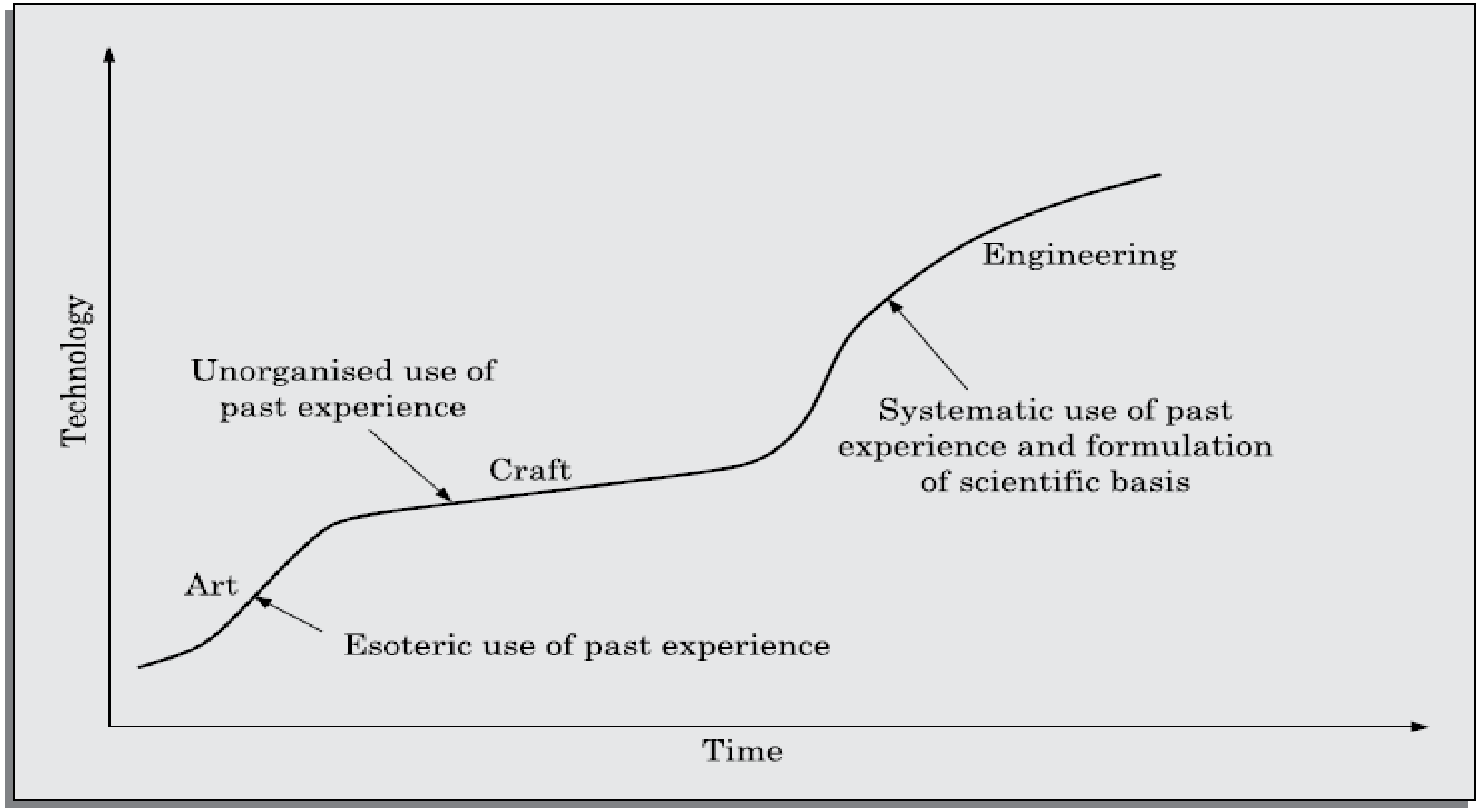


FIGURE 1.1 Evolution of technology with time.

<u>1.1.3 A Solution to the Software Crisis</u>

- Today, **software costs more than hardware**, and large systems often face problems like:
  - Delays
  - Bugs and crashes
  - Difficult maintenance
  - Failing to meet user needs

This is called the **software crisis**.

<u>Causes</u>:

- Rapidly growing complexity
- Lack of trained professionals
- Outdated or inefficient development methods

<u>Solution</u>:

- Broader use of **software engineering principles**
- Further **advancements in the discipline**
- Training developers in systematic, structured approaches

1.1.3 A Solution to the Software Crisis

To understand the present software crisis, consider the following facts. The expenses that organizations all over the world are incurring on software purchases as compared to the expenses incurred on hardware purchases have been showing a worrying trend over the years (see Figure 1.2).

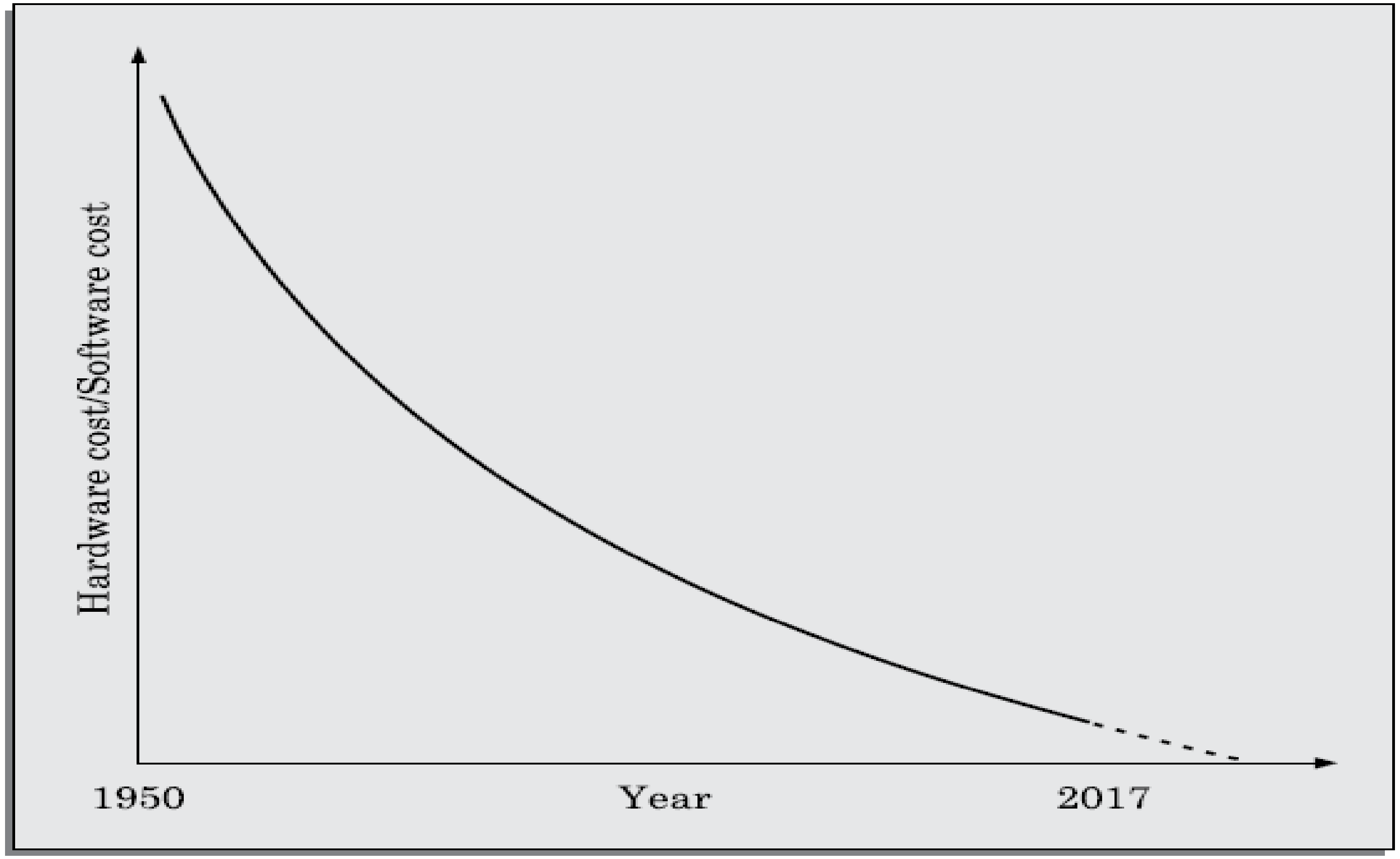


FIGURE 1.2 Relative changes of hardware and software costs over time.

- Software development began as an **art form** driven by personal skill. Over time, it became a **craft** and then a **mature engineering discipline**. Today, software engineering is essential to handle complex, large-scale systems cost-effectively and to address the **ongoing software crisis**.

1.2 A Solution to the Software Crisis

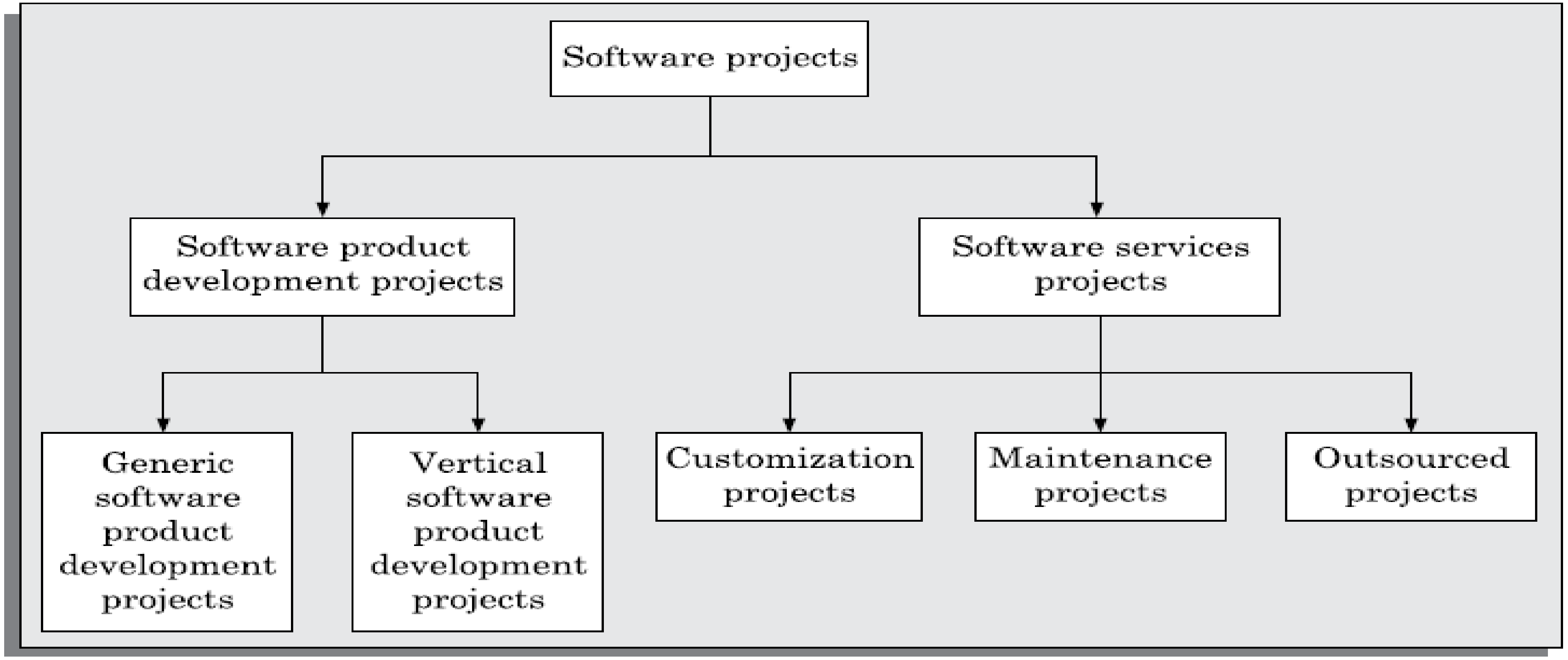
1.2.1 Programs vs. Products

- **Programs (Toy Software):**
  - Created by individuals (e.g., students, hobbyists).
  - Small in size, limited features.
  - Poor user interface, no formal documentation.
  - Author is often the only user and maintainer.
  - Not reliable, hard to maintain, often inefficient.

- **Products (Professional Software):**
  - Built for multiple users by teams.
  - Have user manuals, requirement specs, design and test documents.
  - Carefully designed, tested, and documented.
  - Usually too large and complex for a single person to develop.
  - Require **systematic development methods** (software engineering).
  - Example: Microsoft Office, Oracle DB.

1.2.2 Types of Software Development Projects

- A software development company typically has a large number of on-going projects. Each of these projects may be classified into software product development projects or services type of projects. These two broad classes of software projects can be further classified into subclasses as shown in **Figure 1.3.**



Software projects
Software product development projects
Software services projects
Generic software product development projects
Vertical software product development projects
Customization projects
Maintenance projects
Outsourced projects

FIGURE 1.3 A classification of software projects.

Software projects are classified into two broad types:

A. Software Product Development

- **Generic Products**:
  - Sold to a wide range of users (horizontal market).
  - Off-the-shelf software like MS Windows, Oracle, etc.
  - Developed based on common user needs.

- **Domain-Specific Products**:
  - Targeted to specific industries (vertical market).
  - Examples:
    - **BANCS** (TCS),
    - **FINACLE** (Infosys) for banking,
    - **AspenPlus** for chemical simulation.

B. Software Services

Includes customization, outsourcing, testing, maintenance, consultancy.

- **Dominant in today's market due to:**
  - Large existing codebase,
  - Heavy code reuse,
  - Shorter project timelines (weeks/months instead of years).

- **Outsourced Projects**:
  - A company hands off part of a larger project to another firm.
  - Reasons: lack of expertise, cost-saving, faster delivery.
  - Lower risk but usually one-time revenue.

1.2.3 Software Projects by Indian Companies

- **India’s strength**: Software **services** (e.g., outsourcing, maintenance).
- **Recent shift**: Moving toward **product development**, though still cautious due to:
  - High investment,
  - Market acceptance risks.
- **Example products**: FINACLE (Infosys), Tally ERP.
- 🔺 Trend: Global **software services are growing faster** due to **cloud computing** and **application service provisioning.**

1.3 Exploratory Style of Software Development

Exploratory Style: What Is It?

- **Exploratory development style** is informal and intuitive. Programmers code based on **their own judgment**, without following systematic software engineering principles.
- It starts with a **brief from the customer**, followed by **coding**, then **testing and fixing**, repeated until the program works.
- This is also called **build-and-fix** model. (See Fig 1.4: Coding → Testing → Fixing → Repeat.)

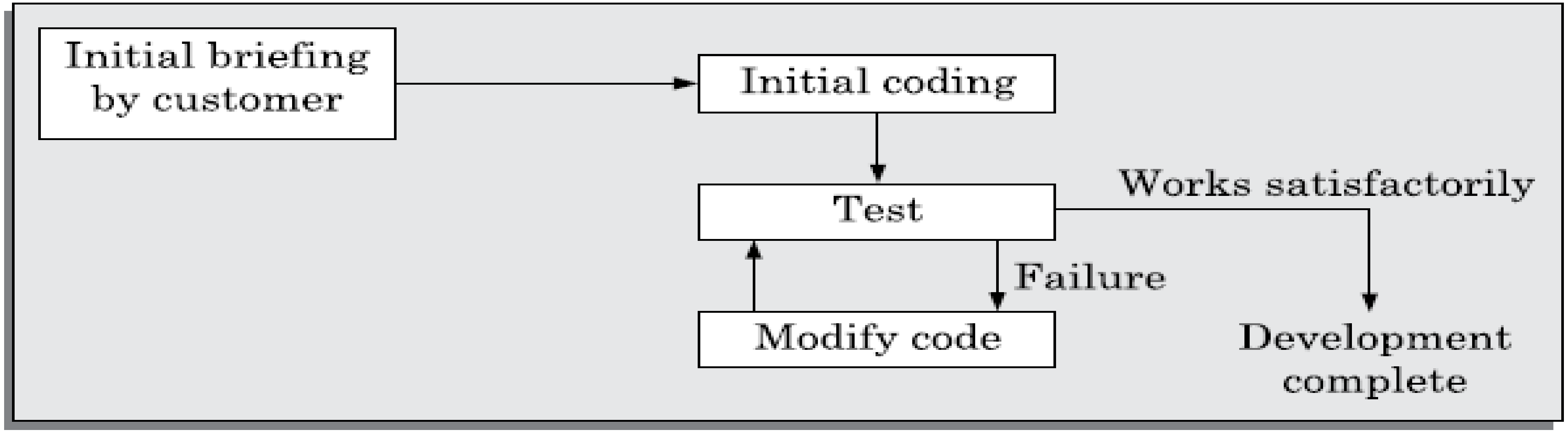


FIGURE 1.4 Exploratory program development.

<u>When It Works (and When It Doesn't)</u>

- Works **only for small programs** (like student assignments).
- **Fails for large, professional software** because:
    - Time and effort increase **exponentially** with program size.
    - Programs become **unmaintainable**.
- Hard to use in **team environments** (due to lack of design and documentation).

Why It Fails: The Human Factor

- Humans have **limited short-term memory** (about 7±2 items) — this limits our ability to manage complexity.
- As program size increases, **perceived complexity** (psychological difficulty) also increases **exponentially**, not linearly.
- Fig 1.5 shows:
  - **Thick line**: Effort using exploratory style (exponential rise).
  - **Thin line**: Effort using software engineering principles (almost linear).
  - **Dotted line**: Ideal case, if a machine wrote the code.

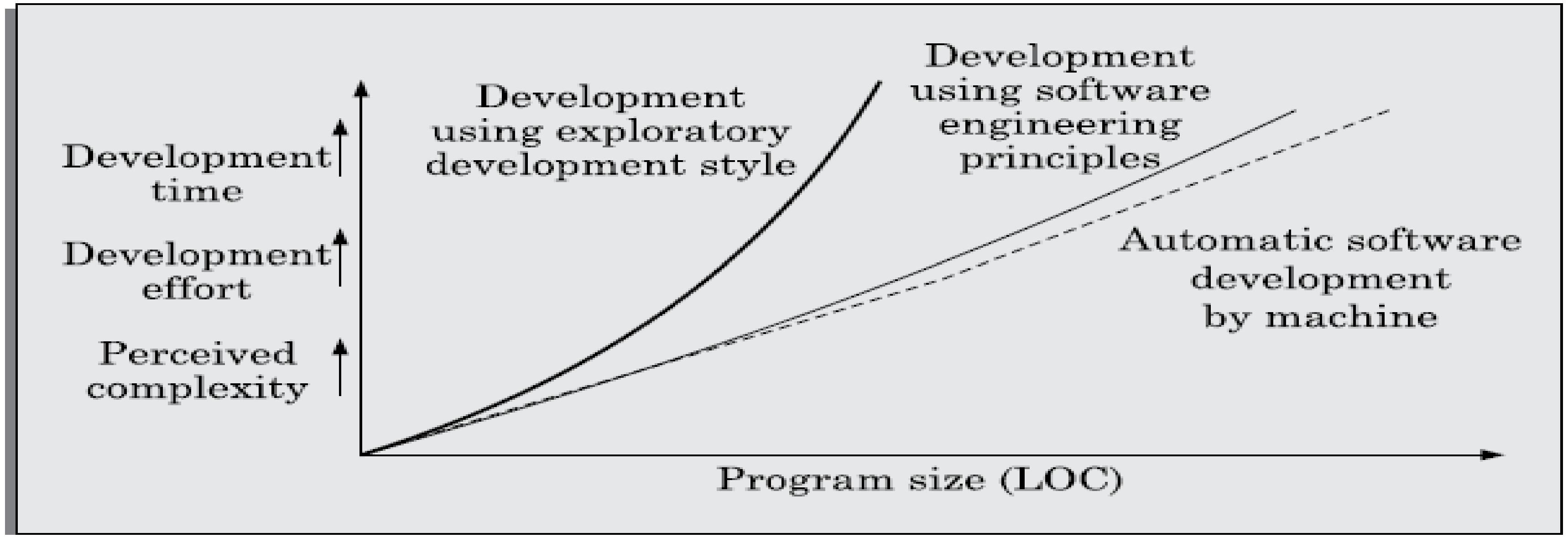


FIGURE 1.5 Increase in development time and effort with problem size.

1.3.1 Perceived Problem Complexity—Human Cognition Mechanism

What Is Perceived Complexity?

- **Perceived (or psychological) complexity** refers to how **difficult a problem feels** to a human programmer, **not** the computational complexity (like time or space complexity).
- As **problem size increases**, perceived complexity **rises rapidly**, especially when using the **exploratory development style**.
- **Software engineering principles** help keep this rise under control by addressing **human cognitive limitations**.

- Psychologists says that the human memory can be thought to consist of two distinct parts [Miller, 1956]: short-term and long-term memories. A schematic representation of these two types of memories and their roles in human cognition mechanism has been shown in Figure 1.6.

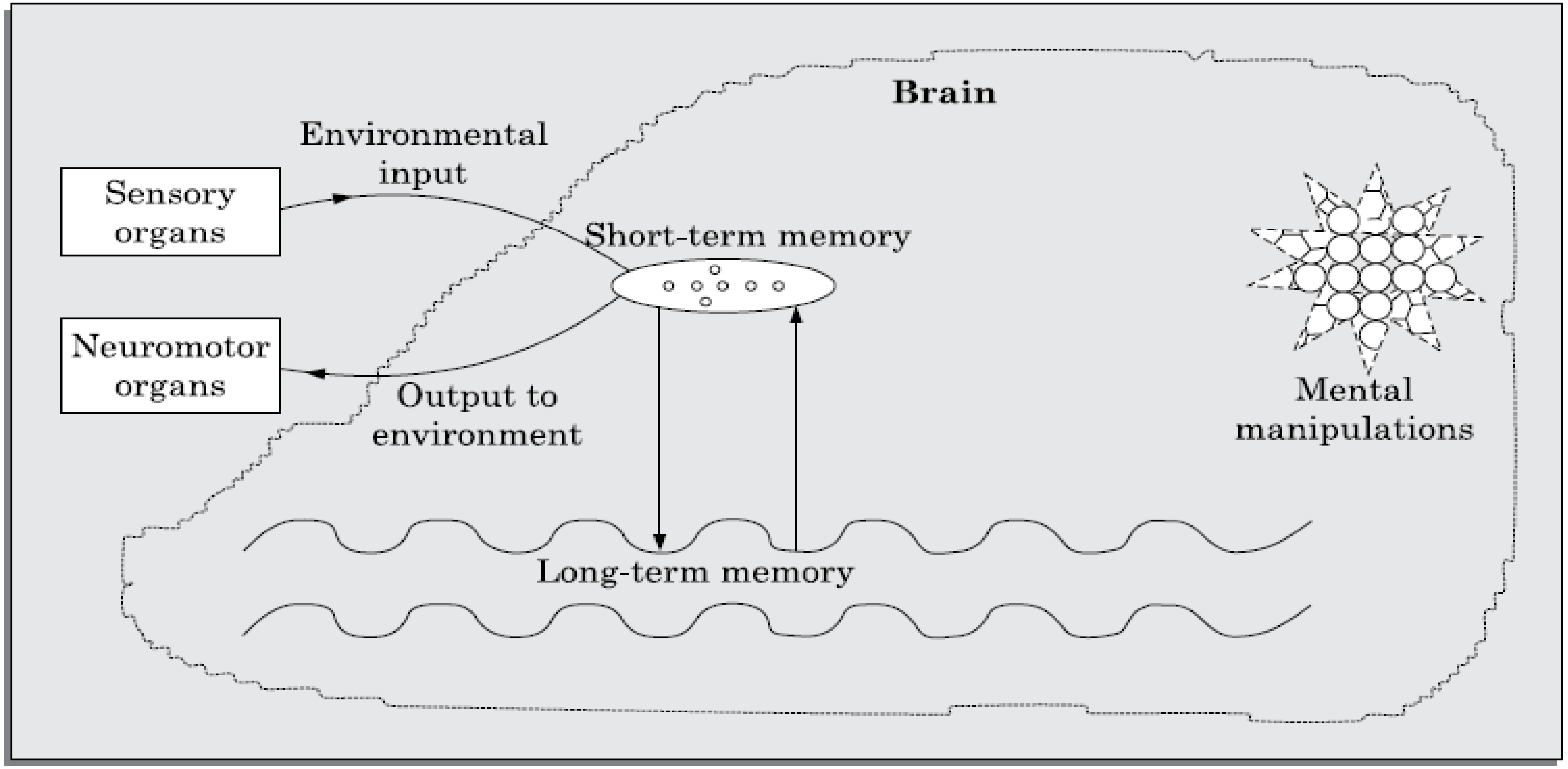


FIGURE 1.6 Human cognition mechanism model.

Human Memory Model (Miller, 1956)

Human memory is divided into:

1. Short-Term Memory (STM)

- Stores info for a **few seconds to minutes**.
- Capacity: Around **7 ± 2 items**.
- Participates in **all thinking and decision-making**.
- New info can **push out** existing data.
- Acts like a **computer cache** — quick but small.

Example: You can remember a phone number briefly while dialing, but forget it after an hour.

2. Long-Term Memory (LTM)

- Has **no fixed upper limit** — stores info for **years**.
- Info gets stored here via:
  - **Repetition** (refreshing)
  - **Linking** with existing knowledge

Chunking: Efficient Memory Use

- A **chunk** is a group of related items stored as **one unit**.
- Helps **compress information** to fit into short-term memory.
- Binary 110010101001 → Octal 6251 (makes it easier to remember)

"Magical Number 7"

- Most people can handle **7 or fewer items** comfortably.
- If a program has too many variables or components, it **exceeds** this mental capacity.
- Leads to **confusion, errors, and more effort**.

Why Exploratory Style Fails for Large Problems

- As program size grows, the number of things a developer must **track** exceeds the STM limit.
- This causes:
  - Confusion
  - Rework
  - **Exponential growth** in time and effort
- Eventually, projects become **unmanageable**.

How Software Engineering Helps

- Uses structured techniques like:
    - **Abstraction**: Focus on one aspect at a time
    - **Decomposition**: Break big problems into smaller, manageable parts
- These techniques are **designed to align with human memory limits**, keeping effort almost **linear** with program size (thin line in Fig 1.5).

- **Perceived complexity** grows exponentially with problem size due to **human cognitive limits**, especially the **short-term memory**.
- This explains why **exploratory programming** breaks down for large tasks.
- **Software engineering** principles are necessary to overcome these limitations and enable the development of large, reliable systems.

1.3.2 Principles Deployed by Software Engineering to Overcome Human Cognitive Limitations

- Human cognitive limitations—especially the limited **short-term memory** (7 ± 2 items)—make it difficult for programmers to handle large, complex software systems.
- To address this, **software engineering** uses two core principles:

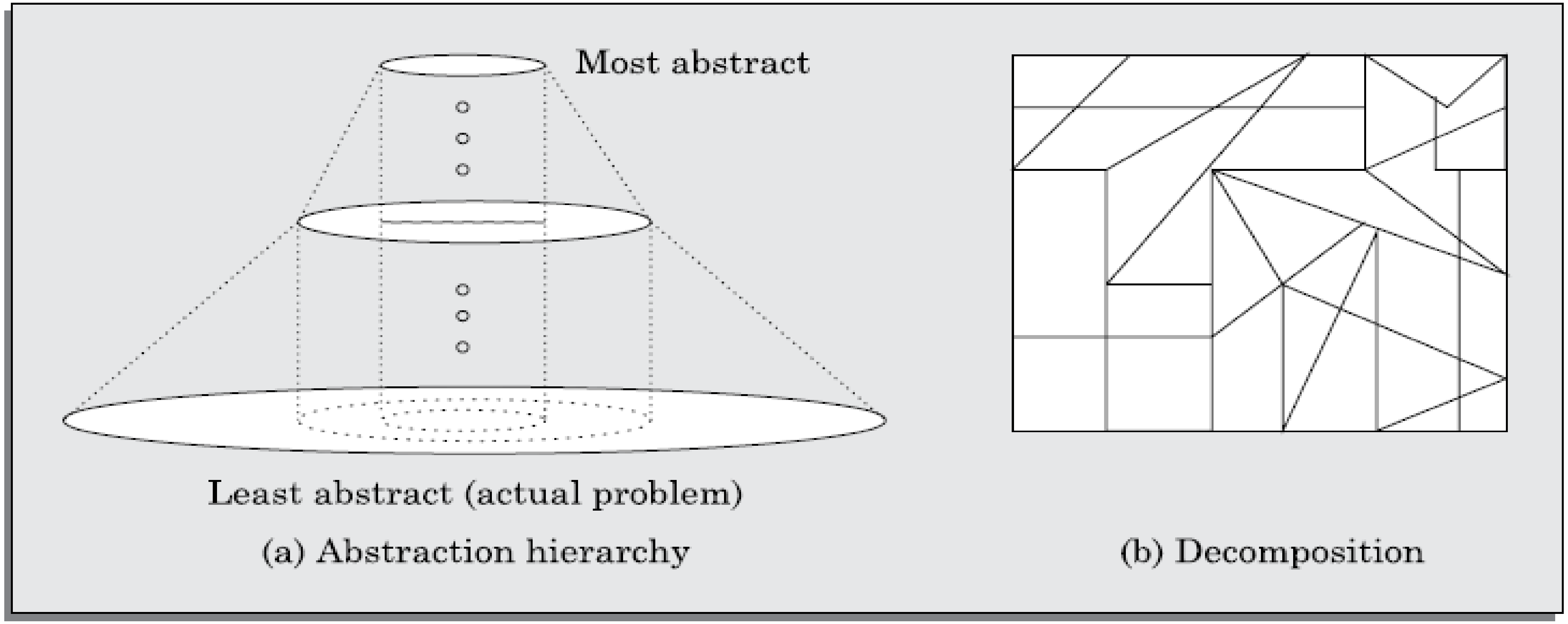


FIGURE 1.7 Schematic representation.

1. Abstraction

- **Definition**: Abstraction means **simplifying a complex problem** by focusing on only one or two relevant aspects and **ignoring the rest**. This is also called **modelling**.

Examples:

- Understanding a **country** through maps (political or physical) rather than visiting every place.
- Studying **living organisms** using an abstraction hierarchy:
  - Top level: Plants, animals, fungi
  - Lower levels: More detailed classifications (species, etc.)

Hierarchy of Abstractions:

- Simple problems → one level of abstraction is enough.
- Complex problems → need **multiple levels of abstraction**, where each level introduces only a few new concepts (within STM limits).
- 👉 **Purpose**: Break down the problem into digestible chunks to fit within human memory constraints.

2. Decomposition (Divide and Conquer)

- **Definition**:
  Decomposition means **breaking a large problem into smaller, independent parts**, each of which can be solved separately.
- **Analogy**:
  Trying to break a tied bunch of sticks is hard. But breaking them **one by one** is easy.
- **Important Condition**:
  The smaller parts must be **independent**. If solving one requires understanding others, decomposition fails.
- **Example**:
  A well-organized **book**:
  - Chapters → Sections → Subsections
  - Each part is focused and easier to understand independently.

- 👉 **Purpose**: Reduce perceived complexity by tackling small, self-contained pieces of the problem.

Why Study Software Engineering?

By learning software engineering, you gain:

1. **Ability to work on large projects in teams**, using industry-standard methods.
2. **Skill to manage complexity** through abstraction and decomposition.
3. Knowledge in:
    - ➢Requirements specification
    - ➢User interface design
    - ➢Testing and quality assurance
    - ➢Project management
    - ➢Maintenance

➢ 🔹 Even for **small programs**, using software engineering principles leads to **higher productivity** and **better quality**.

1.4 Emergence of Software Engineering

Software engineering evolved through **innovations** and **experiences** accumulated over decades. Below is a summary of key milestones:

1.4.1 Early Computer Programming

- Early programs were written in **assembly language**.
- Programs were **small** (a few hundred lines), using **build-and-fix** style.
- Programmers coded directly after hearing the problem, without design or plan.

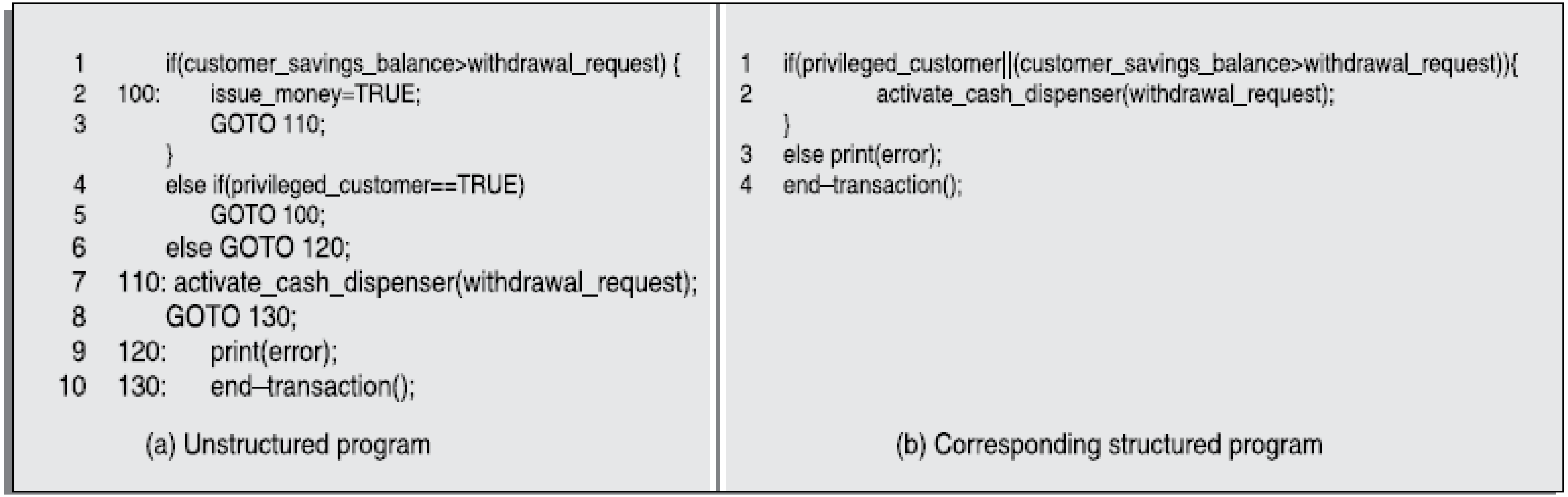
1.4.2 High-Level Language Programming

- Introduction of **semiconductor technology** (1960s) made computers faster.
- High-level languages like **FORTRAN, ALGOL, COBOL** were introduced.
  - Allowed writing **larger programs** easily.
  - Abstracted hardware details.
- But programmers still used **exploratory (code-and-fix)** approach.

1.4.3 Control Flow-Based Design

- Programs became larger and harder to understand/maintain.
- Emphasis shifted to **control flow structure** using **flowcharts**.
- Well-structured flowcharts = easier to understand/debug.

Figure 1.9 illustrates two alternate ways of writing program code for the same problem.



```
       if(customer_savings_balance>withdrawal_request) {
100:       issue_money=TRUE;
           GOTO 110;
       }
       else if(privileged_customer==TRUE)
           GOTO 100;
       else GOTO 120;
110: activate_cash_dispenser(withdrawal_request);
       GOTO 130;
120:       print(error);
130:       end–transaction();
```

(a) Unstructured program

```
if(privileged_customer||(customer_savings_balance>withdrawal_request)){
          activate_cash_dispenser(withdrawal_request);
}
else print(error);
end–transaction();
```

(b) Corresponding structured program

FIGURE 1.9 An example of (a) Unstructured program (b) Corresponding structured program.

- The flow chart representations for the two program segments of Figure 1.9 are drawn in Figure 1.10. Observe that the control flow structure of the program segment in Figure 1.10(b) is much simpler than that of Figure 1.10(a). By examining the code, it can be seen that Figure 1.10(a) is much harder to understand as compared to Figure 1.10(b).

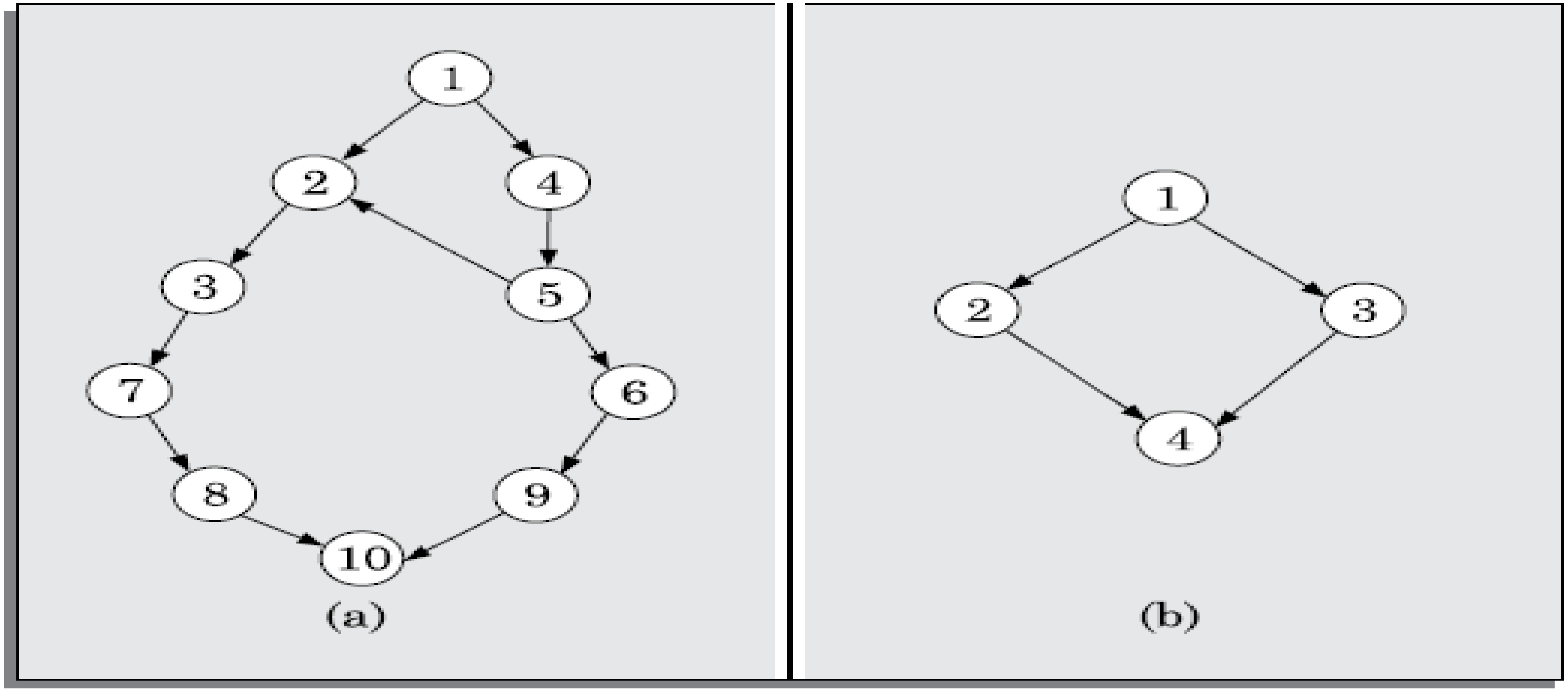


FIGURE 1.10 Control flow graphs of the programs of Figures 1.9(a) and (b).

Are GO TO statements the culprits?

- **GO TO statements** created messy control flows with too many execution paths.
  - Dijkstra’s paper *“GO TO Statements Considered Harmful”* (1968) led to awareness.

To understand his argument, examine Figure1.11 which shows the flow chart representation of a program in which the programmer has used rather too many GO TO statements. GO TO statements alter the flow of control arbitrarily, resulting in too many paths.

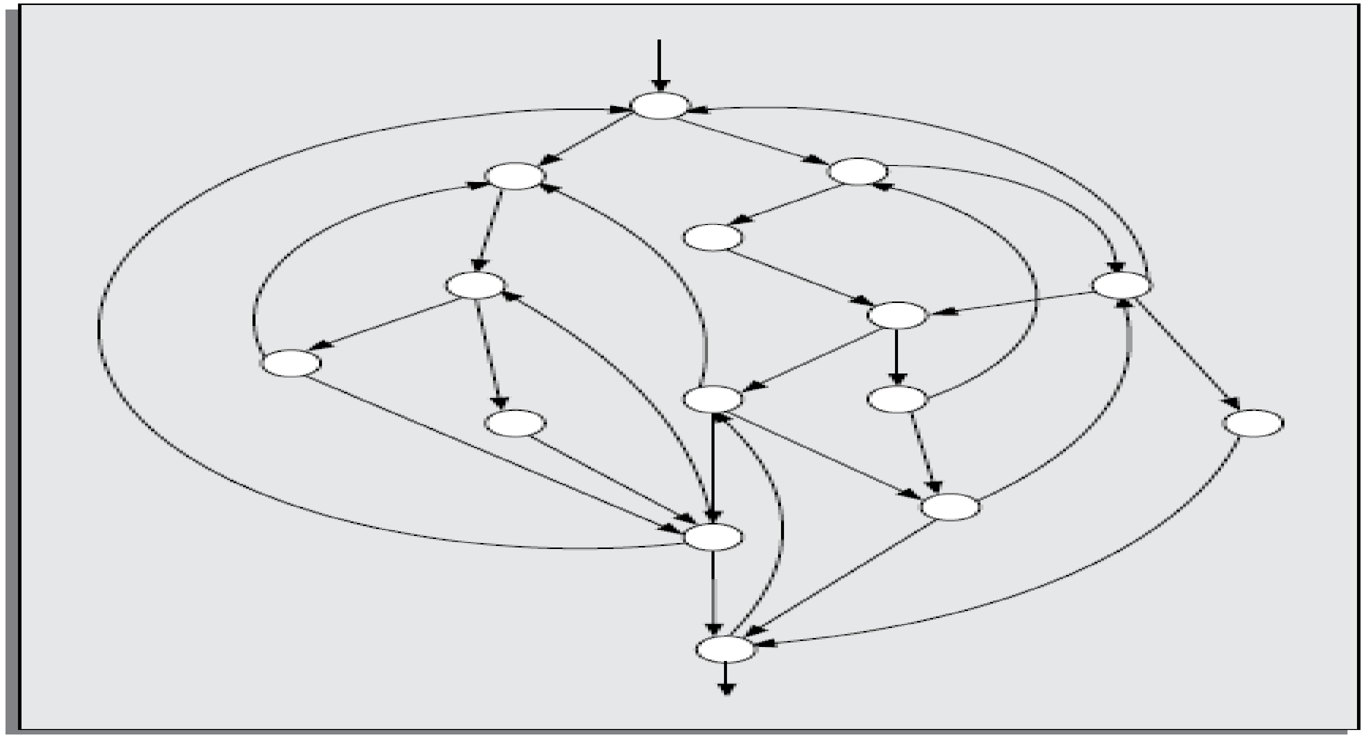


FIGURE 1.11 CFG of a program having too many GO TO statements.

Structured programming—a logical extension

Structured programming emerged:

- Uses only three constructs: **sequence**, **selection**, and **iteration**.
- Encourages **modularity**.
- Reduces errors, improves readability and maintainability.
- Supported by languages like **PASCAL, MODULA, C**.

1.4.4 Data Structure-oriented Design

- With **IC technology**, more complex programs were needed (tens of thousands of lines).
- Developers realized that **data structure** design is more important than control flow.
- Techniques like:
    - **Jackson's Structured Programming (JSP)**: derive control flow from data structure.
    - **Warnier-Orr Methodology**.
- These methods are now outdated and replaced by newer techniques.

1.4.5 Data Flow-Oriented Design

- Introduced with **VLSI** and faster computers.
- Emphasized on **functions** (processes) and **data exchange**.
- **Data Flow Diagrams (DFDs)** used to model systems:
    - Easy to understand and apply.
    - Widely used for **procedural design**.
- Example: **DFD of a car assembly plant** — shows processes and data stores clearly.
- Module – 5 discusses how to use DFDs for software design.

DFDs: A crucial program representation for procedural program design

- DFD has proven to be a generic technique which is being used to model all types of systems, and not just software systems. For example, Figure 1.12 shows the data-flow representation of an automated car assembly plant. If you have never visited an automated car assembly plant, a brief description of an automated car assembly plant would be necessary.

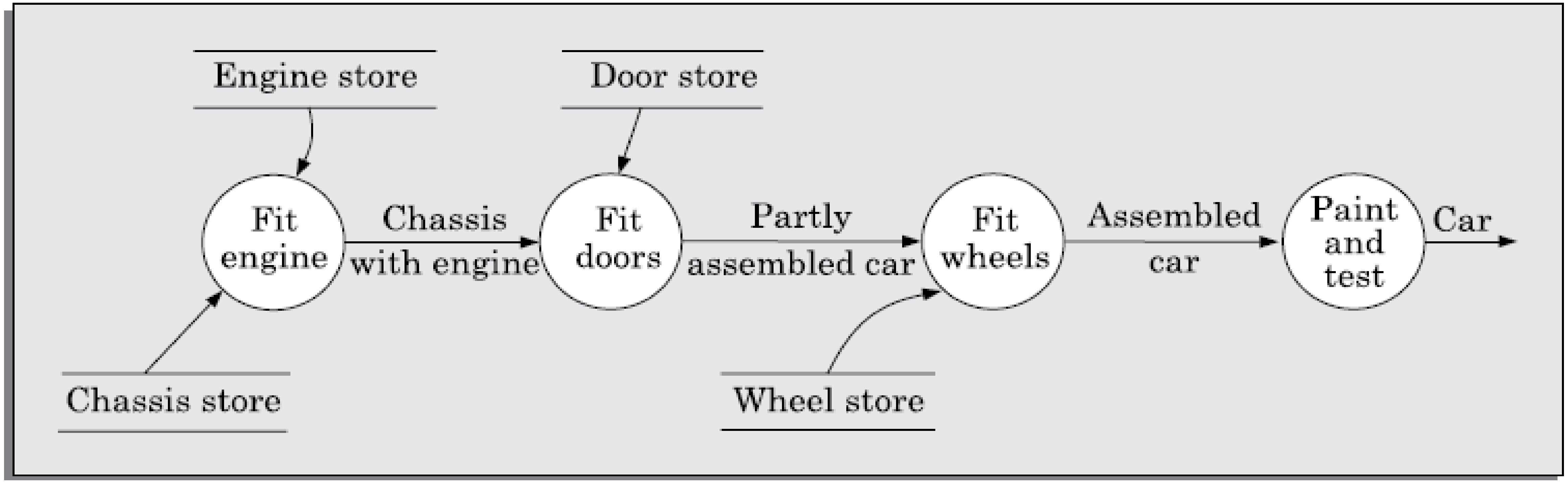


FIGURE 1.12 Data flow model of a car assembly plant.

<u>1.4.6 Object-Oriented Design (OOD)</u>

- Evolved in late 1970s.
- Models **real-world objects** (e.g., employee, payroll).
- Focuses on:
  - **Encapsulation** (data hiding)
  - **Inheritance**, **composition**, **association**
- Widely adopted due to:
  - Simplicity, reusability, maintainability, lower cost/time.

1.4.7 What Next?

Each decade introduced **revolutionary ideas** for more sophisticated and larger software.

- We pictorially summaries this evolution of the software design techniques in Figure 1.13. It can be observed that in almost every passing decade, revolutionary ideas were put forward to design larger and more sophisticated programs, and at the same time the quality of the design solutions improved.

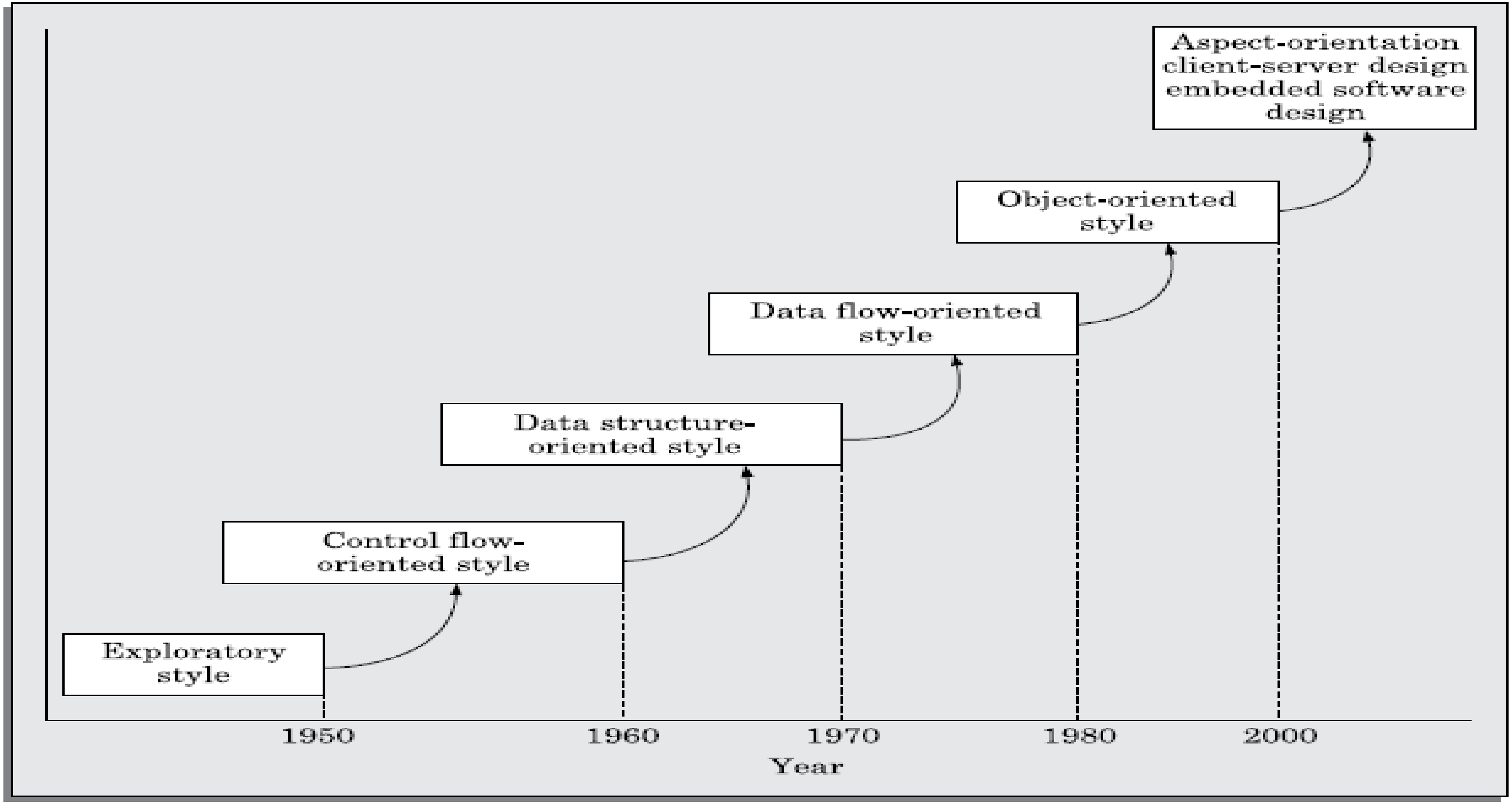

Aspect-orientation client-server design embedded software design
Object-oriented style
Data flow-oriented style
Data structure-oriented style
Control flow-oriented style
Exploratory style
1950
1960
1970
1980
2000
Year

FIGURE 1.13 Evolution of software design techniques.

- Current challenges:
  - Larger program sizes.
  - **Web-based** and **client-server** systems.
  - Rapid growth in **embedded systems**.

- Emerging paradigms:
  - **Aspect-Oriented Programming**
  - **Client-server design**
  - **Embedded software design**
  - **Service-oriented architecture** (driven by cloud/web apps)
  - DevOps and Continuous Architecture, vent-Driven and Reactive Architectures
  - AI-driven & ML-centric Architectures

1.4.8 Other Developments

Besides design techniques, other software engineering advancements include:

- **Life cycle models**
- **Specification techniques**
- **Testing and debugging**
- **Project and quality management**
- **Software metrics**
- **CASE tools (Computer-Aided Software Engineering)**

- These developments have accelerated the **growth of software engineering as a formal discipline**.

1.5 Notable Changes in Software Development Practices

This section compares **exploratory programming** (old approach) with **modern software engineering** practices and highlights the key differences:

1. Error Handling: Prevention vs. Correction

- **Exploratory style**: Errors are fixed *after* the software is built.
- **Modern approach**: Focus is on **preventing errors** and detecting them **early**, ideally in the **same phase** they occur (e.g., design errors fixed in design phase).

2. Coding Is Just One Part

- **Exploratory**: Coding was seen as the entire development process.
- **Modern**: Coding is only **one step**; other tasks like **design, testing, and specification** often require **more time and effort**.

3. Focus on Requirements Specification

- Today, **clear and accurate requirements** are gathered before development begins.
- This avoids **rework**, reduces cost, and ensures **customer satisfaction**.

4. Dedicated Design Phase

- Modern development includes a **structured design phase** using standard design techniques to produce **clear and complete models**.

5. Periodic Reviews (Phase Containment of Errors)

- Reviews are conducted at **every stage** to catch and fix errors **early**.
- This is called **phase containment of errors**, a key principle discussed in the module – 2.

<u>6. Systematic Testing</u>

- Testing is now a **well-defined** process.
- Test cases are designed from the **requirements stage** onward.

<u>7. Improved Documentation and Visibility</u>

- ➢ Earlier, documentation was **poor or missing**.
- ➢ Now, quality documents are created at every stage, helping with:
  - ➢ **Fault diagnosis**
  - ➢ **Maintenance**
  - ➢ **Project management** (discussed in Chapter 3)

8. Thorough Project Planning

- Modern projects are **carefully planned** to avoid delays and resource issues.
- Planning includes:
  - **Estimations**
  - **Scheduling**
  - **Tracking**
  - Use of tools for **cost estimation**, **configuration management**, etc.

9. Use of Metrics

- **Quantitative measurements** (metrics) are now used to:
  - Monitor product quality
  - Manage projects
  - Assure software quality

❖ Modern software development has shifted from an **ad hoc, fix-it-later** mindset to a **systematic, well-planned, and error-preventive** engineering approach. This leads to better **quality, maintainability, and customer satisfaction**.

1.6 Computer Systems Engineering

Computer Systems Engineering involves developing both **hardware and software** together, especially when the system runs on **specialized hardware**, not on general-purpose computers like desktops or servers.

Examples of Systems with Custom Hardware

- **Robots**
- **Factory automation systems**
- **Mobile phones**

These systems require:

- A **custom processor**
- Specialized components (e.g., mic, speaker)
- Software written specifically for that hardware

What is Systems Engineering?

- A broader field that includes **software engineering** as a part.
- Involves designing the **entire system**, including both:
  - **Hardware**
  - **Software**

Hardware-Software Partitioning

- A key stage in systems engineering.
- Decides **which parts of the system** should be done in hardware and which in software.

Criteria	**Hardware**	**Software**
Speed	Faster	Slower
Flexibility	Hard to change	Easy to modify
Complexity	Hard to implement complex logic	Easier to implement complex logic
Cost, space, power	Higher	Lower

Concurrent Development

- ➢ **Hardware and software** are developed **simultaneously** (in parallel branches).
- ➢ Challenge: You can't test the software easily because the actual hardware may not be ready.

Solution: Use Simulators

- ➢ **Simulators** are created to **mimic the hardware** during software development and testing.
- ➢ Once both hardware and software are ready, they are **integrated and tested together**.

Project Management

- Required **throughout the development** of both hardware and software to coordinate tasks, track progress, and manage risks.

❖ **Computer Systems Engineering** is the discipline of building systems that require both software and specialized hardware. It includes careful **hardware-software partitioning**, **parallel development**, and **simulator-based testing**, making it broader and more complex than software engineering alone.

Software Engineering (Code: CSE3122)

B. Tech CSE 5th Sem.

(School of Computer Engineering)

Dr. Kranthi Kumar Lella
Associate Professor,
School of Computer Engineering,
Manipal Institute of Technology,
MAHE, Manipal – 576104.

SOFTWARE ENGINEERING

Module – 2 (SOFTWARE LIFE CYCLE MODELS)

CONTENTS

2.1 A FEW BASIC CONCEPTS

Software Life Cycle

The **software life cycle** is similar to a biological life cycle—it describes the **stages** a software goes through:

1. **Inception**: A customer expresses the need for software, but requirements are unclear.
2. **Development**: Software evolves through identifiable **phases** (e.g., design, coding) based on developer actions.
3. **Operation (Maintenance)**: After release, users use the software and request **bug fixes and improvements**. This phase is usually the longest.
4. **Retirement**: The software is discarded when it's no longer useful due to **changing needs or technology**.

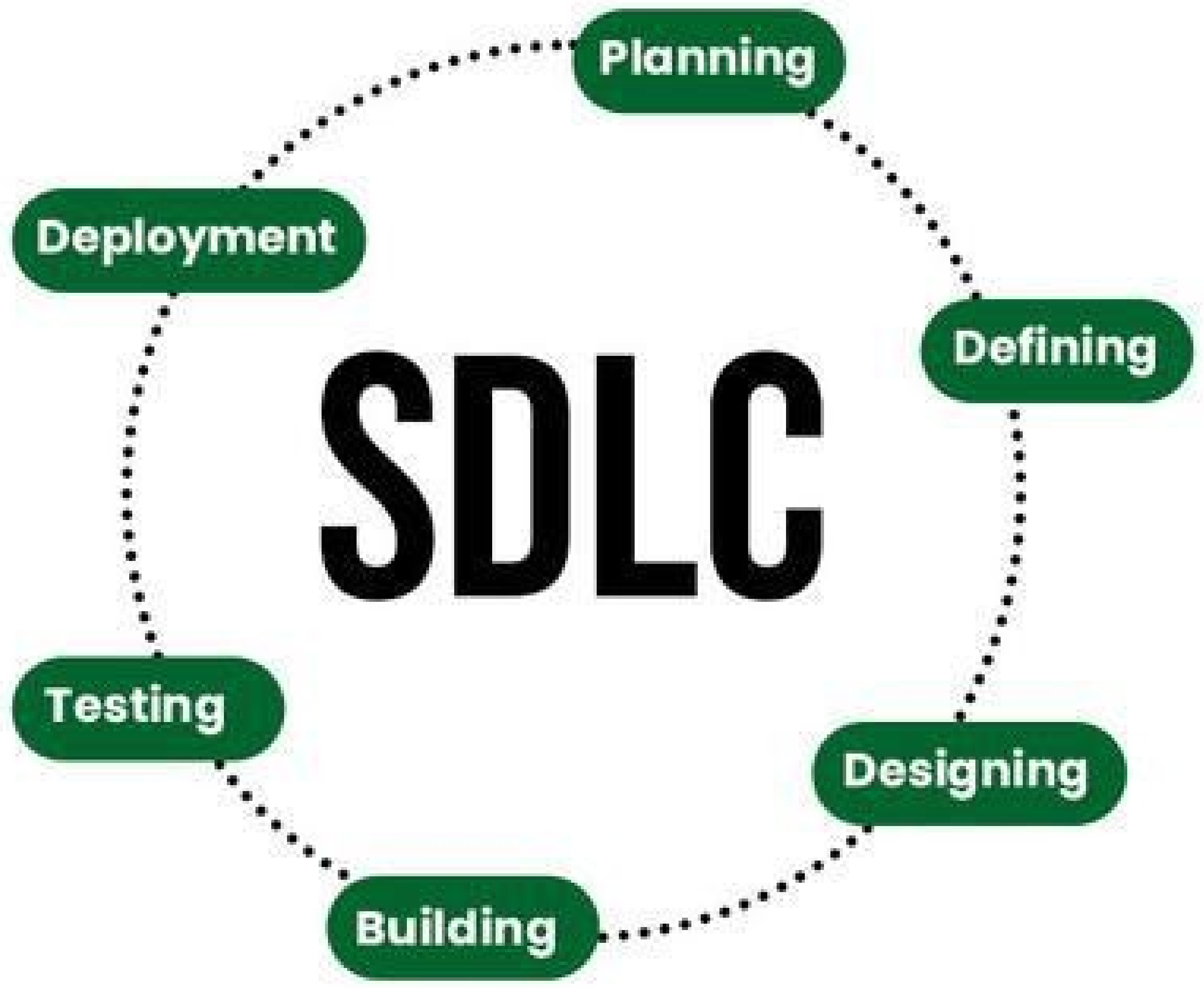
Software Development Life Cycle (SDLC) Model

An **SDLC model** (also called **software process model**) outlines **well-defined activities** needed to transition from one stage to another in the software life cycle.

- Example: Moving from requirements to design involves requirement gathering, analysis, documentation, and customer review.
- SDLC is often shown as a **graphical model** with phases and transitions, along with **textual descriptions** of activities.

Process vs Methodology

- A **process** has a **broader scope**; it covers all or major software activities (e.g., design process, testing process).
- A **methodology** is **narrower**, describing **specific steps** to perform an activity (e.g., a method for high-level design).
- A process may recommend **methodologies** for each phase.


SDLC
Planning
Defining
Designing
Building
Testing
Deployment

Why Use a Development Process?

Using a well-defined development process:

- ➢Ensures **systematic and disciplined** development.
- ➢Is **crucial for team-based projects** (programming-in-the-large).
- ➢Avoids problems like poor coordination, unclear responsibilities, and integration issues.

➢ 🔸 Example: If team members develop independently without coordination, **conflicts and mismatches** will arise—leading to project failure.

➢In contrast, **programming-in-the-small** (e.g., student assignments) might succeed without a formal process.

Why Document the Development Process?

Documenting the process is important because:

1. It clearly defines **activities**, their **sequence**, and **methodologies**, reducing **confusion and misinterpretation**.
2. It signals management's **seriousness**, encouraging developers to follow it strictly.
3. It allows for **tailoring** of the process to suit specific project needs.
4. It is **required by quality standards** like **ISO 9000** and **SEI CMM** for certification and client trust.
5. Helps identify **inconsistencies**, **omissions**, and improves **training** for new employees.

Phase Entry and Exit Criteria

A good SDLC defines:

- **Entry criteria**: Conditions that must be met to **start** a phase.
- **Exit criteria**: Conditions to **complete** a phase (e.g., reviewed and approved SRS to exit requirements phase).

- Without clear criteria:
  - Developers may **claim early completion**, causing confusion.
  - Project managers can’t accurately track progress, leading to the **“99% complete syndrome”**—developers claim near-completion while work is far from done.

- Understanding and following a well-documented **SDLC model** is essential for:
  - Delivering **high-quality software**
  - Avoiding **project failures**
  - Managing **team-based development effectively**

2.2 Waterfall Model and Its Extensions

- The **Waterfall Model** was widely used in the **1970s** and still finds use today.
- It is the **most useful** model for team-based software development.
- All other life cycle models are **extensions** of the **classical waterfall model**, tailored for specific project needs.

2.2.1 Classical Waterfall Model

- **Nature of the Model**
    - The **classical waterfall model** is **simple** and follows a **linear** sequence of phases.
    - It is idealistic because it **assumes no mistakes** during development, but in reality, **errors are discovered in later stages**.
    - It has **no provision to revisit** or correct earlier phases, making it **impractical for non-trivial projects**.
    - Still, it is important to study as **a foundational model**, and it is **implicitly used for documentation purposes**.

Phases of the Classical Waterfall Model

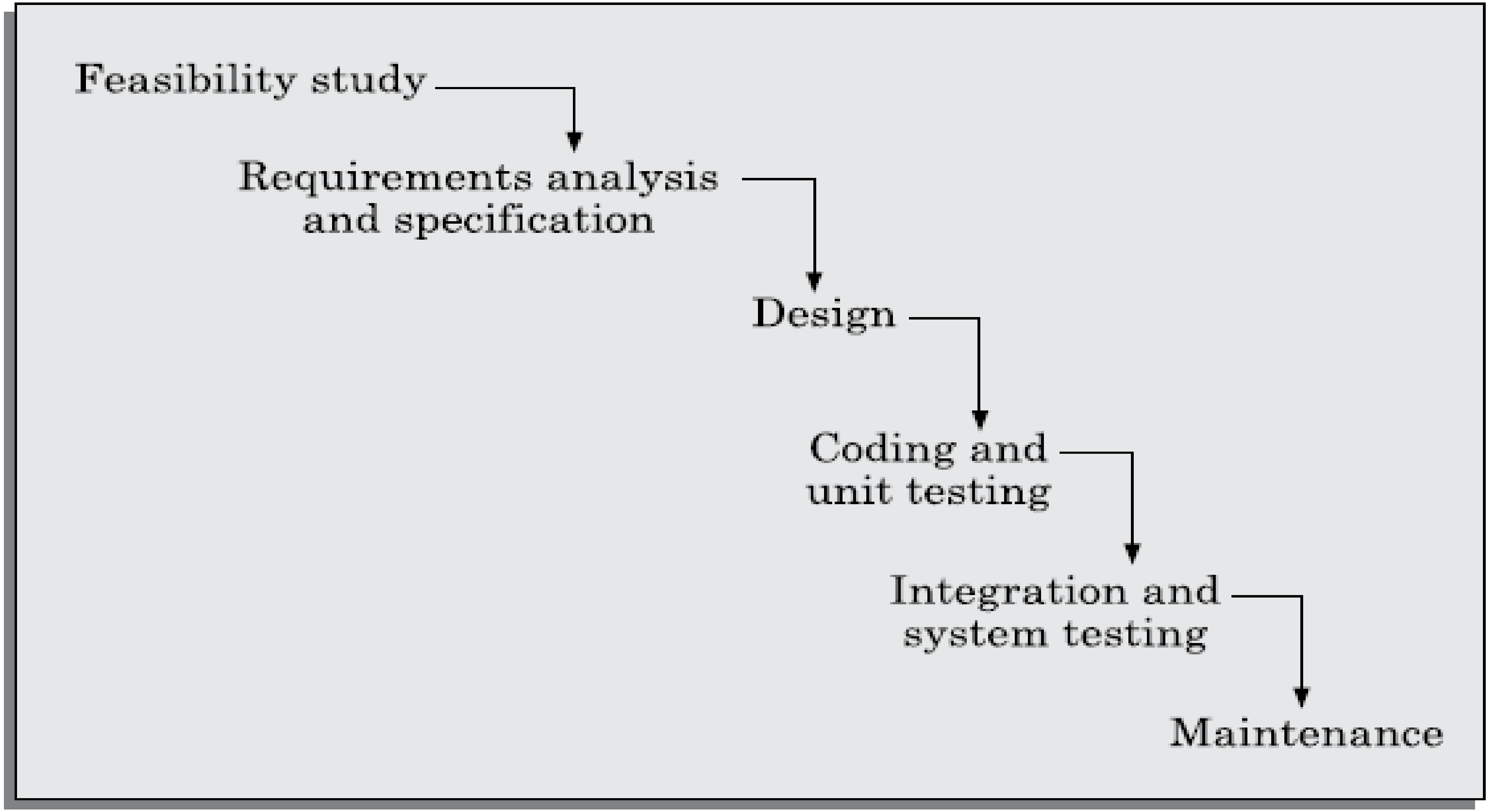

Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

FIGURE 2.1 Classical waterfall model.

Phases of the Classical Waterfall Model (cntd..)

The model consists of **six main phases** (see Fig. 2.1):

1. **Feasibility Study**
2. **Requirements Analysis and Specification**
3. **Design**
4. **Coding and Unit Testing**
5. **Integration and System Testing**
6. **Maintenance**

- ➢ The first five are **development phases**; after these, the software is delivered to the customer.
- ➢ Once delivered, the **operation or maintenance phase** begins, where bug fixes and updates are made.

- ➢ Note: **Project management** is a critical activity throughout all phases but isn't treated as a separate phase.

Effort Distribution (Fig. 2.2)

- The **maintenance phase** consumes about **60%** of total project effort.
- Within development phases, **integration and system testing** requires the **most effort**.

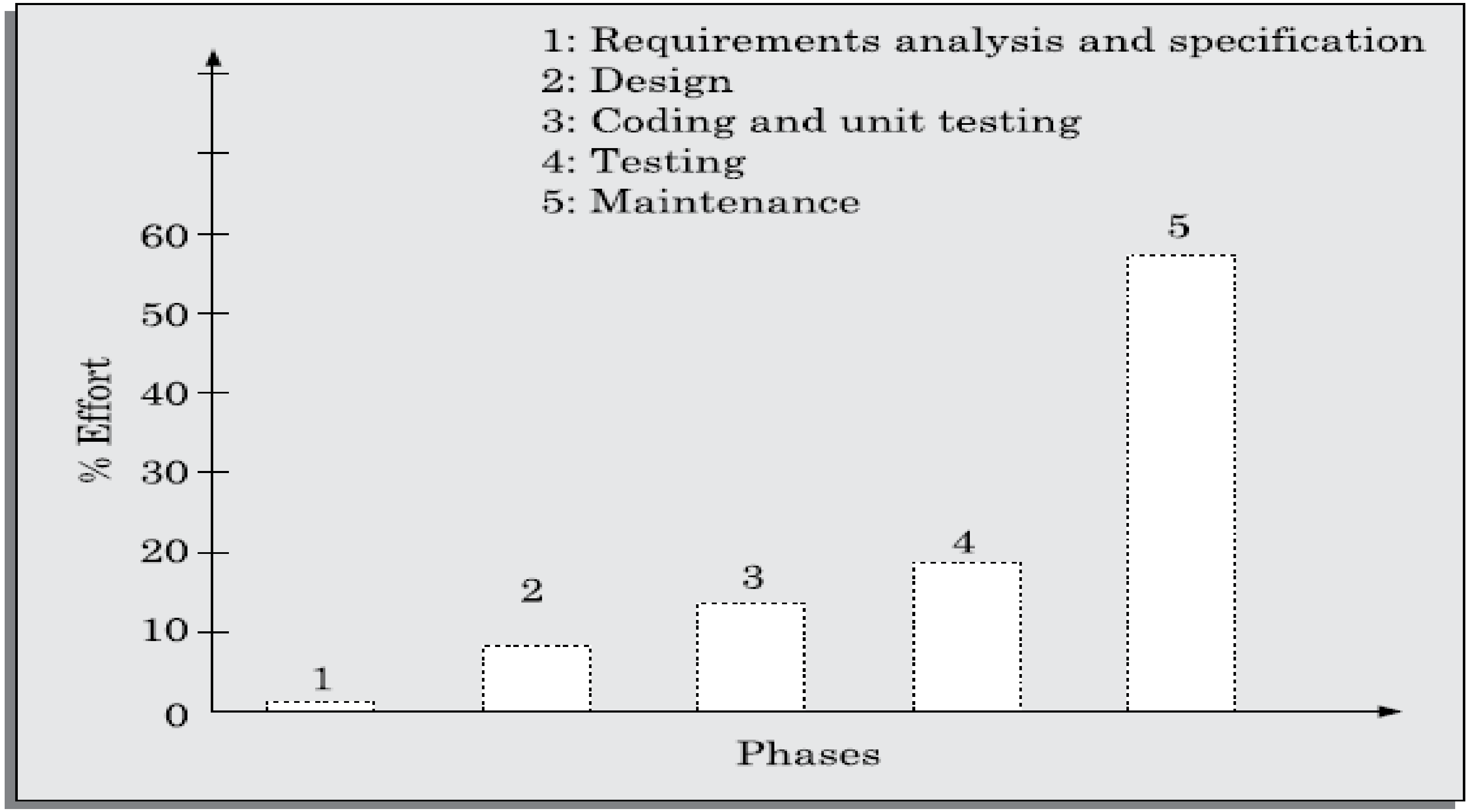


FIGURE 2.2 Relative effort distribution among different phases of a typical product.

Feasibility Study (First Phase)

The goal is to determine if the project is **technically and financially feasible**.

🔹 **Key Activities:**

1. **Collect basic information**: Inputs, processing needs, expected outputs, and constraints.
2. **Understand the customer's needs**:
   - Only focus on **key requirements**, not UI layouts, specific algorithms, or database designs.
3. **Formulate possible solution strategies**:
   - Example: Client-server model vs standalone application.
4. **Evaluate solution strategies**:
   - Estimate **resources, cost, and time**.
   - Compare solutions to choose the best one.
   - If **no solution is feasible**, the project is **abandoned**.

🔸 Outcome: A **high-level decision** on whether to proceed with the project, and **which solution strategy** to follow.

Case Study 2.1 – Feasibility Study Example

Company: Galaxy Mining Company Ltd. (GMC Ltd.)
Need: A system to manage a **Special Provident Fund (SPF)** for miners, separate from the standard PF, to allow **quick compensation**.

📌 Problem:

- GMC Ltd. has **50 mines** across **8 Indian states**.
- Monthly SPF deductions from miners need to be sent to a **Central SPF Commissioner (CSPFC)**.
- Manual record-keeping is time-consuming and delays settlements.

💡 Solution Request:

- GMC Ltd. asked **Adventure Software Inc.** to build the SPF software.
- Budget: **₹1 million**.

🔍 Feasibility Study:

- **Two solution strategies**:
  - **Centralized database** via satellite link.
    - Fails completely if communication breaks.
  - **Local databases** at mine sites + **periodic updates** to central DB via dial-up.
    - **More fault-tolerant** and affordable.
    - Local systems work even if communication fails.
- The **second solution** is technically and financially feasible and was **approved** by GMC Ltd.

🔷 Phases of the Classical Waterfall Model (Fig. 2.1)

1. Feasibility Study

- ➢Check **technical and financial feasibility**.
- ➢Identify problem scope, high-level strategies, cost, time, and resource estimates.
- ➢Outcome: Decide **whether to proceed** and **which strategy** to adopt.

2. Requirements Analysis and Specification

Objective: Understand and document customer requirements.

- 🔹 **Requirements Gathering & Analysis**:
  - Collect and analyze customer needs.
  - Remove **inconsistencies** and **incompleteness**.

- 🔹 **Requirements Specification**:
  - Create the **SRS (Software Requirements Specification)** document.
  - Written in **end-user language**, acts as a **contract**.
  - Serves as a base for further development and testing.

3. Design

Goal: Translate SRS into software architecture.

- 🔷 **Procedural Design Approach**:
  - Based on **Data Flow Diagrams (DFD)**.
  - Two parts:
    - **High-Level Design (Architecture):** Break into modules.
    - **Detailed Design:** Design internals like algorithms and data structures.

- 🔷 **Object-Oriented Design (OOD)**:
  - Identify objects and relationships.
  - Easier maintenance and faster development.

4. Coding and Unit Testing

- Translate design into **source code**.
- Each module is **unit tested**:
    - Design test cases
    - Debug and verify correctness

5. Integration and System Testing

- Modules are **incrementally integrated**.
- Two levels of testing:
    - **Integration Testing:** Check module interfaces.
    - **System Testing:** Ensure full system meets **SRS**.
- **Types of system testing:**
    - **α-testing:** By development team
    - **β-testing:** By friendly customers
    - **Acceptance Testing:** By customer to approve software

6. Maintenance

- Maintenance requires **more effort** than development (approx. 60%).
- 🔹 **Types**:
  - **Corrective:** Fix bugs
  - **Perfective:** Improve performance/features
  - **Adaptive:** Port to new platforms or environments

🔷 Shortcomings of the Classical Waterfall Model

- **No Feedback Paths**:
  - Assumes **no mistakes** in any phase; no option to revise earlier phases.
- **Difficult to Handle Changes**:
  - Customer **change requests** can't be easily handled after requirement phase.
- **Late Testing**:
  - Integration/testing occurs **late**, making error correction **costly**.
- **No Overlapping of Phases**:
  - Sequential execution leads to **inefficient resource use**.
  - In practice, **phases overlap** (e.g., testing starts after SRS).

📌 Is the Waterfall Model Still Useful?

- Though **rarely used directly** for development, it is useful for:
- Understanding **foundation** of other models.
- **Documentation purposes**, as suggested by **Parnas** and **Hoare**.

❖ Like a mathematician presents a **clean proof**, even after multiple trials and errors, the final software documentation is written **as if** development followed the waterfall model exactly.

2.2.2 Iterative Waterfall Model

To overcome the **rigid limitations** of the classical waterfall model by allowing **feedback** and **error correction** during development.

Main Change:

Introduction of **feedback paths** from each phase to its **previous phases** (except feasibility phase).

📌 This means if an error is found in a later phase (e.g., testing), earlier phases (like design or coding) can be **revisited and corrected**.

📌 No feedback is allowed to **feasibility phase** because once a project is approved, it's typically not abandoned due to **legal/moral commitments**.

The feedback paths introduced by the iterative waterfall model are shown in Figure 2.3. The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase.

Please notice that in Figure 2.3 there is no feedback path to the feasibility stage. This is because once a team accepts to take up a project, it does not give up the project easily due to legal and moral reasons.

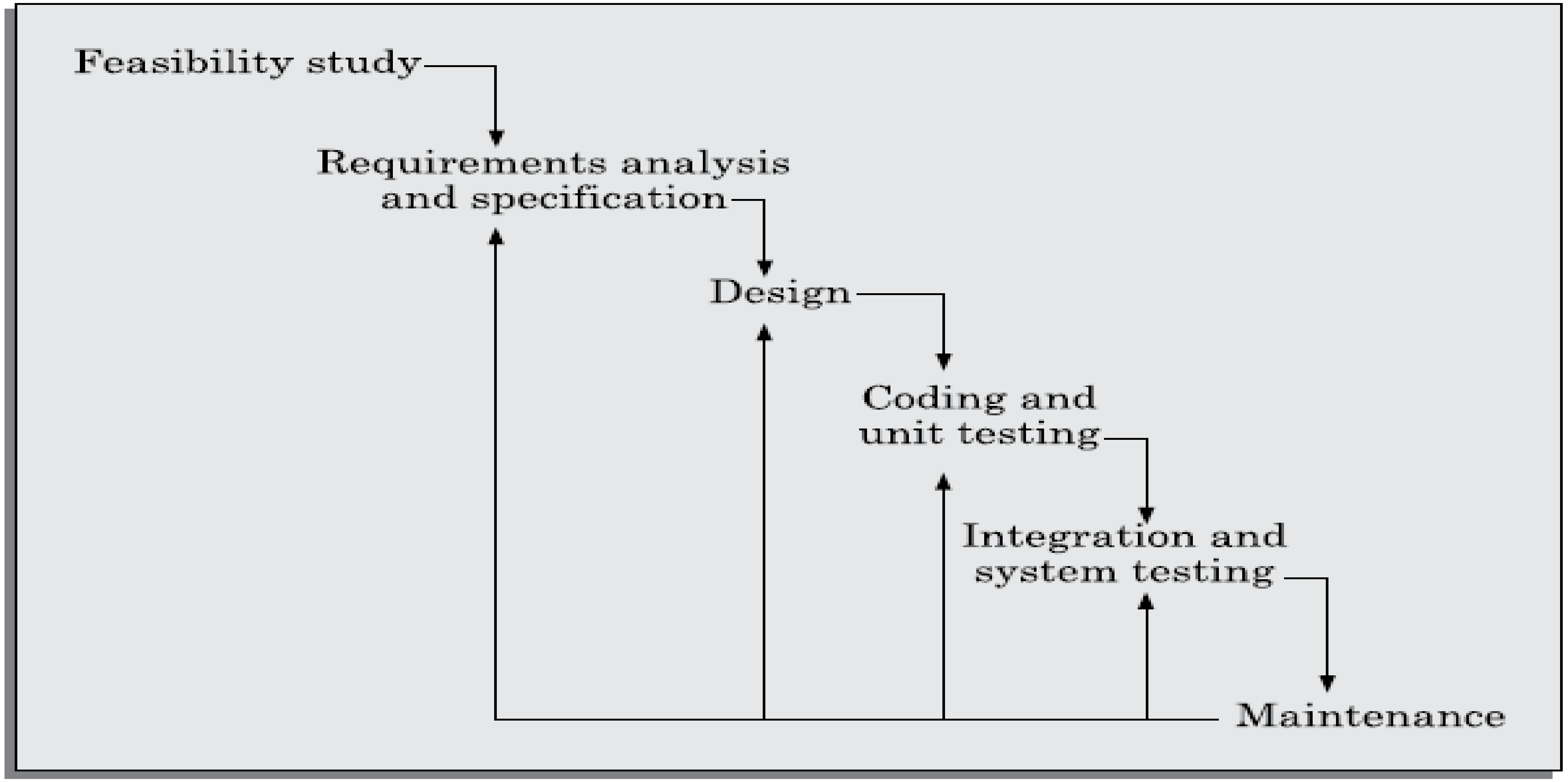

Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

FIGURE 2.3 Iterative waterfall model.

Error Detection Strategy – Phase Containment of Errors:

Detect errors in the same phase where they were introduced.

- ➢Early detection = lower cost and time.
- ➢Example: Fixing a design error **during design** is cheaper than fixing it during **testing**.
- ➢**Technique used**: Rigorous **review** of documents (like SRS, design docs, etc.).

Phase Overlap (Refer: Fig. 2.4)

- Even though the model shows a step-by-step sequence, in practice, **phases often overlap** due to:

🔹 **Late detection of errors** — forces rework of earlier phases.

🔹 **Workload differences** — team members who finish early move on to the next phase instead of waiting idly.

- If we consider such rework after a phase is complete, we can say that the activities pertaining to a phase do not end at the completion of the phase, but overlap with other phases as shown in Figure 2.4.

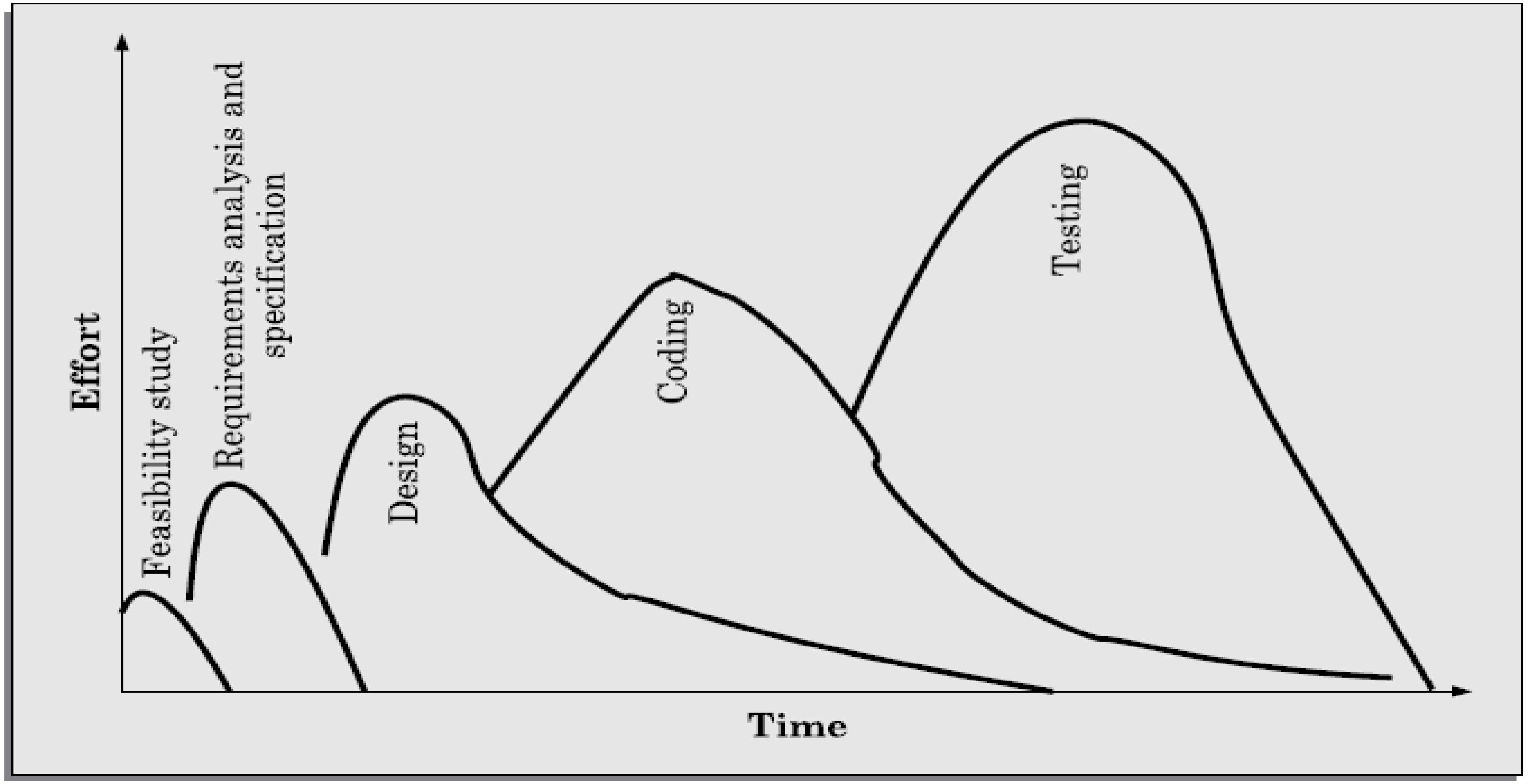


FIGURE 2.4 Distribution of effort for various phases in the iterative waterfall model.

<u>Shortcomings of Iterative Waterfall Model</u>

Even with feedback and overlap, the model has **major limitations**, especially in modern projects:

<u>1. Change Requests Not Supported</u>

- Requirements are **frozen** early.
- **Any change** needs major **replanning and rework**.
- But in real life, requirements **often change** as customers understand their needs better **during development**.

<u>2. No Incremental Delivery</u>

- Software is delivered **only after full development and testing**.
- If the project takes **months or years**, business needs might change, making the final product **irrelevant or outdated**.

3. Rigid Phase Sequence

- Team members may **idle** while waiting for others.
- Causes **resource wastage** and **inefficiency**.

4. Late Error Detection = High Cost

- Validation (testing) happens **after coding**.
- Bugs found late cause **expensive rework**.

5. Limited Customer Interaction

- Interaction only at **beginning** and **end**.
- No involvement during design or testing = software often **misaligned** with customer needs.

6. Heavy weight Model

Emphasizes **extensive documentation**, which:

- Consumes time
- Reduces agility
- May not directly add customer value

7. No Support for Risk or Reuse

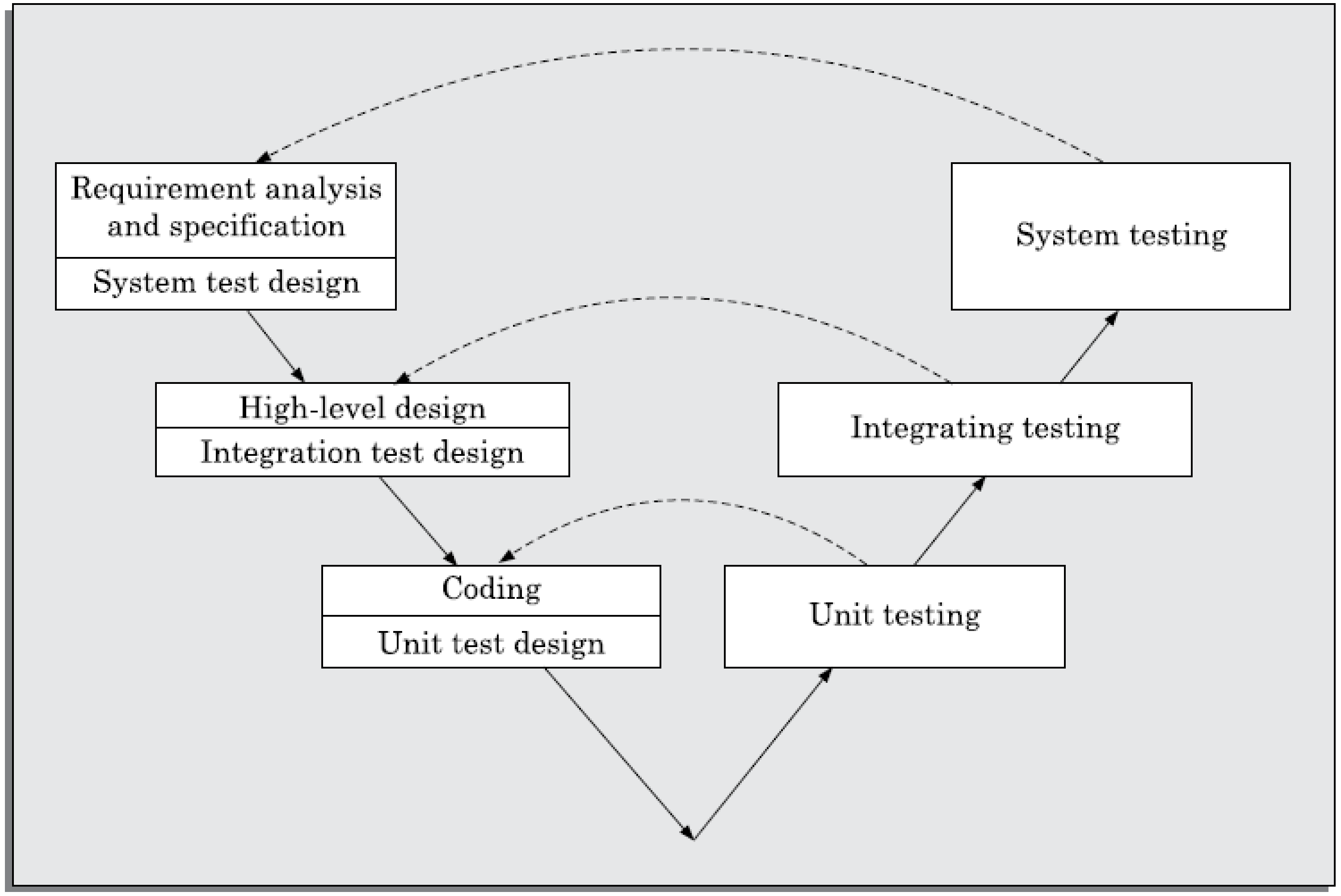
Not suited for:

- Projects with **uncertainty or risk**
- **Reuse-based development**, like using existing components or code libraries

- Iterative waterfall is **an improvement** over classical waterfall.
- However, it's still not suited for **modern, agile, fast-paced projects**.
- Hence, **Agile models** (discussed in Section 2.4) are preferred.

2.2.3 V-Model (Validation and Verification Model)

The **V-model** is a **variant of the waterfall model**, named after its **V-shaped structure** that visually represents the **development (left side)** and **validation (right side)** phases As shown in Figure 2.5 (V-model).



Key Features:

- **Verification and validation** are performed **throughout** the development life cycle.
- Suitable for **safety-critical systems** (e.g., medical, aviation, defense) where **high reliability** is essential.

Structure:

- **Left side (Development Phases)**:
  - Each phase (e.g., requirements, design, coding) includes:
    - Creation of the work product.
    - Designing corresponding **test plans and test cases**.
- **Right side (Validation Phases)**:
  - Each validation phase corresponds to a development phase.
  - Actual testing is performed:
    - 🔹 **Unit testing** → Validates **coding**
    - 🔹 **Integration testing** → Validates **design**
    - 🔹 **System testing** → Validates **requirements**
- ✅ This ensures **early detection** of defects and better product quality.

V-Model vs. Waterfall Model:

Feature	Waterfall Model	V-Model
Testing timing	Only in the **testing phase**	**Throughout** the life cycle
Parallel test planning	❌ No	✅ Yes (during development)
Team involvement	Testers join **late**	Testers involved **from start**
Error detection	**Late** (expensive to fix)	**Early** (less cost, better quality)

Advantages of V-Model:

1. **Faster development**: Because test planning happens early, the final testing phase is shorter.
2. **Better test quality**: Test cases are written early, with more care (less time pressure).
3. **Efficient manpower use**: Testers work **throughout** the project, not just at the end.
4. **Improved testing effectiveness**: Testers understand the system deeply as they are involved from the beginning.

Disadvantages of V-Model:

Since it's based on the **classical waterfall model**, it inherits its **limitations**, such as:

- **Strict Phase Sequencing**
- **Limited flexibility**
- **Poor support for changing requirements**
- **Heavy documentation**
- Not suitable for **iterative**, **agile**, or **reuse-based** projects

❖ The **V-model improves** upon the classical and iterative waterfall models by emphasizing **early test planning** and **parallel development–validation**. However, it is still **not suitable for dynamic or modern agile projects** due to its waterfall-based nature.

2.2.4 Prototyping Model

The **Prototyping Model** is a **popular life cycle model** and can be considered an **extension of the waterfall model**. It involves building a **working prototype** of the system **before** the actual software is developed.

A prototype is a basic, early version of the final software that has:

- **Limited functionality**
- **Low reliability**
- **Inefficient performance**

It is built **quickly using shortcuts**, such as:

- Dummy functions
- Table lookups instead of real computations
- This is also called **rapid prototyping**, often using tools like **4GLs** for GUI construction.

When to Use the Prototyping Model

This model is useful in three main situations:

- **Developing GUI (Graphical User Interface):**
  - Helps demonstrate **input formats**, **dialogs**, and **outputs** to users.
  - Users can give feedback by interacting with the prototype instead of imagining a UI.
- **When Technical Solutions Are Unclear:**
  - Used to **test uncertain technologies** (e.g., compiler development, response time of hardware).
  - Helps the team **learn and resolve technical risks** before real development.
- **When Perfection Isn't Possible Initially:**
  - As Brooks (1975) stated: *“Plan to throw away the first version.”*
  - Helps to refine requirements and design before developing an **efficient final product**.

Life Cycle Activities in Prototyping Model

As shown in Figure 2.6, the model consists of two main phases:

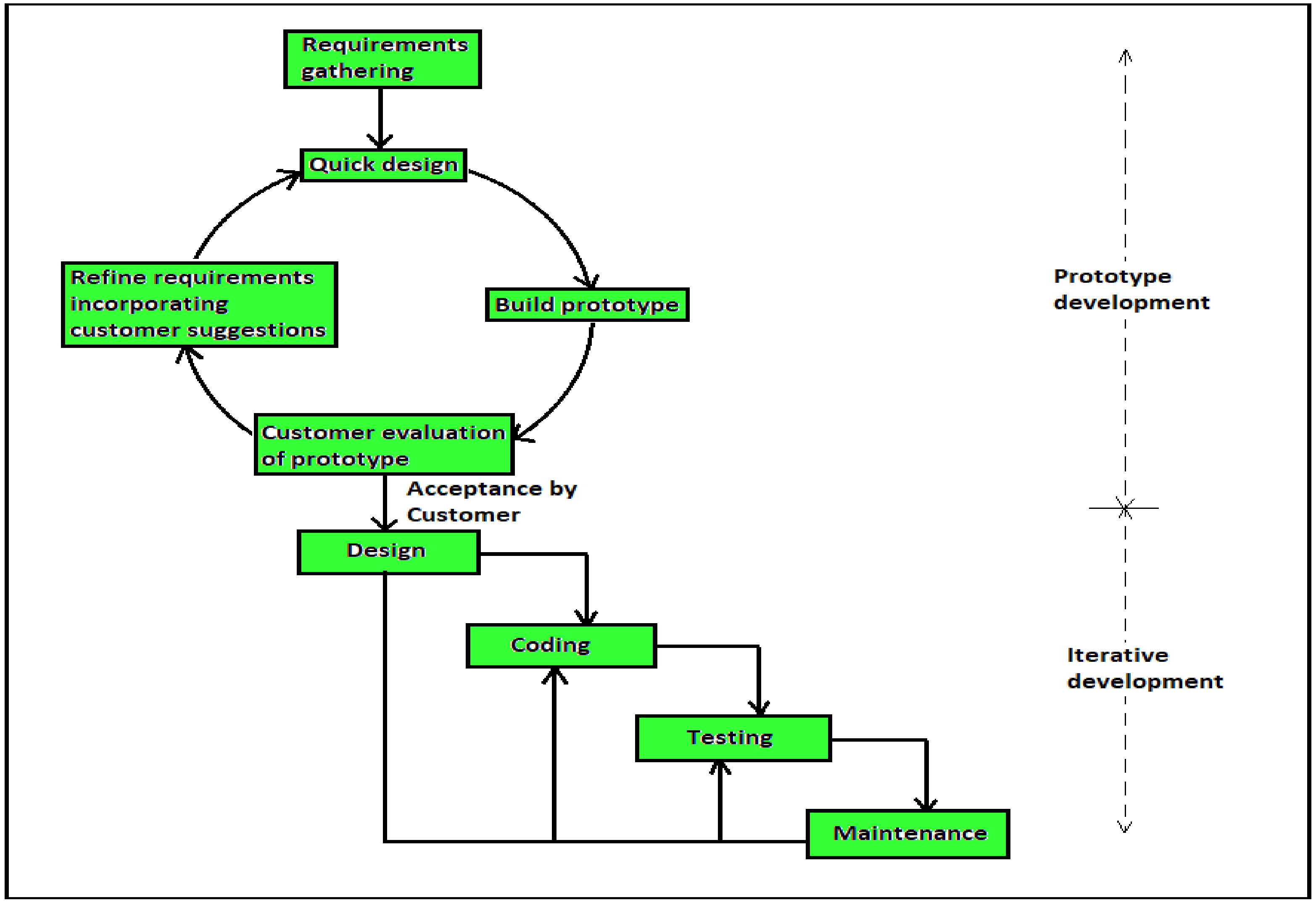


FIGURE 2.6 Prototyping model of software development.

1. Prototype Development:

- Start with **initial requirement gathering**
- Do a **quick design** and build the prototype
- Submit it to the **customer for feedback**
- Refine requirements and modify prototype **iteratively** until the customer **approves**

2. Iterative Waterfall Development:

➢ Once prototype is approved, use **iterative waterfall** model to develop the actual software

➢ Even if a prototype exists, an **SRS (Software Requirements Specification)** is needed for:

  ➢ **Traceability**

  ➢ **Verification**

  ➢ **Test planning**

➢ For GUI parts, SRS may not be needed, as the **approved prototype serves as animated specifications.**

➢ ⚠️ **Prototype code is usually discarded**, but the **experience gained** helps in real development.

Advantages of Prototyping Model

- Excellent for projects with:
  - **Unclear requirements**
  - **Technical uncertainties**
- Helps clarify requirements early
- Reduces **future changes** and **redesign costs**

Disadvantages of Prototyping Model

- **Increases cost** if used for:
  - Routine projects with no significant risks
- Only helps if **risks are known upfront**
- **Ineffective** for risks identified **later** during development (e.g., staff leaving midway)

- The **Prototyping Model** is best suited for projects involving **unclear requirements** or **technical risks**, especially **GUI development**. It helps refine requirements early but may add **unnecessary cost** for well-understood, routine projects.

2.2.5 Incremental Development Model

In the **incremental life cycle model**, software is developed and delivered in **increments**—small parts that gradually build up the full system.

- **First**, the overall requirements are broken into **smaller, manageable units** called *increments*.
- The **first increment** delivers a **basic working version** with **core features**.
- In **each iteration**, a **new increment** adds **additional features** or **refines existing ones**, until the **complete system** is built.

📌 **Figure 2.7** illustrates the concept of incrementally building a system.

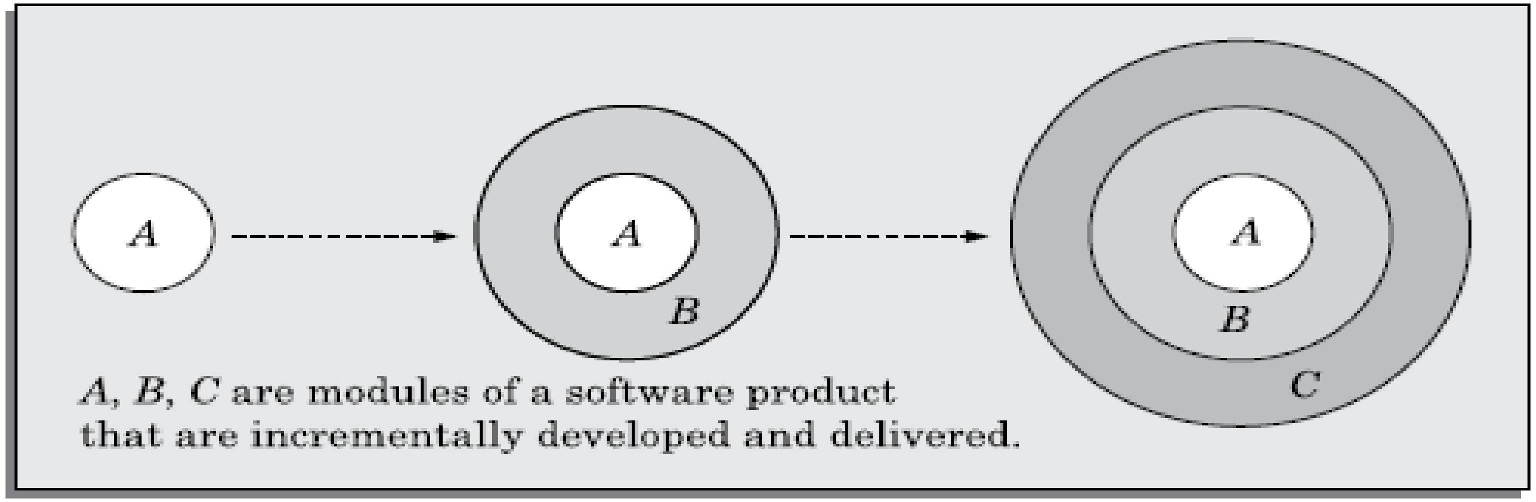

A
A
B
A
B
C
A, B, C are modules of a software product
that are incrementally developed and delivered.

FIGURE 2.7
Incremental software development.

Life Cycle Activities in Incremental Model

- **Requirements Breakdown:**
  - All software requirements are split into **modules/features**, called *increments*.
- **No Long-Term Plan:**
  - At any point, the team **plans only for the next increment**, making it easier to handle **change requests**.
- **Core Features First:**
  - Development starts with **core features**, which do **not depend on other parts**.
  - **Non-core features**, which depend on core services, are added in later increments.
- **Iterative Construction:**
  - Each increment is developed using the **iterative waterfall model**.
  - After each version is delivered, **customer feedback** is collected and used in the next iteration.

- **Successive Delivery:**
  - Each version delivered to the customer:
    - Adds **new features**
    - **Improves existing features**
  - After the final increment (**increment n**), the full software system is complete and deployed.

📌 This process is shown schematically in **Figure 2.8**.

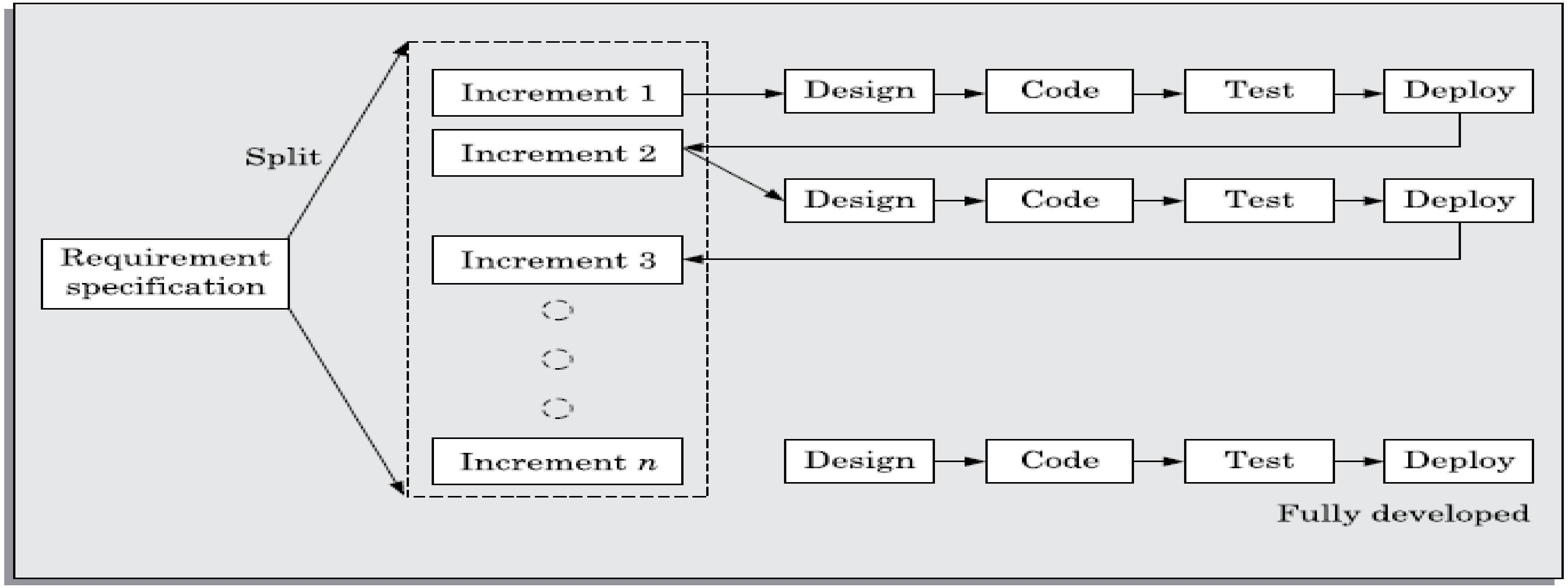


FIGURE 2.8 Incremental model of software development.

Advantages of Incremental Model

✔ **Error Reduction:**

- Core modules are delivered and used early.
- They get **thorough testing** through repeated use.
- This increases **reliability** of the final product.

✔ **Incremental Resource Deployment:**

- Customers don't need to commit **large resources at once**.
- Developers also **avoid bulk resource usage**, making the process **cost-effective and flexible**.

The **Incremental Model** breaks development into manageable chunks and delivers value early. It allows:

- **Better handling of changes**
- **Frequent customer feedback**
- **Reduced errors**
- **Gradual deployment of resources**

- It's ideal for projects where requirements evolve over time or where early delivery of parts of the system is beneficial.

2.2.6 Evolutionary Model

- The **Evolutionary Model** is a software development life cycle model where the system is built **incrementally**—one feature at a time—and refined over time based on **user feedback**. It shares similarities with the **incremental model** but introduces a key difference:
    - **Requirements, plans, and solutions evolve during the development**, rather than being finalized upfront.

Key Idea: Evolve As You Go

- The model follows a **"design a little, build a little, test a little, deploy a little"** approach.
- After getting a **rough idea** of what is needed, development starts immediately.
- New features and refinements are added with **each version**, based on customer feedback.

📌 A visual representation is shown in **Figure 2.9** (Evolutionary development diagram).

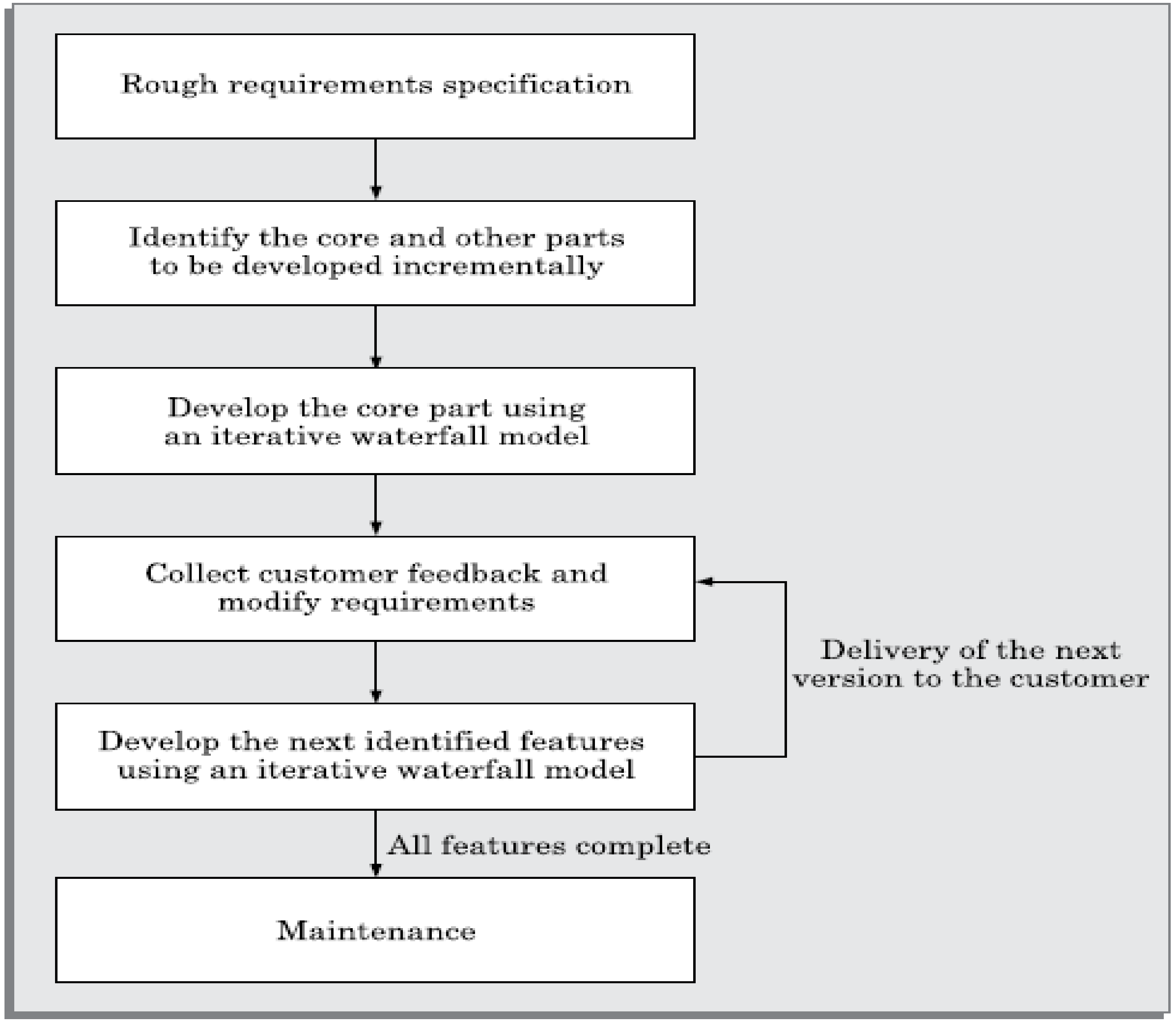

Rough requirements specification
Identify the core and other parts to be developed incrementally
Develop the core part using an iterative waterfall model
Collect customer feedback and modify requirements
Delivery of the next version to the customer
Develop the next identified features using an iterative waterfall model
All features complete
Maintenance

FIGURE 2.9 Evolutionary model of software development.

Comparison with Incremental Model

Aspect	Incremental Model	Evolutionary Model
Requirements	Fully defined *before* development begins	Evolve during development iterations
Start of development	After complete SRS document is ready	Starts with rough requirements
Feedback integration	Limited (feedback mainly after increments)	Strong emphasis on continuous feedback
Nature	Plan-driven, sequential increments	Adaptive and evolving

Advantages

1. ✔️ **Better understanding of requirements:**
    1. Users interact with early, partial versions of the software.
    2. Feedback helps refine and clarify actual needs early on.
    3. Result: **Fewer change requests** after final delivery.
2. ✔️ **Easier change handling:**
    1. Since there is a change in **long-term plans**, adapting to changes is easier.
    2. **Rework effort is minimal** compared to sequential models.

Disadvantages

- **Difficult to divide features:**
  - For some projects, especially **small or tightly coupled systems**, splitting features into incremental parts can be **complex and time-consuming**.
- **Ad-hoc design risk:**
  - Since only the **current increment** is designed at a time, overall system architecture can become **unstructured** or **inefficient**, harming **maintainability**.

- The **evolutionary model** allows building and refining software through **multiple iterations** based on real-time feedback.
- It is ideal when **requirements are unclear** or **likely to change**.
- However, it may not suit **small, well-defined projects** where complete planning is feasible from the beginning.

❖ Modern agile methods combine both incremental and evolutionary principles to get the best of both worlds.

2.3 Rapid Application Development (RAD)

RAD is a software development model introduced in the early 1990s to overcome the **rigid, change-resistant nature** of the **waterfall model**. It combines key elements of the **prototyping** and **evolutionary** models and promotes **faster development**, **early user feedback**, and **easier incorporation of changes**.

Main Goals of RAD

- **Reduce development time and cost**
- **Easily handle change requests**
- **Minimize communication gaps** between developers and customers.

How RAD Works

- **Development happens in short cycles** (called **time boxes**)
- Each cycle focuses on a **small functionality**
- A **prototype** is quickly created, shown to the customer, and **refined** based on feedback
- **Prototypes are not discarded** (unlike in the prototyping model); instead, they are **enhanced** to become the final product
- A **customer representative** is part of the development team to ensure clarity of requirements.

How RAD Supports Change and Speed

✅ **Change Accommodation:**

- Customers suggest changes soon after using each increment.
- Since each feature is developed independently, changes can be made without reworking the entire system.

✅ **Fast Development:**

- Minimal long-term planning
- **Heavy reuse** of existing components
- Emphasis on **rapid prototyping** using tools that support:
  - Visual development
  - Reusable components

When RAD is Suitable

RAD works well when:

- **Customized software** is needed
    - e.g., adapting existing educational software for different institutions
- **Non-critical software**
    - Performance and reliability are not mission-critical
- **Tight project schedules**
    - Aggressive deadlines make RAD attractive
- **Large software systems**
    - Easier to break into incremental deliveries

When RAD is Unsuitable

Avoid RAD if the project has:

- **Generic products (widely distributed)**
  - Need high performance/reliability in competitive markets
- **Requirement for optimal performance or reliability**
  - e.g., OS or flight simulator
- **Lack of similar past projects**
  - Limited scope for reuse makes RAD ineffective
- **Monolithic design**
  - If the software can’t be split into parts, incremental development is hard

RAD vs. Other Models

Comparison	RAD Model	Other Models
vs. Prototyping	Prototype evolves into final product	Prototype is thrown away
vs. Iterative Waterfall	Small features are incrementally developed with feedback	Entire system is developed before delivery; harder to change
vs. Evolutionary	Quick-and-dirty prototypes with fast delivery	Each version is carefully developed using waterfall steps; more systematic

- RAD emphasizes **speed, user involvement, reuse**, and **incremental delivery**
- Effective for projects needing **quick turnaround and custom solutions**
- Not ideal for **performance-critical, generic, or tightly coupled** software

2.4 AGILE DEVELOPMENT MODELS

Why Agile was Introduced:

- Waterfall models are rigid and not suitable for modern, dynamic projects.
- Around **40% of requirements arrive after development begins**.
- Agile responds better to changing customer needs, fast delivery demands, and customization.

Agile Model Highlights:

- **Incremental & Iterative:** Breaks down work into small parts delivered in short cycles (time-boxed).
- **Customer collaboration:** Customer representatives are involved throughout.
- **Flexibility:** Activities not needed for a specific project can be skipped.
- **Minimal documentation:** Focuses on working software and face-to-face communication.
- **Team Size:** Small (5–9 members) for effective collaboration.

<u>Agile Principles (Agile Manifesto, 2001):</u>

- Working software over comprehensive documentation.
- Welcomes requirement changes.
- Regular, frequent delivery (every few weeks).
- Emphasizes **people and communication** over tools and processes.
- Encourages **pair programming** and **customer collaboration**.

<u>Extreme Programming (XP):</u>

- Proposed by **Kent Beck (1999)**.
- Based on doing known good practices *to the extreme*.
- **Key Practices:**
  - Pair programming
  - Test-driven development (TDD)
  - Continuous integration
  - Simple design with **daily refactoring**
  - Focus on **user stories**, not formal use cases
  - Frequent feedback

Scrum Model

- Project is divided into small parts of work.
- Incrementally developed and delivered over time boxes that are called sprints.
- S/W developed over a series of manageable chunks.
- Team members assume three fundamental roles.
  - ✓ software owner - Communicates the customer's vision and priorities.
  - ✓ scrum master - Facilitates the process between the Product Owner and the team, ensuring smooth workflow and removing obstacles.
  - ✓ team member - Developers who design, build, and test the product increments.

Lean Software Development:

- Originated from Toyota manufacturing.
- Focus: **eliminate waste**, improve flow, and reduce delays.
- **Kanban Board:** Visualizes workflow using cards/stages.
    - Limits WIP (Work In Progress)
    - Makes bottlenecks and delays visible
    - Improves cycle time and predictability

Model	Key Traits
Waterfall	Heavy documentation, rigid, good for stable, critical systems
Agile	Lightweight, flexible, customer-focused, iterative
RAD	Fast prototyping, but lacks Agile's discipline
Exploratory	Similar informality, but lacks Agile's structure
XP	Emphasizes testing, feedback, simplicity
Lean + Kanban	Efficiency-focused, bottleneck visualization, WIP limits

2.5 SPIRAL MODEL

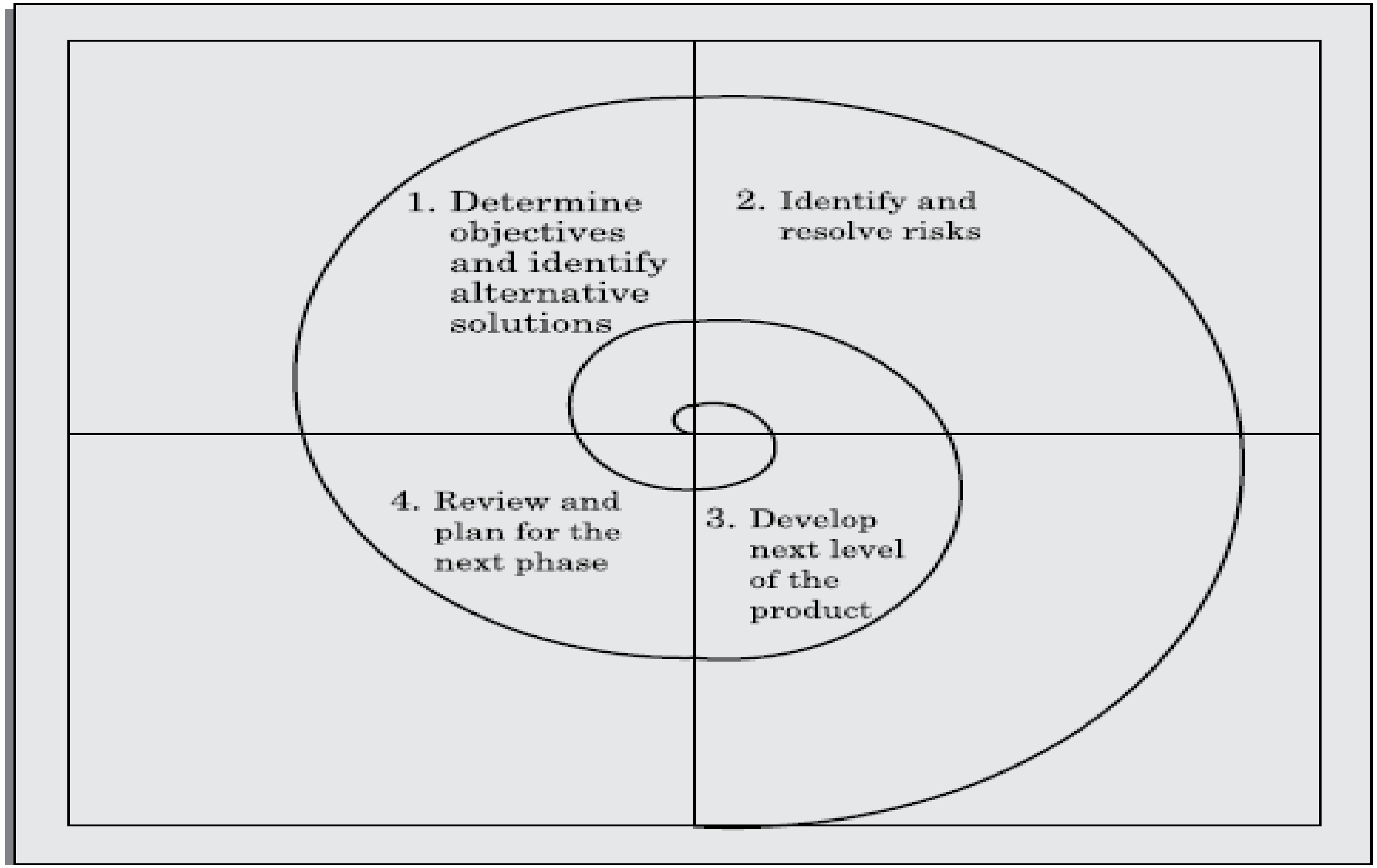

1. Determine objectives and identify alternative solutions
2. Identify and resolve risks
3. Develop next level of the product
4. Review and plan for the next phase

FIGURE 2.10 Spiral model of software development.

- The **spiral model** which resembles a spiral with **multiple loops**.
- Each **loop** is considered a **phase** of development.
- The **number of loops is not fixed** and depends on project-specific risks.

Key Feature: Risk Handling

- Unlike the **prototyping model**, which assumes risks are known **before** development starts, the spiral model **identifies and resolves risks throughout the project**.
- In **every phase**, a **prototype** may be built to analyze risks and test solutions.
- Example: If **remote data access** is slow, a prototype can test performance and suggest improvements like caching or faster communication.

2.5.1 Phases of Spiral Model

Each **phase (loop)** is divided into **4 quadrants**:

- **Quadrant 1**:
  - Define objectives and identify possible **risks**.
  - Propose **alternative solutions**.
- **Quadrant 2**:
  - **Evaluate** the alternatives.
  - **Develop prototype(s)** to compare solutions.
- **Quadrant 3**:
  - **Develop and test** the selected solution.
  - Build the **next version** of the software.
- **Quadrant 4**:
  - **Review** the results with the **customer**.
  - **Plan the next iteration** of the spiral.
- Each iteration moves **outwards** from the center, gradually building the complete system.

Key Role: Project Manager

- **Decides** the number of phases based on emerging risks.
- **Adapts** the model dynamically for the project.

Parallel Cycles

- Features that can be developed independently may go through **parallel spirals** to speed up development.

Advantages

- Excellent for **risk-driven** projects.
- Supports **prototyping in every phase**.
- More **flexible** than waterfall or prototyping.
- Suitable for **large, high-risk** projects.

Disadvantages

- **Complex** to manage and implement.
- Needs **experienced staff**.
- Not suitable for **outsourced** or low-risk projects.

Spiral Model as a Meta Model

- It **includes** features from:
  - **Waterfall model** (systematic approach)
  - **Prototyping model** (risk resolution via prototypes)
  - **Evolutionary model** (software evolves across iterations)

  - A **single spiral loop** behaves like the **waterfall model**.

Case Study: Galaxy Inc.

- Project: Satellite-based mobile communication across Earth.
- **High risk** due to unknown challenges (e.g., satellite handoff).
- Used **spiral model** because:
  - Risks couldn't be fully known upfront.
  - Project required **millions of lines of code**.
  - Used **parallel spirals** to speed up development.
  - Project successfully completed in **5 years** after multiple refinements in **cost and schedule**.

The spiral model is ideal for:	It's not suitable for:
Large, complex, high-risk projects	Small or low-risk projects
Projects where new risks emerge during development	Teams with inexperienced staff
Teams with experienced developers	Outsourced projects

2.6 Comparison of Software Life Cycle Models

1. Waterfall Model (Classical)

- Acts as the **basic model**.
- Does **not allow correction** of errors found in later phases.
- Hence, **not suitable** for practical projects - E-Commerce, Mobile Apps, Healthcare, AI/M, Banking System.

2. Iterative Waterfall Model

- Adds **feedback paths** to correct earlier errors.
- Widely used due to its **simplicity** and ease of understanding.
- Best for **well-understood, low-risk** problems.
- **Not suitable** for large or highly risky projects - Smart City, Autonomous Vehicle, National Healthcare Records, Spacecraft Navigation, Online Stock Trading.

3. Prototyping Model

- Ideal when **requirements or technical solutions** are not well understood.
- All **risks must be identifiable** at the beginning.
- Popular for designing **user interfaces**.
- Not Suitable for - Payroll systems, calculator software, Embedded systems, Defense systems, Backend enterprise systems, etc..,

4. Evolutionary Model

- Suitable for **large problems** that can be broken down into **modules**.
- Supports **incremental development and delivery**.
- Commonly used in **object-oriented** development.
- Only applicable if **incremental delivery is acceptable** to the customer.
- Not suitable - Compiler Design, Banking Transaction, Microcontrollers, Navigation and Control Systems, Tax Filing and Regulatory Reporting, etc..,

5. Spiral Model

- A **meta-model** that encompasses all other models.
- Built for **flexibility** and **risk handling**.
- Best for **large, complex, high-risk** projects where risks emerge over time.
- More **complex** than other models, which can deter its use in simple projects.
- Not suitable - Simple CRUD Applications (Create, Read, Update, Delete), Educational or Academic Mini Projects, Time-Critical Projects , Small Utility Tools, Maintenance Projects or Bug Fixes, etc..,

Prototyping vs Spiral Model

- **Prototyping Model**: Use when **risks are few and known early**.
- **Spiral Model**: Use when **risks are high or appear during development**.

Customer Perspective

- Initially, **customer confidence** is high across all models.
- As time passes and no working software appears, confidence drops, leading to **frustration.**
- **Evolutionary/incremental models** show working versions early, **maintaining customer trust**.
- Also help customers **adjust gradually** to new software.
- From a financial view, **incremental models** reduce upfront cost — software can be purchased **in parts**.

2.6.1 Selecting a Suitable Model

Factors to Consider:

- **Software Characteristics**:
  - **Agile model**: Best for small, service-based projects.
  - **Iterative waterfall**: Suitable for **product or embedded** software.
  - **Evolutionary model**: Preferred for **object-oriented** systems.
- **Team Characteristics**:
  - **Experienced team**: Can use simpler models like iterative waterfall even for complex systems.
  - **Inexperienced team**: May need prototyping even for basic applications.
- **Customer Characteristics**:
  - **Inexperienced customers**: Tend to **change requirements frequently**.
  - In such cases, **prototyping** helps avoid later change requests.

- ❖ **Waterfall (Classical)**: Theoretical, not used in real projects.
- ❖ **Iterative Waterfall**: Simple and common, but limited to well-defined problems.
- ❖ **Prototyping**: For unclear requirements but known risks.
- ❖ **Evolutionary**: For large systems needing gradual delivery.
- ❖ **Spiral**: Most flexible and robust, but complex; best for large, risk-prone projects.

SOFTWARE ENGINEERING

Module – 3 (REQUIREMENT ANALYSIS AND SPECIFICATION)

CONTENTS

Importance of Requirements Analysis and Specification

- In **plan-driven life cycle models**, it's **essential** to **understand and document exact customer requirements** *before* starting development.
- In the past, many projects failed because developers started coding **without confirming** what the customer really wanted.
- This leads to:
  - Frequent changes in later phases
  - Increased **development cost**
  - **Customer dissatisfaction**
  - Possible **legal disputes**

- ✔️ Therefore, experienced developers treat **requirements analysis and specification** as a **critical phase** in software development.

Role of a Good Requirements Document

- A good **requirements document**:
  - Clarifies what features the software should have
  - Serves as a **foundation** for later phases like design, coding, testing, etc.
- In **outsourced projects**, it acts as a **contract** between the customer and developer.
- Even in **in-house product development**, the marketing team may act as the customer, so precise documentation is still important.
- In **small service-based projects**, **agile methods** may allow **incremental development of requirements**.

When Does Requirements Analysis Start?

- Begins **after the feasibility study**, when the project is confirmed to be **technically and financially viable**.

Who Performs This Phase?

- Conducted by **experienced team members**, often called **System Analysts**.
- Analysts may spend time at the **customer site**.
- Responsibilities:
  - Collect data about what the software must do
  - Understand the exact user requirements
  - Identify and remove inconsistencies, ambiguities, and incompleteness
  - Create the **Software Requirements Specification (SRS)** document

Validation of the SRS Document

- **Internal Review** by the project team to ensure:
    - Accuracy
    - Completeness
    - Consistency
    - Unambiguity
- **Customer Review**:
    - Customer reviews and approves the SRS.
    - Once approved, the SRS becomes the **basis for all future development**.
    - Also acts as a **legal agreement** between customer and developer.

Main Activities in Requirements Analysis & Specification

- **Requirements Gathering and Analysis**:
    - Collecting and analyzing customer needs.
- **Requirements Specification**:
    - Documenting the analyzed requirements in the form of the SRS.

- ❖ Requirements analysis and specification is vital to **project success**.
- ❖ It helps avoid **costly changes** and **disputes** later.
- ❖ The **SRS document** is the key outcome, serving as both a **blueprint** and a **contract**.
- ❖ Conducted by **system analysts**, reviewed by both developers and customers.
- ❖ Forms the **foundation** for all subsequent development activities.

3.1 Requirements Gathering and Analysis

Goal of Requirements Analysis and Specification Phase

- To **clearly understand customer requirements** and organize them into a formal document called the **Software Requirements Specification (SRS)**.
- The **SRS** is the final and crucial outcome of this phase.

Realistic Expectation

- Customers **do not provide all requirements** in a single document.
- Requirements are **scattered,** vague, and come from **multiple sources** (different stakeholders).
- These must be **gathered and then analyzed** to remove issues like **ambiguities, inconsistencies, and incompleteness**.

Two Main Tasks

- **Requirements Gathering (also called Elicitation)**
- **Requirements Analysis**

3.1.1 Requirements Gathering

- **Objective:** **Collect all software requirements from stakeholders.**
- **Challenges:**
  - Multiple stakeholders with partial views
  - Scattered and unstructured information
  - No existing system in case of new software
- **If a manual system exists (e.g., office record-keeping):**
  - Easier to gather input/output forms and process steps.
- **Typical Activities:**
  - **Study existing documents** like the Statement of Purpose (SoP)
  - **Visit customer site**
  - **Interview** users and representatives
  - Use methods like:
    - **Questionnaires**
    - **Task analysis**
    - **Scenario analysis**
    - **Form analysis**

Important Techniques Used in Requirements Gathering

- **Studying Existing Documentation**
  - Read SoP documents that outline context, goals, stakeholders, and expected features.
- **Interview**
  - Identify different user categories and their needs.
  - Use **Delphi technique** to circulate a draft, gather feedback, and iterate until agreement.
- **Task Analysis**
  - Break down software into user-performed **tasks**.
  - Define steps for each task (e.g., issuing a book in a library system).
- **Scenario Analysis**
  - Understand how tasks behave under different conditions (scenarios).
  - E.g., Book issue successful, book reserved, or member exceeded issue limit.
- **Form Analysis**
  - Analyze existing manual **input forms and output reports**.
  - Understand how outputs are generated from inputs

✅ Case Study: Office Automation in CSE Department

- ➢A team of students automated grading, leave, and store tasks for a department.
- ➢Analyst interviewed staff, collected forms, understood tasks, scenarios, and outputs.
- ➢Input/output data formats and office procedures were accurately captured.

3.1.2 Requirements Analysis

➤ **Objective:**

- ➢Analyze the gathered data to:
- ➢Fully understand customer requirements
- ➢Eliminate **ambiguities, inconsistencies, and incompleteness**

- **Pre-analysis Questions:**
  - What is the problem?
  - Why is it important?
  - What are the inputs/outputs?
  - What processes are needed?
  - What are the potential difficulties?
  - What external interfaces exist?

Common Problems in Requirements

1. Anomaly (Ambiguity)

- Multiple meanings due to vague terms
- **Example**: “Switch off heater when temperature is high” (what is "high"?)

2. Inconsistency

- Conflicting requirements from different stakeholders
- **Example**: One says “switch off furnace at 500°C”; another says “keep it on and trigger water spray”.

3. Incompleteness

- ➢Missing necessary features or data
- ➢**Example**: Requirement to email parents about grades, but no provision to input parent email addresses.

➤ Can all problems be detected?

- ➢Many issues are caught during careful review.
- ➢Some are **subtle** and hard to detect.
- ➢Use of **formal methods** can help systematically find these issues.
- ➢Formal specification is especially used for **safety-critical systems**.

- ❖Requirements must be **gathered methodically** from multiple sources.
- ❖They are then **analyzed** to ensure clarity, consistency, and completeness.
- ❖The outcome is a validated **SRS document** that forms the **foundation** for the rest of development.

3.2 Software Requirements Specification (SRS)

Once all requirements are gathered and cleaned (i.e., **no inconsistencies, ambiguities, or incompleteness**), the analyst organizes them into a **structured, informal document** called the **Software Requirements Specification (SRS)**.

🔸 Importance of the SRS

- ➢One of the **most important and hardest** documents to write in the software development life cycle.
- ➢Must satisfy the needs of **multiple users with different perspectives**.

3.2.1 Users of SRS Document

- Various stakeholders use the SRS for different purposes:

User	Purpose
Users, Customers, Marketers	Verify the system meets their needs.
Software Developers	Develop software based on clearly defined requirements.
Test Engineers	Design test cases and test plans from input/output details.
User Documentation Writers	Understand features to write user manuals.
Project Managers	Estimate cost, effort, and plan the schedule.
Maintenance Engineers	Understand system functions for debugging, updating, or enhancements.

📌 **The SRS acts like a contract between the developers and the customer, and can even be used in legal disputes.**

3.2.2 Why Create an SRS Document?

- Benefits of having a well-prepared SRS:
- **Contractual agreement**: Sets clear expectations between developers and customers.
- **Reduces rework**: Early clarity avoids redesign, recoding, and retesting later.
- **Cost and schedule estimation**: Helps project managers estimate effort, budget, and delivery time.
- **Baseline for validation & verification**: Used to check if the final product meets the documented requirements.
- **Supports future extensions**: Serves as a reference for planning enhancements.

3.2.3 Characteristics of a Good SRS Document

- As per **IEEE 830 Standard**, a good SRS must be:

- Figure 3.1, the SRS document describes the output produced for the different types of input and a description of the processing required to produce the output from the input (shown in ellipses) and the internal working of the software

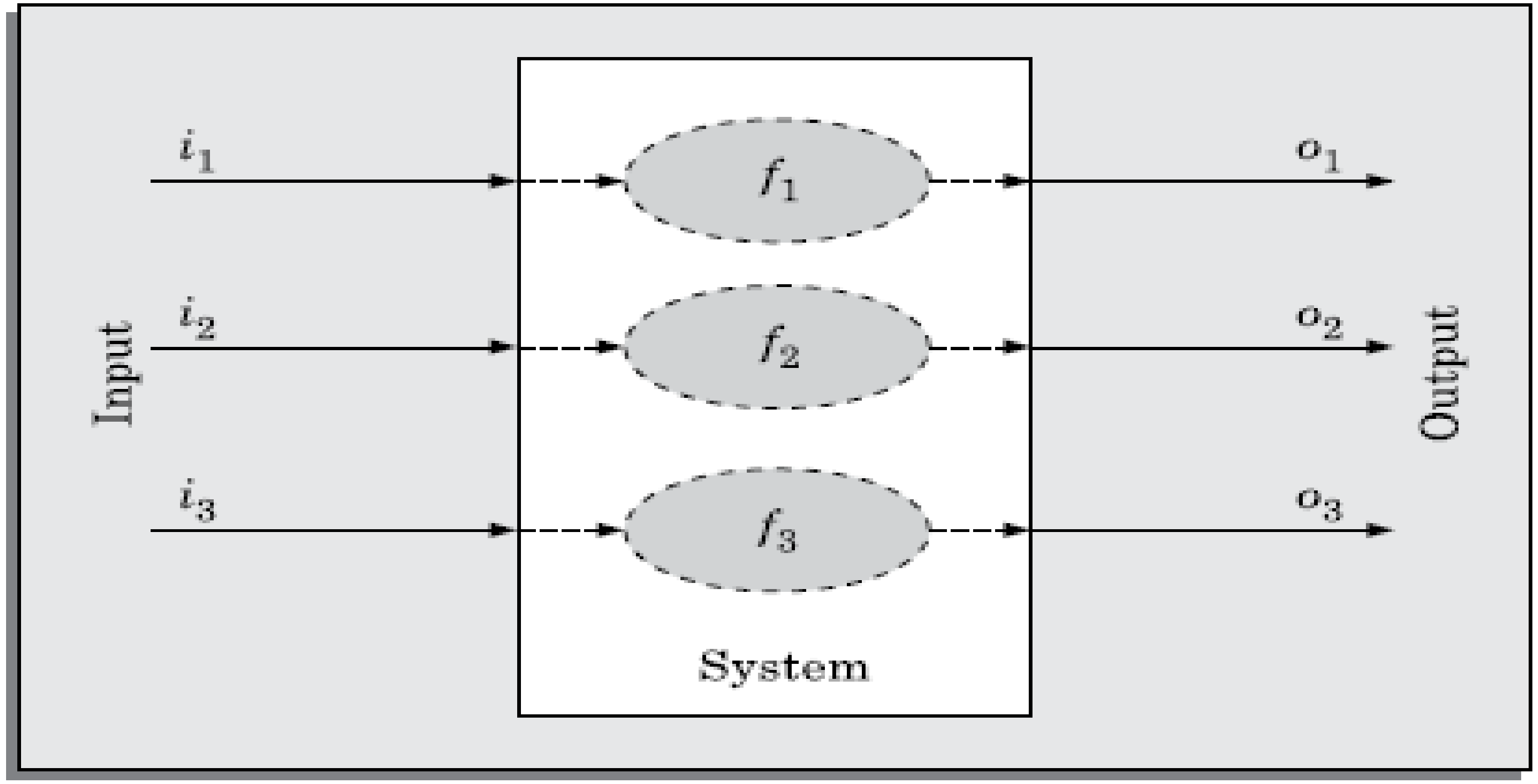


FIGURE 3.1 The black-box view of a system as performing a set of functions..

Characteristic	Description
Concise	Clear, brief, complete, and consistent; avoids unnecessary detail.
Implementation-independent	Focuses on what the system does, not how; avoids design decisions.
Traceable	Each requirement can be traced to its design, code, and test case, and vice versa.
Modifiable	Well-structured to accommodate changes without confusion.
Identifies responses to errors	Specifies how the system reacts to undesired or exceptional events.
Verifiable	Requirements must be testable. Vague terms like “user-friendly” are avoided.

✅ **The SRS provides a black-box view: it specifies the external behavior without discussing internal logic.**

3.2.4 Attributes of Bad SRS Documents

- Common problems found in poorly written SRS documents:

Problem	Explanation
Over-specification	Includes design-level decisions (e.g., internal storage mechanisms).
Forward References	Refers to content explained much later, reducing readability.
Wishful Thinking	Contains unrealistic or vague desires that are hard to implement.
Noise	Irrelevant details (e.g., staff working hours) that clutter the document.

Other common issues include:

- **Ambiguity**
- **Incompleteness**
- **Contradictions**

✅ **These issues must be avoided or used as a checklist to review and improve the quality of the SRS.**

3.2.5 Important Categories of Customer Requirements (IEEE 830)

A well-structured SRS document **categorizes requirements** into key sections:

1. Functional Requirements

- Describe **what the system should do**.
- Each function (fi) is like a mathematical transformation:

  f: I → O → input data (Ii) transforms into output data (oi).

- These include:
  - **User inputs**
  - **Processing logic**
  - **Expected outputs**

- Each function must clearly mention its inputs and corresponding outputs.

🧠 **Note: Functional requirements form the core of the SRS and are essential for system design and testing.**

2. Non-Functional Requirements

These are **mandatory quality attributes** and constraints **not related to specific functions**, but affect overall system behavior and quality.

They include:

- **a) Design and Implementation Constraints**

Restrictions on how the system must be built.

- Examples:
  - Use Oracle DBMS
  - Follow specific security protocols
  - Use a particular programming language or platform

- **b) External Interfaces**
  - Requirements related to interaction with:
    - Hardware
    - Software
    - Communication channels
    - Users (GUI layout, screen design standards, error messages, etc.)

c) Other Non-Functional Requirements

- Performance (e.g., 100 transactions/sec)
- Reliability
- Security
- Usability
- Maintainability
- Portability

❖ 💡 IEEE 830 recommends documenting:

- **External interfaces**
- **Design & implementation constraints**

in separate sections, followed by **other non-functional requirements**.

3. Goals of Implementation

- These are **optional, non-binding suggestions** from the customer.
- Developers may consider them during design.
- Examples:
  - Easy support for future enhancements
  - Reusability
  - Flexible for adding new device support

- 📌 Difference:
  - **Non-functional requirement**: Must be tested for compliance.
  - **Goal**: Desirable, not tested during system acceptance.

4. **Classification Criteria**

Type	**Definition**
Functional Requirement	Input → Output transformation (functions of the system)
Non-Functional Requirement	Testable characteristics not expressible as functions (e.g., performance, security)
Goal of Implementation	Non-testable customer suggestions (e.g., design for future ease of change)

<u>3.2.6 Functional Requirements</u>

📌 **What Are They?**

- Functional requirements describe **high-level functions** the user **explicitly invokes** to get useful work done.
- Each high-level function:
  - Accepts input
  - Processes it
  - Produces output

<u>Example:</u>

In a **Library System**, a function like search-book:

- Takes keywords as input
- Searches the database
- Displays matched books

<u>Important Clarification</u>

- **Not every activity is a high-level function.**
- Example: Printing ATM receipts is **part of** withdraw money, not a separate high-level function.

🔄 High-Level Function = Multiple Sub-Requirements

- A function often involves **step-by-step interactions** between user and system.
- Each interaction step can be a **sub-requirement**.

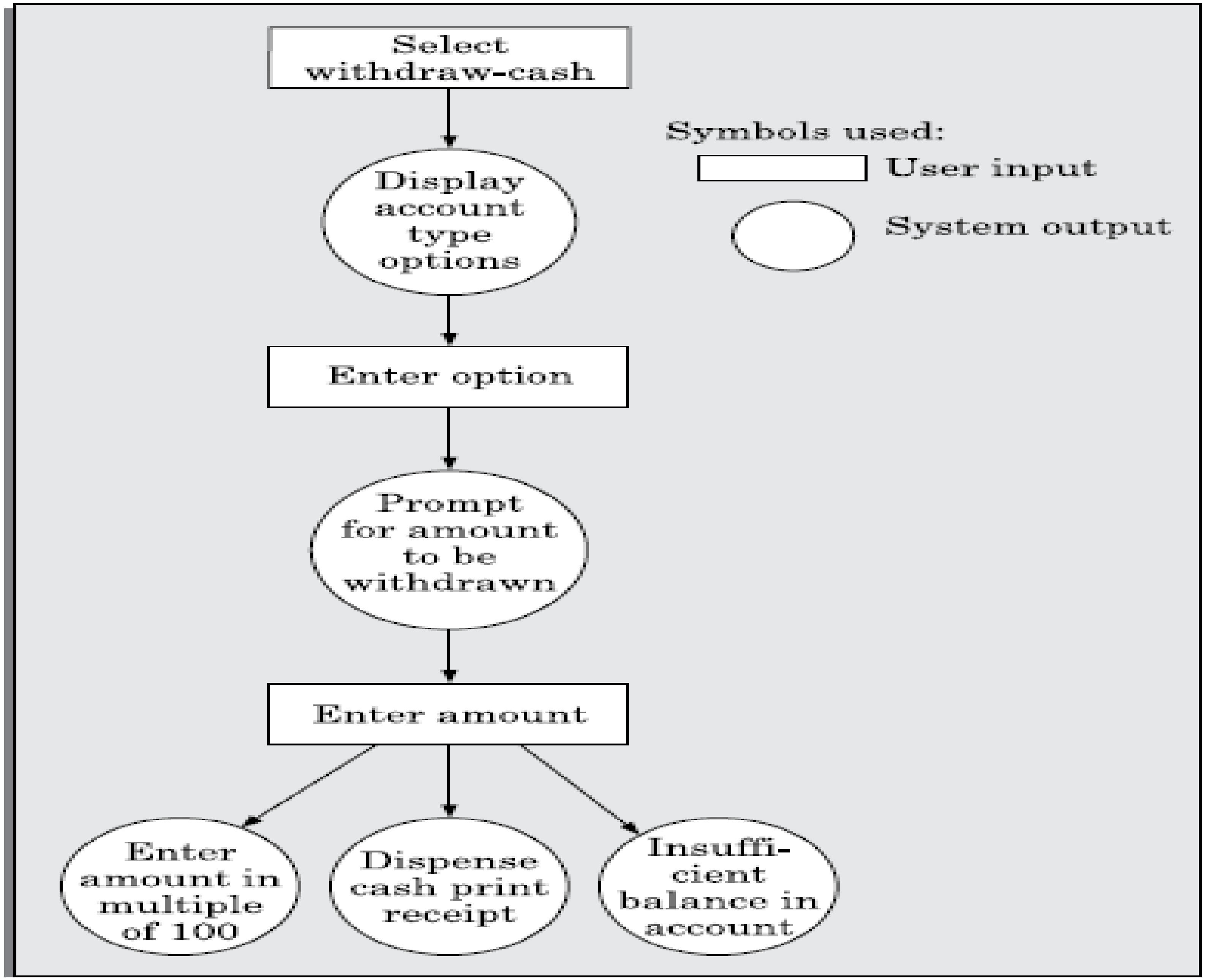

Select withdraw-cash
Display account type options
Enter option
Prompt for amount to be withdrawn
Enter amount
Enter amount in multiple of 100
Dispense cash print receipt
Insufficient balance in account
Symbols used:
User input
System output

FIGURE 3.2 User and system interactions in high-level functional requirement.

Can You Identify All Inputs and Outputs Exactly?

- Ideally yes, but not always possible, especially **without a working prototype**.
- In such cases, use **high-level terms** to describe the data.
- Example: In ATM withdraw-cash:
  - User input is split across steps (e.g., account type, amount), not given all at once.

Aspect	Description
Functional Requirements	What the system should do (functions with input and output).
Non-Functional Requirements	Quality and technical constraints (performance, interfaces, security, etc.)
Goals of Implementation	Optional suggestions (e.g., future flexibility) — not tested during acceptance.
High-Level Functions	User-invoked useful services, often involving multiple interaction steps.

3.2.7 How to Identify the Functional Requirements?

✦ Steps to Identify Functional Requirements:

- ➢**Start with informal problem description** or domain understanding.
- ➢**Identify user types** (different users → different expectations).
- ➢**List services** expected by each user.

✦ Subjectivity in Identification:

- ➢Some decisions involve judgment:
  E.g., In a **Library System**, entering book details might be a sub-function of "Issue Book" instead of a separate high-level function.

- Use **common sense** and **visualize usage scenarios** to determine what constitutes a **meaningful unit of functionality** for the user.

3.2.8 How to Document Functional Requirements?

📌 What to Include for Each Functional Requirement:

- **State** when the function is invoked.
- **Input domain**
- **Output domain**
- **Processing logic**
- **Scenarios** (alternate paths depending on user input)

Example: Withdraw Cash from ATM

ID	Requirement	Input	Output	Processing
R.1	Withdraw Cash	-	Cash or Error Message	Check balance, dispense cash or error
R.1.1	Select withdraw amount option	Menu option	Prompt for account type	-
R.1.2	Select account type	Savings/Checking/Deposit	Prompt for amount	-
R.1.3	Enter amount	Amount (₹100–₹10,000)	Dispensed cash + printed receipt / error	Validate balance and debit if enough funds exist

Example: Search Book in Library

ID	Requirement	Input	Output	Processing
R.1	Search Book	Key words	Matching book details	Search by title/author
R.1.1	Select search option	Option click	Prompt for keywords	-
R.1.2	Search and display	Keywords	Book title, author, year, ISBN, availability, etc.	Keyword matching and display

Example: Renew Book

- Involves login, selecting a book, validating, and confirming.
- Has multiple sub-requirements (R.2.1 to R.2.3).

3.2.9 Traceability

✦ **What is Traceability?**

- Traceability ensures that every requirement:

  Can be **linked to its design component**

  Can be **linked to code**

  Can be **linked to its test cases**

✦ **Importance:**

- Confirms that **all requirements** are implemented and tested.
- Helps assess the **impact of changes** or bugs.

✦ **How to Ensure Traceability:**

- **Number every requirement uniquely**.
  - E.g., R.1, R.1.1, R.1.2, R.2, etc.

3.2.10 Organization of the SRS Document (IEEE 830)

Key Sections in SRS:

Section	What It Covers
1. Introduction	Purpose, scope, system environment, product overview
2. Overall Description	Product perspective, features, user classes, hardware/software environment
3. Functional Requirements	Grouped by user class or operation mode; each function with ID and details
4. External Interface Requirements	UI, hardware, software, communication interfaces
5. Other Non-Functional Requirements	Performance, security, safety, design constraints
6. Goals of Implementation	Optional suggestions (e.g., future-proofing, reusability, extensibility)

IEEE 830-1998 Standard for SRS

- Title
- Table of Contents
- 1. Introduction
  - 1.1 Purpose
  - 1.2 Project Scope
  - 1.3 Environmental Characteristics
- 2. Overall Description
- 3. Functional Requirements
- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices

Index

•

- This document specifies the requirements for the Task Manager Application.

- To create, manage, and track tasks with reminders and priorities, syncing across multiple devices.

- Outline the environment(Hardware and Software) with which the software will interact

IEEE 830-1998 Standard – Section 2 of SRS

- Title
- Table of Contents
- 1. Introduction
- 2. Overall Description
  - 2.1 Product Perspective
  - 2.2 Product Features
  - 2.3 User Characteristics
  - 2.4 Operating Environments
  - 2.5 Design and Implementation Constraints
  - User documentation

•This section explains how the product fits into the larger system, its relationship to other systems, and its dependencies. **Interfaces with other software/hardware, Dependencies on external systems, High-level context diagram (optional).**

This section lists the main functions and capabilities of the product.
- **Create, read, update, delete tasks**
- **Set reminders**
- **Categorize tasks**
- **Synchronize tasks across devices**

•This section describes the different user types (classes), their skills, experience, and technical knowledge relevant to using the product. **Types of users (beginner, advanced, admin, etc.), Education/technical skill levels, Frequency of use**

• specifies the hardware, software, network, and system configurations. **Operating systems (Windows, macOS, Linux), Hardware requirements (Processor, RAM, storage), Browser versions (Chrome 115+, Firefox 100+), Network type (LAN, Cloud, 5G)**

• This describes limitations and restrictions that affect design or coding decision. **Regulatory compliance, Programming languages to be used, Database requirements, Communication protocols (HTTPS), Performance limitations**

•This section lists the manuals, guides, online help, and training materials that will be delivered with the product for end users. **User manual (PDF or printed), Installation guide, FAQ/help section in the application, Video tutorials and quick-start guides**

IEEE 830-1998 Standard – Section 3 of SRS (2)

- …
- 1. Introduction
- 2. Overall Description
- 3. Functional Requirements
  - 1. User Class(Operation mode) 1
    - (a) Functional requirement 1.1
    - (b) Functional requirement 1.2
  - 2. User Class(Operation mode) 2…….
- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices

Index

- Classify the functionalities either based on the specific functionalities invoked by different users
- Or the functionalities that are available in different modes etc. depending on what may be appropriate

- User Interfaces
- Hardware Interfaces
- Software Interfaces
- Communication Interfaces

- Performance Requirements
- Safety Requirements
- Security Requirements

IEEE 830-1998 Standard – Section 3 of SRS (2)

- …
- 1. Introduction
- 2. Overall Description
- 3. Functional Requirements
- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices

Index

- User Interfaces
- Hardware Interfaces
- Software Interfaces
- Communication Interfaces

IEEE 830-1998 Standard – Section 3 of SRS (2)

- …
- 1. Introduction
- 2. Overall Description
- 3. Functional Requirements
- 4. External Interface requirements
- 5. Other non-functional Requirements

Appendices

Index

- Performance Requirements
- Safety Requirements
- Security Requirements

Example: Personal Library Software – Functional & Non-Functional Requirements

Functional Requirement Categories:

1. **Manage Own Books**
   - Register book
   - Issue/return/query books
2. **Manage Friend Details**
   - Register/update/delete friends
3. **Manage Borrowed Books**
   - Record borrowed/deregister returned books
4. **Manage Statistics**
   - View count, investment, transaction summaries

Non-Functional Requirements:

- Free public DBMS (N.1)
- Runs on Windows & UNIX (N.2)
- Web-query support (N.3)

Summary Table: Requirement Types

Type	Description
Functional Requirement	Input → Output transformation, invoked directly by user
Non-Functional Requirement	Constraints (e.g., performance, platform, security, usability, etc.)
Goal of Implementation	Optional suggestions to developers (not validated during acceptance)
Traceability	Ability to trace requirement ↔ design ↔ code ↔ test

3.2.11 Techniques for Representing Complex Logic

When a **high-level functional requirement** involves **complex decision-making** or **multiple alternate flows**, describing it in plain text becomes difficult and error-prone.

- To handle such complexity, two techniques are commonly used:
  - **Decision Trees**
  - **Decision Tables**

These are especially useful in:

- Identifying **all possible conditions and outcomes**
- Designing **clear test cases**
- Making **logic easier to validate and implement**

When to Use:

Scenario	**Use?**
Few alternatives, simple logic	Plain text is sufficient
Many conditions, multiple outcomes	Use decision trees/tables
Need clear test case generation	Use decision tables

Decision Tree

✦ **Definition:**

- A **graphical representation** of decision logic.
- **Internal nodes**: conditions
- **Edges**: outcomes (yes/no or true/false)
- **Leaf nodes**: actions to be taken

🧪 **Example: Library Membership System (LMS)**

- **Supported Options:**
  - ➢New Member
  - ➢Renew Membership
  - ➢Cancel Membership

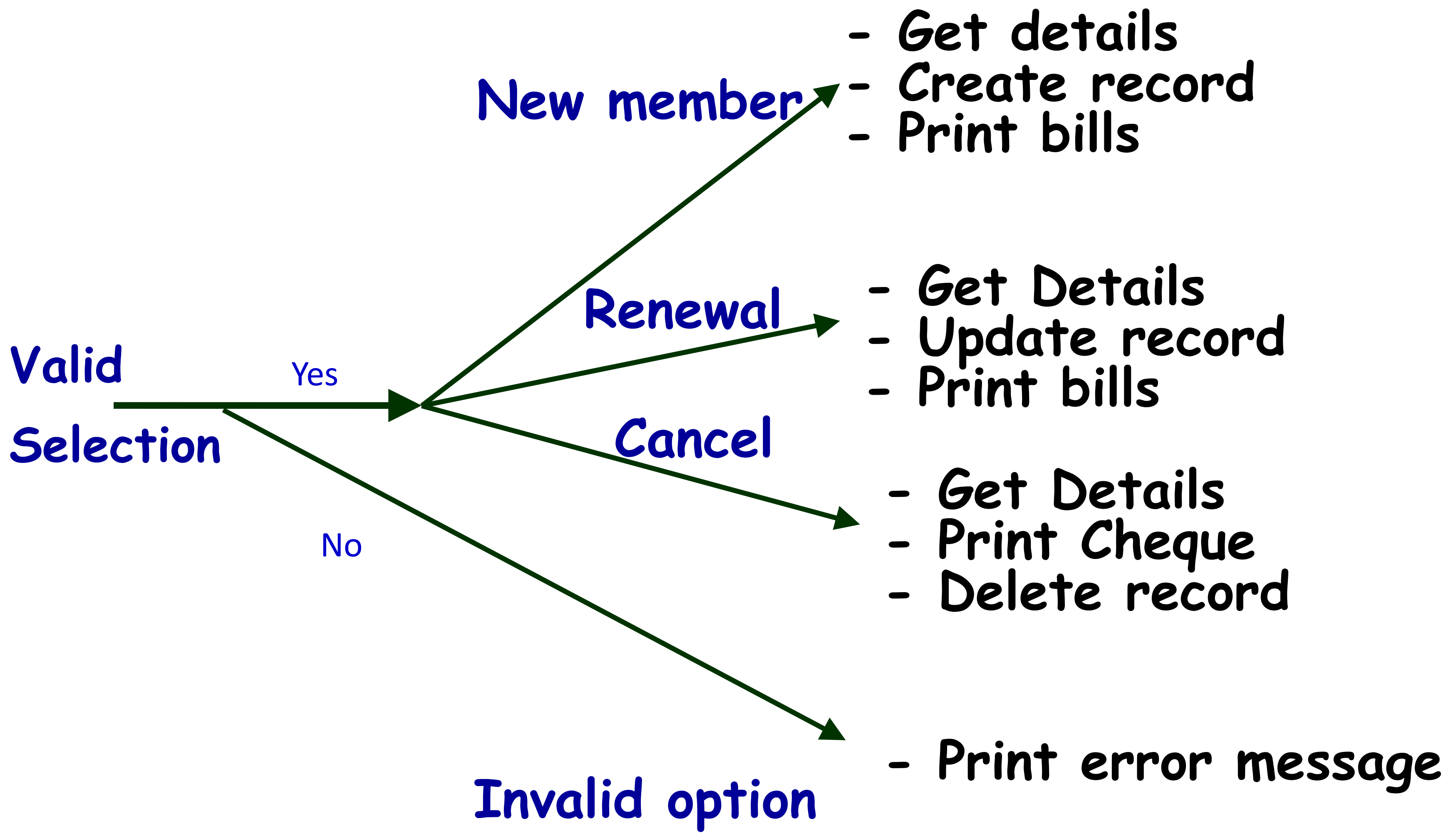
📌 **Logic (simplified):**

- ➢ If **invalid option**, → Show error message
- ➢ If **New Member**, → Ask details → Create record → Generate bill
- ➢ If **Renew**, → Ask member info → Validate → Update date → Generate bill / Show error
- ➢ If **Cancel**, → Ask name → Validate → Cancel → Print cheque → Delete record

- ➡️ This logic is shown using a **decision tree in Figure 4.3**.

- ✅ **Best For:**
  - Clear **hierarchical** and **multi-level decisions**
  - Easy visualization (when conditions are few)

Decision Tree Example



Decision Table

✦ **Definition:**

- A **matrix-style tabular representation** where:
- **Rows**: conditions (top) and actions (bottom)
- **Columns (Rules)**: unique combinations of condition outcomes

Decision Tree vs Decision Table

Feature	Decision Tree	Decision Table
Readability	Easier for small number of conditions	Better for large number of combinations
Order of Decision Making	Explicitly shown	Abstracted (not shown)
Multi-level Decision Representation	Supports it easily	Not intuitive for multi-level decisions
Compactness (for many conditions)	Becomes cluttered	More compact and structured
Test Case Generation	Manual	Easier and more direct

- 🧪 **Example: Decision Table for LMS**

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Valid selection	NO	YES	YES	YES
New member selected	-	YES	NO	NO
Renewal selected	-	NO	YES	NO
Cancel selected	-	NO	NO	YES
Actions				
Show error	✅			
Ask member info		✅		✅
Create record		✅		
Generate bill		✅	✅	
Ask for mem. details			✅	
Update expiry			✅	
Print cheque				✅
Delete record				✅

How to Read This Table:

- Each **column (Rule)** represents one scenario or decision path.
- **'×'** indicates the **action to be performed** under that rule.
- **'-'** indicates that the condition is **not relevant** or not evaluated for that rule.

3.3 Formal System Specification

Why Formal Methods?

Formal methods are mathematical approaches used in software engineering to **precisely describe, model, and verify systems**. They help ensure a system is **correctly implemented**—meaning it behaves exactly as its specification demands.

4.3.1 What is a Formal Technique?

A **formal technique**:

- Is a **mathematical method** to:
  - Specify systems (hardware/software)
  - Check whether a specification is possible (realisable)
  - Prove an implementation matches its specification
  - Verify system properties **without execution**

A formal specification language has:

- **Syntactic domain (syn):** Symbols + rules to form valid statements.
- **Semantic domain (sem):** Meaning/interpretation of those statements.
- **Satisfaction relation (sat):** Defines when the system model satisfies the specification.

📌 **System Development Hierarchy:**

Every stage (requirements → design → coding) serves as:

- **Implementation** of the previous stage
- **Specification** for the next stage
- Formal techniques can verify that **each stage aligns with the previous**.

Syntactic and Semantic Domains

➢ **Syntactic domain:**

Includes an alphabet and rules to form **well-formed formulas**.

➢ **Semantic domain:**

Varies by type:

- ➢ **Abstract Data Types:** Algebras, theories, etc.
- ➢ **Programs:** Input-output functions
- ➢ **Concurrent Systems:** States, events, transitions, etc.

Satisfaction Relation

- Used to check if a **system model** matches its specification.
- A **semantic abstraction function** maps system behavior to simplified representations.
- Two types:
  - Behavior-preserving
  - Structure-preserving

Model vs Property-Oriented Methods

- **Model-Oriented:**
  - Describes **how** the system behaves by **building a model**.
  - Uses **mathematical structures** like sets, sequences, tuples.
  - Example methods: **Z, CSP, CCS**

- **Property-Oriented:**
  - Describes **what** the system should do by listing **axioms or properties**.
  - Allows flexibility in implementation.
  - Example methods: **Axiomatic, Algebraic Specification**

📌 Example (Producer/Consumer):

- Property-oriented: State rules like "Consumer can only consume after production."
- Model-oriented: Define actions like produce(p) and consume(c).

📌 Usage:

- **Property-oriented** → Better for **requirements specification**
- **Model-oriented** → Better for **system design**

4.3.2 Operational Semantics

Operational semantics define **how system behavior is represented** during execution. Common types:

1.Linear Semantics:

- ➢Describes system behavior as **sequences of events/states**
- ➢Concurrent actions shown by **interleaving** (e.g., a||b = a;b or b;a)

2.Branching Semantics:

- ➢Uses a **graph** to show multiple paths from a state
- ➢Better shows decision points but still uses interleaving for concurrency

3.Maximally Parallel Semantics:

- ➢All concurrent actions run **together**
- ➢Assumes full resources are always available (unrealistic)

4.Partial Order Semantics:

- ➢Describes events with a **partial order**
- ➢Some events must occur after others; **true concurrency is preserved**
- ➢More **realistic** model for concurrent systems

Merits of Formal Methods

- ✅ Precise specifications
  ✅ Early detection of design flaws
  ✅ Mathematical correctness (provable)
  ✅ No ambiguity in meaning
  ✅ Support for automated verification (e.g., theorem provers)
  ✅ Executable specifications = early feedback/prototyping
- Example: In real-time systems, poor requirement specs cause **80% of cost overruns**, so formal methods can save time and money by catching problems early.

Limitations of Formal Methods

- ❌ Hard to learn and use
  ❌ Incompleteness of logic = full correctness not always provable
  ❌ Complex systems = complex specifications (hard to manage)
  ❌ Large mathematical formulas = hard to understand

Balanced Use: Formal + Informal

- Formal specs **do not replace** informal ones—they **complement** them.
- Use **formal methods** to define precise logic and verification steps.
- Use **informal reasoning** for better readability and documentation.
- Combine both for better understanding and correctness.

🔍 **Example:**

Jones (1980) suggests using formal methods to define verification rules, but informal language for argumentation and explanation.

Aspect	**Property-Oriented**	**Model-Oriented**
Focus	Desired properties (axioms)	Direct system behavior (model)
Suitable for	Requirements specification	System design specification
Flexibility	High (many possible implementations)	Low (tight model)
Tools/Examples	Axiomatic, Algebraic	Z, CSP, CCS

Formal methods enhance software quality through **precision**, **verification**, and **early error detection**, though they require **expertise and effort**.

SOFTWARE ENGINEERING

Module – 4 (SOFTWARE DESIGN)

CONTENTS

Software Design Phase Overview

During the **software design phase**, the main goal is to **transform customer requirements** (as written in the **SRS – Software Requirements Specification** document) into a **design document** that will guide implementation.

- The **design process** starts with the **SRS document** and ends with a **complete design document**.
- This process is typically shown **schematically** (as in Figure 4.1) to illustrate how input requirements flow into structured design output.
- The **design document** acts as a **blueprint** for developers, detailing how the system should be built.
- It should be **detailed and precise enough** so that developers can directly use it to **write code** in the next phase (the **coding or implementation phase**).

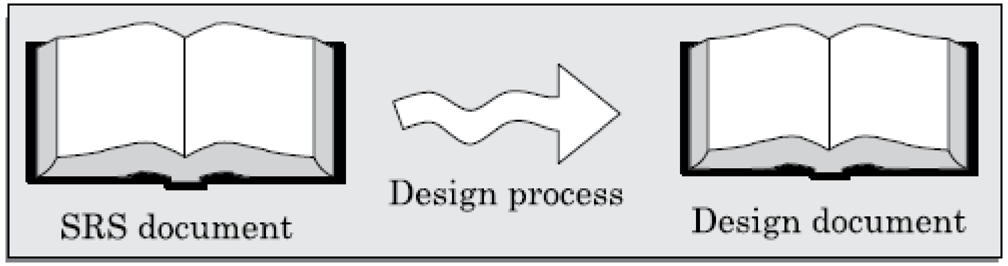


FIGURE 4.1 The design process.

4.1 Overview of the Design Process

The **design process** transforms the **SRS (Software Requirements Specification)** document into a **design document**. This involves breaking down the system into modules and specifying their behavior, structure, relationships, and implementation approach.

4.1.1 Outcome of the Design Process

The **output** of the design phase includes:

- **Modules**:
  - Each module contains related functions and shared data.
  - Each performs a specific task (e.g., student registration module in academic software).
- **Control Relationships**:
  - These are function calls between modules.
  - Must be identified clearly.
- **Module Interfaces**:
  - Specifies what data is exchanged when one module calls another.
- **Data Structures**:
  - Each module must have appropriate structures to store and manage its internal data.
- **Algorithms**:
  - Designed for each function, with focus on correctness, efficiency (time and space).

- Design documents are created through multiple iterations and reviewed to ensure they satisfy the SRS.

4.1.2 Classification of Design Activities

Design is not a one-step process. It involves two main stages:

- **High-Level (Preliminary) Design**:
  - Also called **software architecture**.
  - Breaks down the system into independent, cohesive modules with low coupling.
  - Represented using:
    - **Structure charts** (for procedural design).
    - **UML diagrams** (for object-oriented design).
  - Focuses on **module hierarchy**, **control relationships**, and **interfaces**.

Detailed Design:

- Follows high-level design.
- Results in a **Module Specification (MSPEC)**.
- Specifies each module's:
  - Internal data structures
  - Algorithms
- Detailed enough for programmers to begin **coding**.

This text focuses mainly on high-level design and not on MSPECs.

4.1.3 Classification of Design Methodologies

- Design methodologies can be grouped into:
- **Procedural Design**:
  - Based on functions and procedures.
- **Object-Oriented Design**:
  - Based on objects, classes, and interactions.
- Both are fundamentally different and will be studied in later chapters.

Design Does Not Have One Unique Solution

- Even with the same method, different designers may produce different designs due to subjective decisions.
- A good design is chosen by comparing **alternative solutions**.
- Key question: *How to judge a good design?* (Discussed later)

Analysis vs. Design

Aspect	Analysis	Design
Goal	Understand and model requirements	Plan implementation
Output	Generic, abstract models	Detailed, implementable models
Focus	What the system must do	How the system will do it
Tools (Procedural)	Data Flow Diagrams (DFD)	Structure Charts
Tools (OOP)	UML diagrams	UML diagrams
Detail	Abstract (not implementable)	Concrete (ready for coding)

- **The design process refines the requirements into a structured blueprint for implementation, passing through high-level and detailed stages, influenced by chosen methodologies and involving creative, subjective decisions.**

4.2 How to Characterize a Good Software Design

- There is **no universal definition** of a "good" software design, as it can vary depending on the **type of application**.
- For **embedded systems**, minimizing memory size might be more important than understandability.
- In other systems, **maintainability and clarity** might be more crucial.
- Even though design quality criteria differ across applications and among designers, **most experts agree** on four **essential qualities** of a good software design:

✅ Key Characteristics of Good Design:

- **Correctness** – Implements all specified system functionalities correctly.
- **Understandability** – Easy to read, follow, and comprehend.
- **Efficiency** – Uses system resources (time, memory, CPU) optimally.
- **Maintainability** – Easy to change or update after release.

4.2.1 Understandability: A Major Concern

- When multiple correct design options exist, the **most understandable one** is generally the best.
- **Why is understandability important?**
- Complex systems **exceed human cognitive limits**, making them hard to implement and maintain.
- **60% of software lifecycle cost** goes into maintenance—an understandable design significantly reduces this.
- Poor understanding leads to more **bugs**, **high development cost**, and **lower reliability**.

How to Improve Understandability?

Two key principles help:

  - **Abstraction** – Hides unnecessary details to simplify.
  - **Decomposition** – Breaks down complex systems into smaller parts.

- These principles lead to **modular** and **layered** designs.

🔹 Modularity

- A **modular design** breaks the problem into independent or loosely connected **modules**.
- Follows the **"divide and conquer"** principle.
- Easier to understand, develop, test, and maintain each module separately.

Characteristics:

- Modules have **limited interactions**.
- Helps manage complexity.
- Inter-module relationships should be minimal and well-defined.

🧠 ***Note:*** Though we cannot precisely measure modularity, we can assess it using:

- **Cohesion** – how closely related functions within a module are.
- **Coupling** – how dependent one module is on others.

- **A highly modular system has high cohesion and low coupling.**

- For example, consider two alternate design solutions to a problem that are represented in Figure 4.2, in which the modules *M*1*, M*2*,* etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow. It can easily be seen that the design solution of Figure 4.2(a) would be easier to understand since the interactions among the different modules is low.

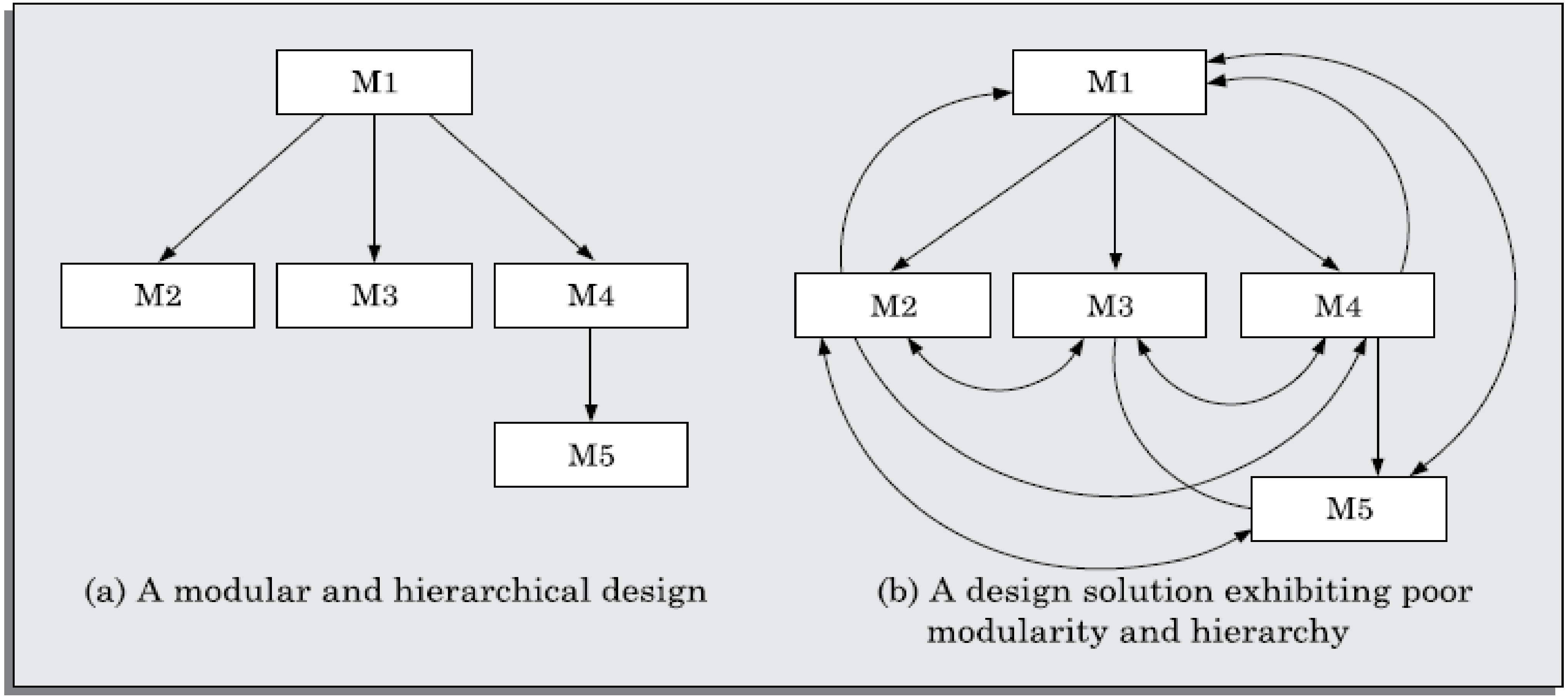

M1
M2
M3
M4
M5
(a) A modular and hierarchical design
M1
M2
M3
M4
M5
(b) A design solution exhibiting poor modularity and hierarchy

FIGURE 4.2 Two design solutions to the same problem.

🔹 Layered Design

- A **layered design** arranges modules in **hierarchical layers**:
    - A module **only interacts with modules directly below it**.
    - Higher layers (like managers) **delegate tasks** to lower layers (like workers).

Benefits:

- Promotes **control abstraction** (lower modules don't know about upper modules).
- Makes debugging easier—failures can be traced by checking only **modules below the point of failure**.
- Enhances **structure**, **clarity**, and **separation of concerns**.

- When module interactions are drawn, a layered design results in a **tree-like structure**.

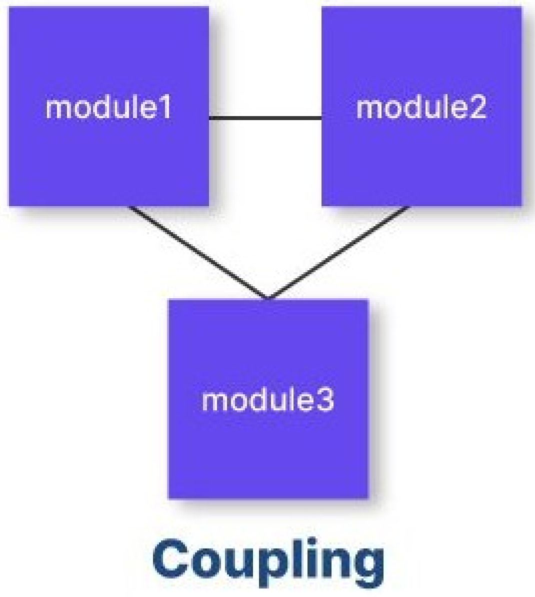
Quality	**Description**
Correctness	Accurately implements all required functionalities.
Understandability	Should be clear, simple, and easy to comprehend.
Efficiency	Optimizes time, space, and computational resources.
Maintainability	Easy to update, modify, or extend.

4.3 COHESION AND COUPLING

🔹 Overview

A **good software design** relies on **effective decomposition** of the problem into modules.
This is achieved when:

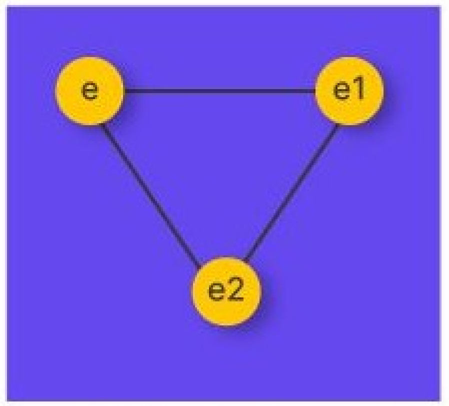
- **Cohesion** is **high** within modules (internal strength).
- **Coupling** is **low** between modules (external dependency).



Coupling

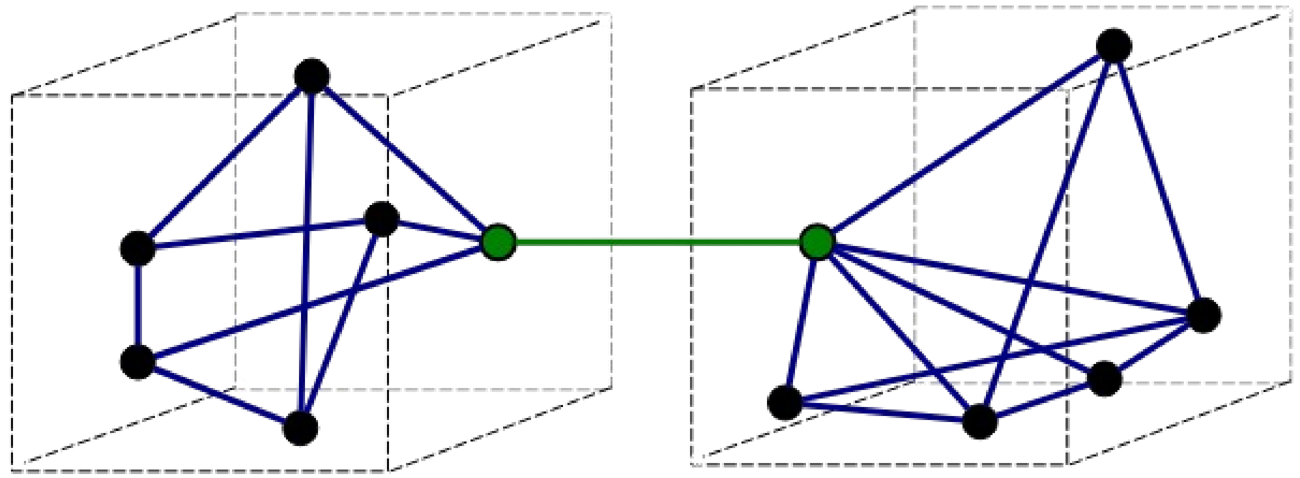
✅ Coupling – *Interdependence between modules*

- Two modules are **tightly coupled** if:
  - * They exchange **large volumes of data** through function calls.
  - * They share **common/global data**.
- Two modules are **loosely coupled** if:
  - * They exchange **few or no data items**.
  - * Use only **simple data types** (like integers, floats).

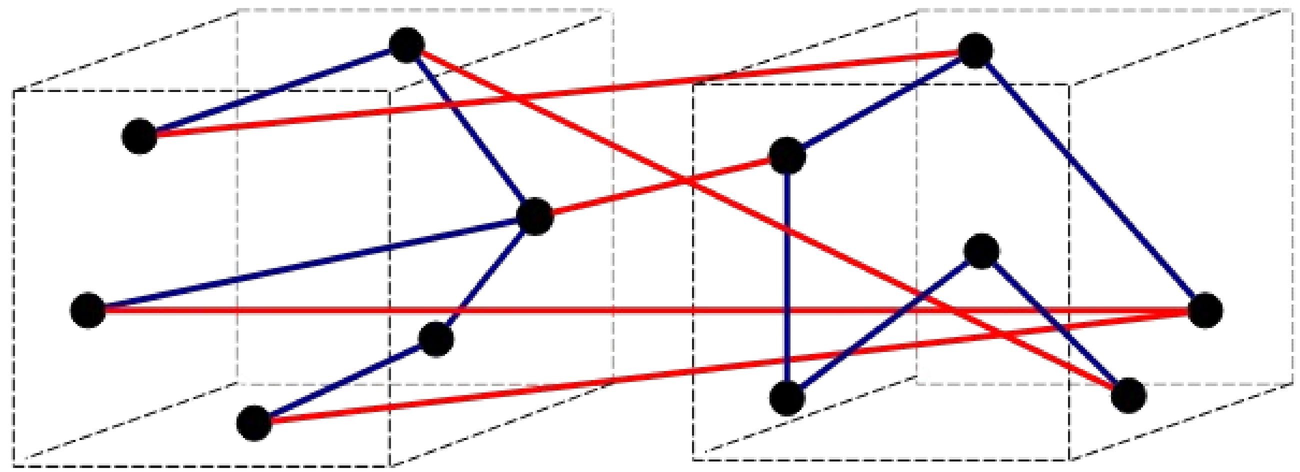


Cohesion

4.3 COHESION AND COUPLING



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

✳ Coupling indicates:

- How strongly one module **depends on** another.
- Affects **debugging**, **testing**, and **independent development**.

✅ Cohesion – *Internal strength of a module*

Cohesion measures how closely the **functions within a module work together** to achieve a **single purpose**.

- A **highly cohesive module** has related functions working toward one goal.
- A **low cohesive module** has unrelated functions grouped without logic.

✳ Cohesion indicates:

- How well a module is **focused**.
- Higher cohesion = better **modularity**, **reusability**, and **maintainability**.

✅ Cohesion – *Internal strength of a module*

Cohesion measures how closely the **functions within a module work together** to achieve a **single purpose**.

- ➢ A **highly cohesive module** has related functions working toward one goal.
- ➢ A **low cohesive module** has unrelated functions grouped without logic.

✳ Cohesion indicates:

- ➢ How well a module is **focused**.
- ➢ Higher cohesion = better **modularity**, **reusability**, and **maintainability**.

🔹 Functional Independence

- A module is **functionally independent** when it:
  - Performs a **single, well-defined task**.
  - Has **minimal interaction** with other modules.

✳ Benefits of Functional Independence:

- **Error isolation**: Bugs don’t spread across modules.
- **Ease of reuse**: Self-contained modules are reusable.
- **Understandability**: Modules can be understood in isolation.

4.3.1 Classification of Cohesion (from worst to best):

Cohesion Type	**Definition**
🔴 Coincidental	Functions are unrelated and grouped arbitrarily (worst). Example: One module handling books and librarian leave.
🟠 Logical	Functions perform similar types of tasks (e.g., all print functions), but not for a common goal.
🟡 Temporal	Functions executed at the same time (e.g., during startup/shutdown).
🟡 Procedural	Functions executed in sequence, but for different purposes.
🟢 Communicational	Functions operate on the same data structure.
🟢 Sequential	Output of one function is input to the next.
🟢 Functional	All functions work together to accomplish one single task (best).

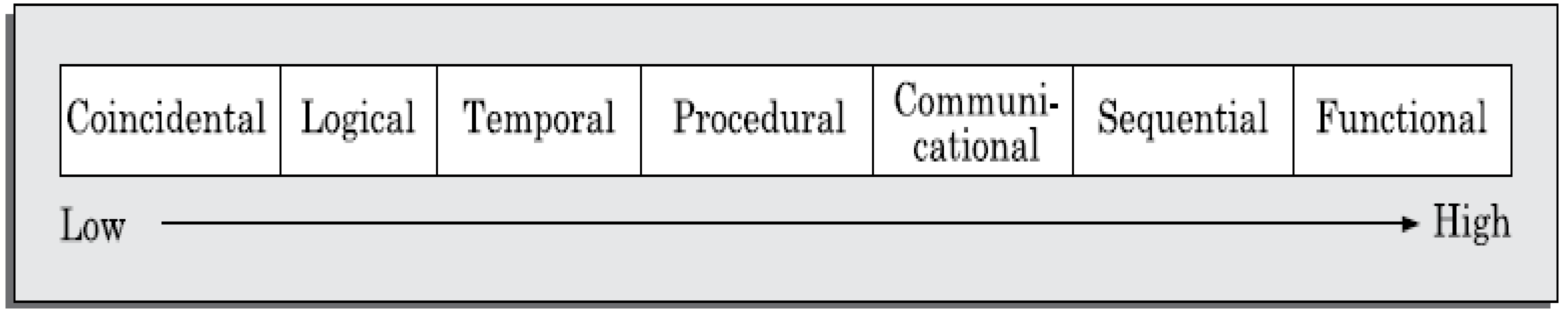
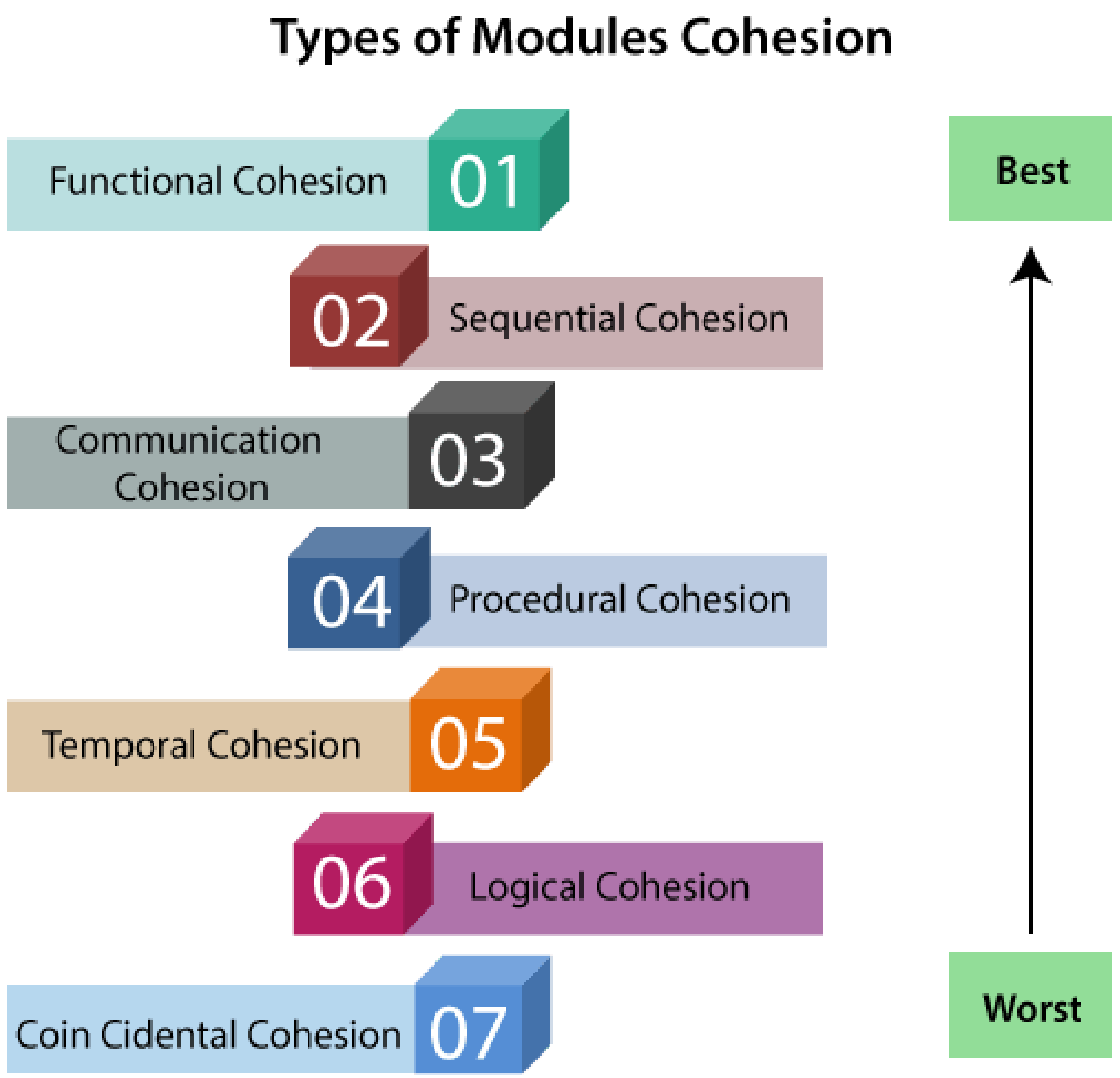
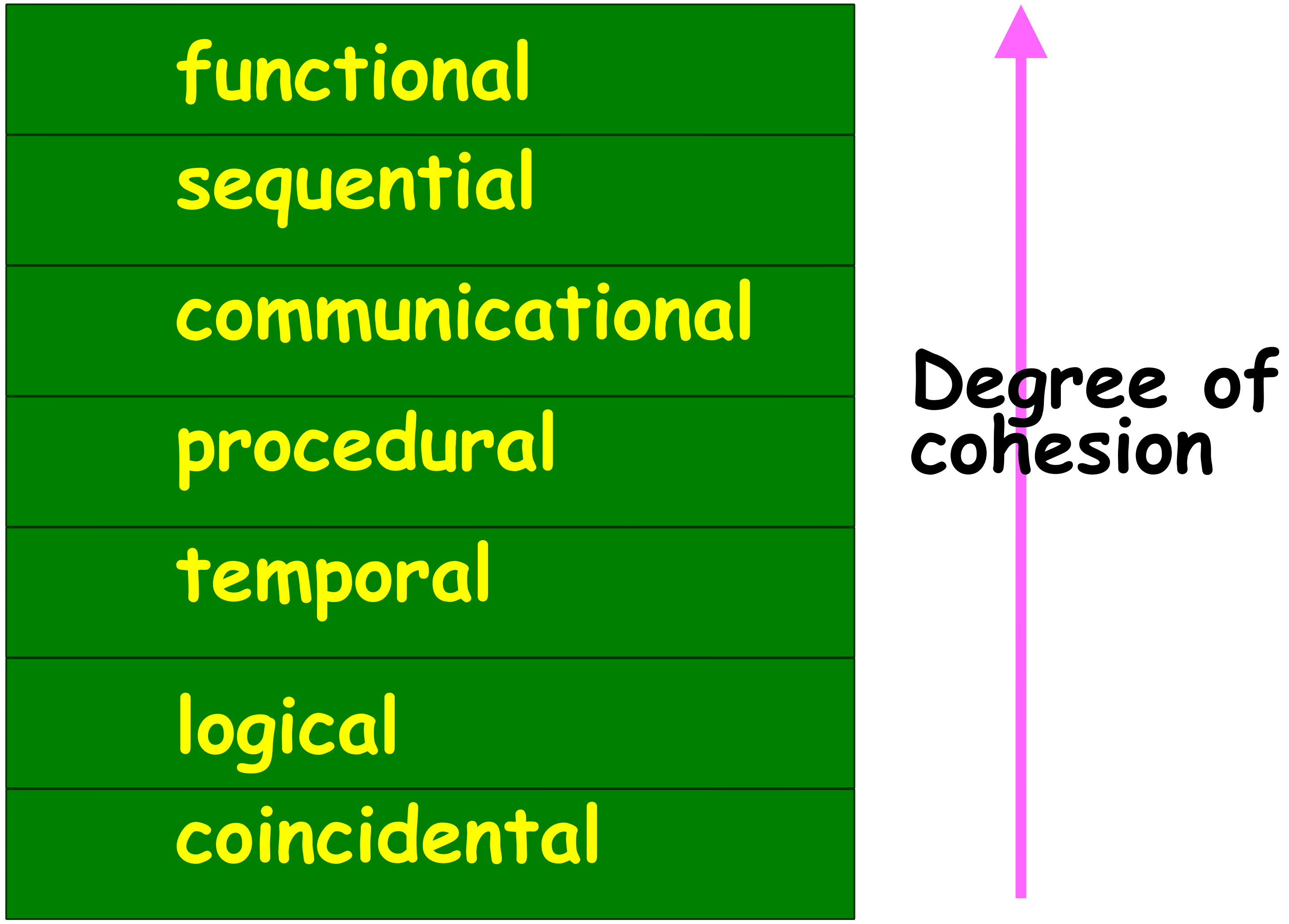


FIGURE 4.3 Classification of cohesion.

4.3.1 Classification of Cohesion (from worst to best):



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.

Coincidental Cohesion - example

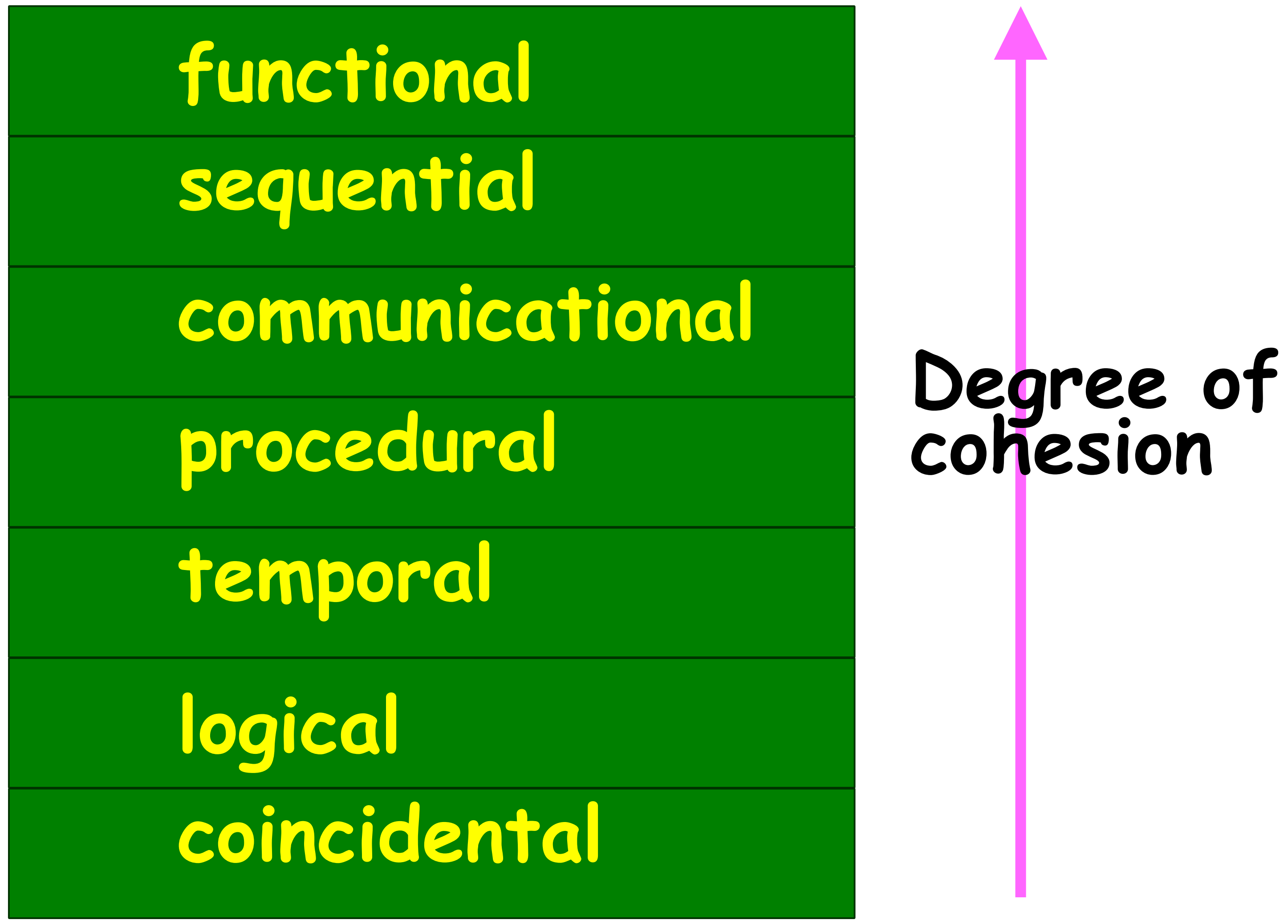
```
Module AAA{

Print-inventory();

Register-Student();

Issue-Book();
};
```

Classification of Cohesiveness



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.

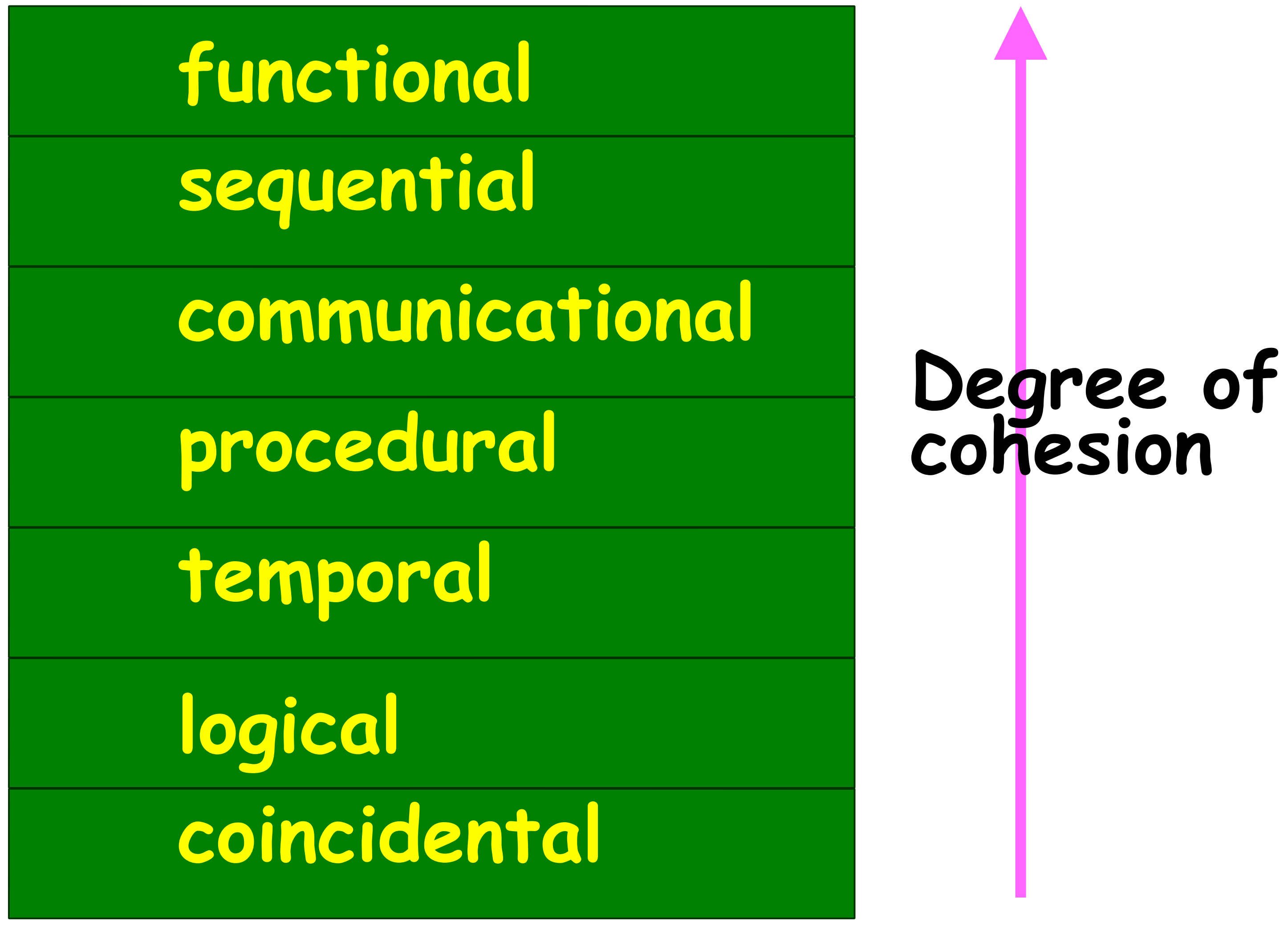
Logical Cohesion

```
Module print{
void print-grades(student-file){ ...}

void print-certificates(student-file){...}

void print-salary(teacher-file){...}
}
```

Classification of Cohesiveness



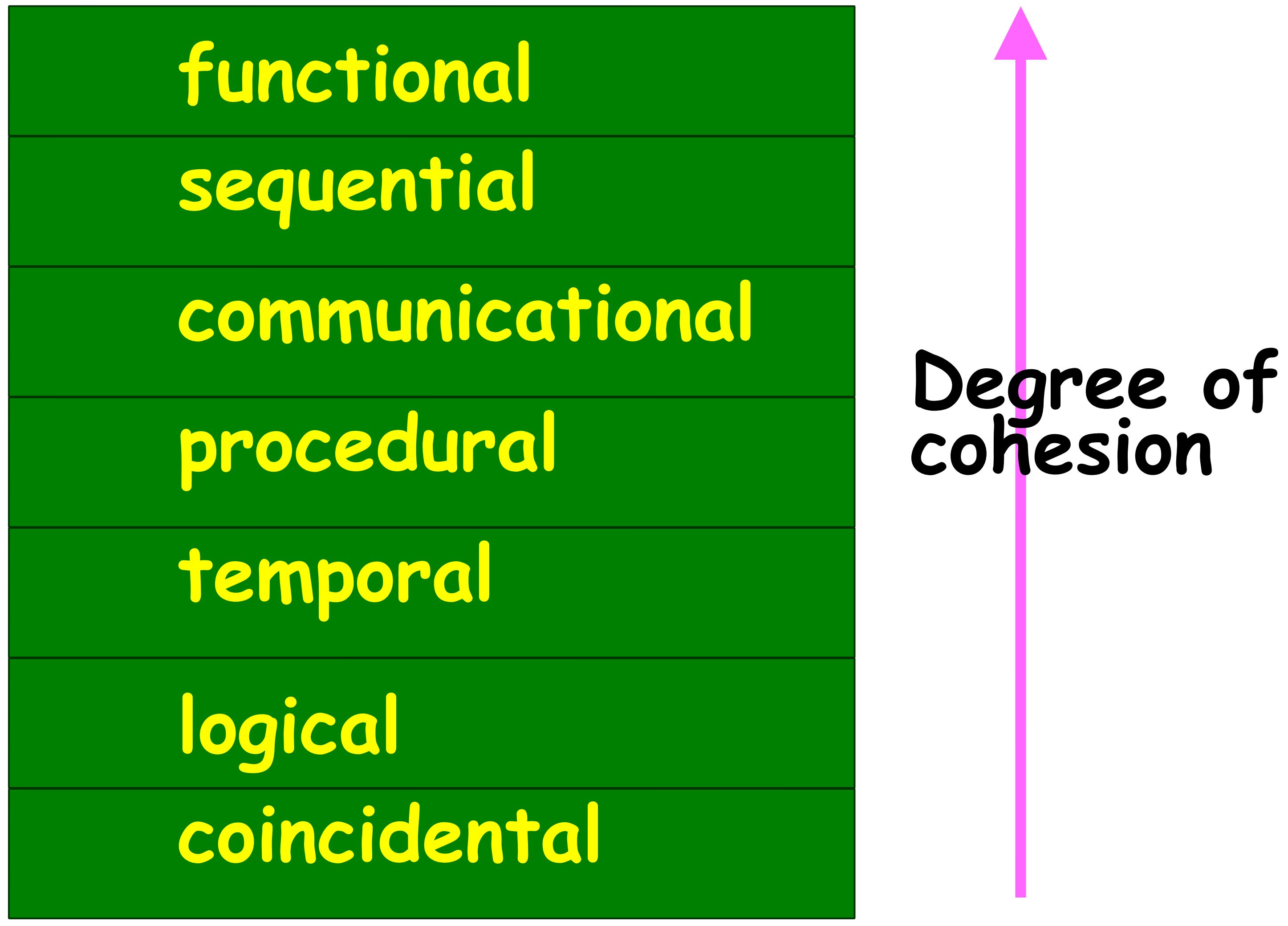
Temporal cohesion

- The module contains tasks so that:
  - all the tasks must be executed in the same time span.
- Example:
  - The set of functions responsible for
    - initialization,
    - start-up, shut-down of some process, etc.

Temporal Cohesion – Example

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

Classification of Cohesiveness



Procedural cohesion

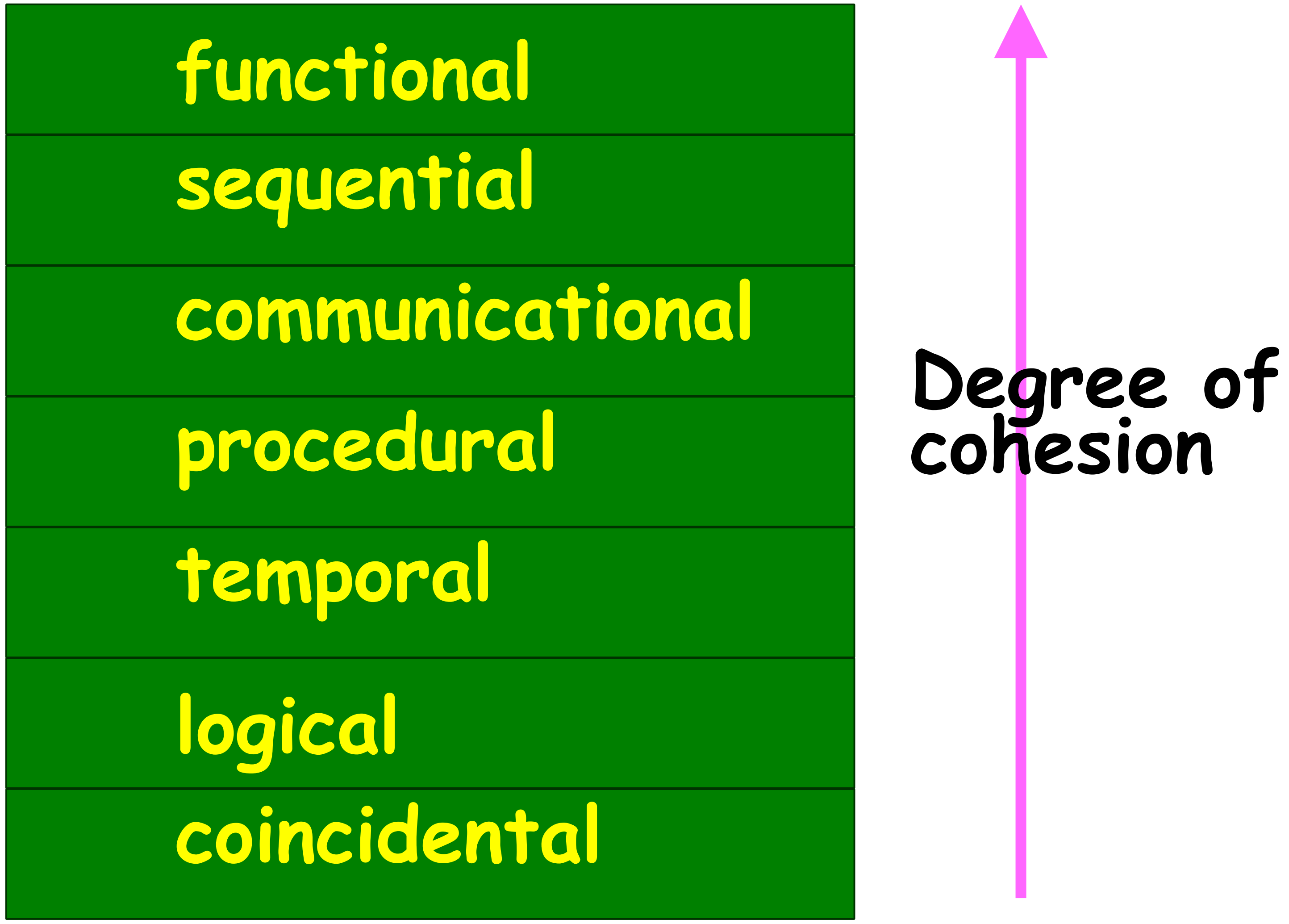
- The set of functions of the module:
  - all part of a procedure (algorithm)
  - certain sequence of steps have to be carried out in a certain order for achieving an objective,
    - e.g. the algorithm for decoding a message.

Procedural Cohesion - example

```
Module AAA{
Login();
Place-order();
Check-order();

Print-bill();
Update-inventory();
Logout();
};
```

Classification of Cohesiveness

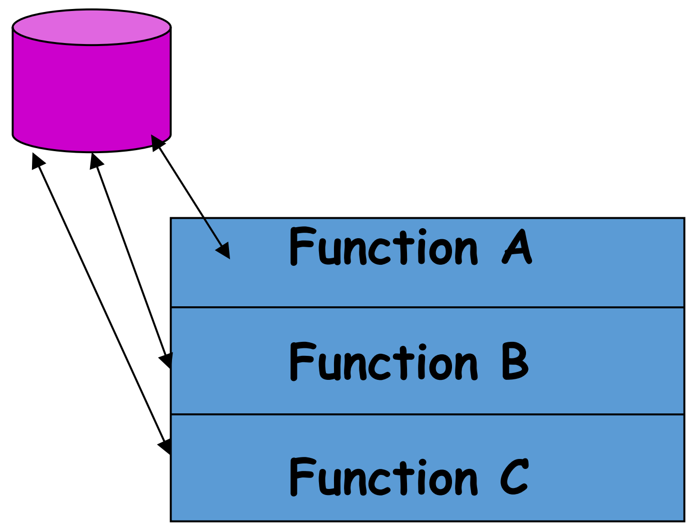


Communicational cohesion

- All functions of the module:
  - reference or update the same data structure,
- Example:
  - The set of functions defined on an array or a stack.

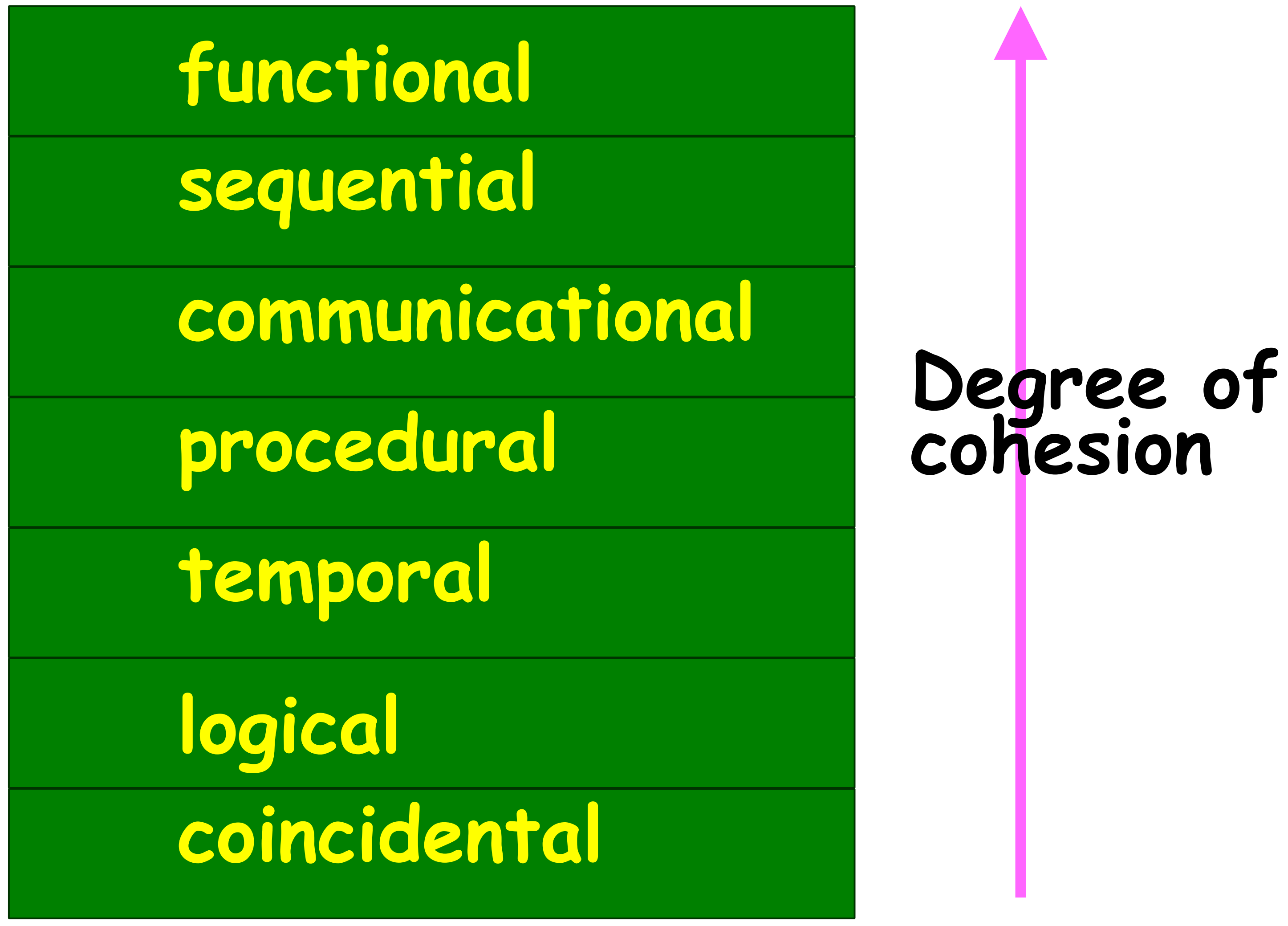
Communicational Cohesion

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



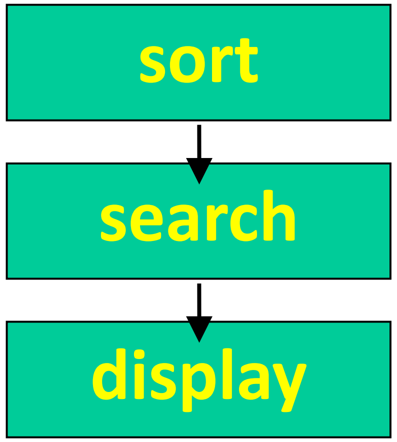
Communicational
Access same data

Classification of Cohesiveness

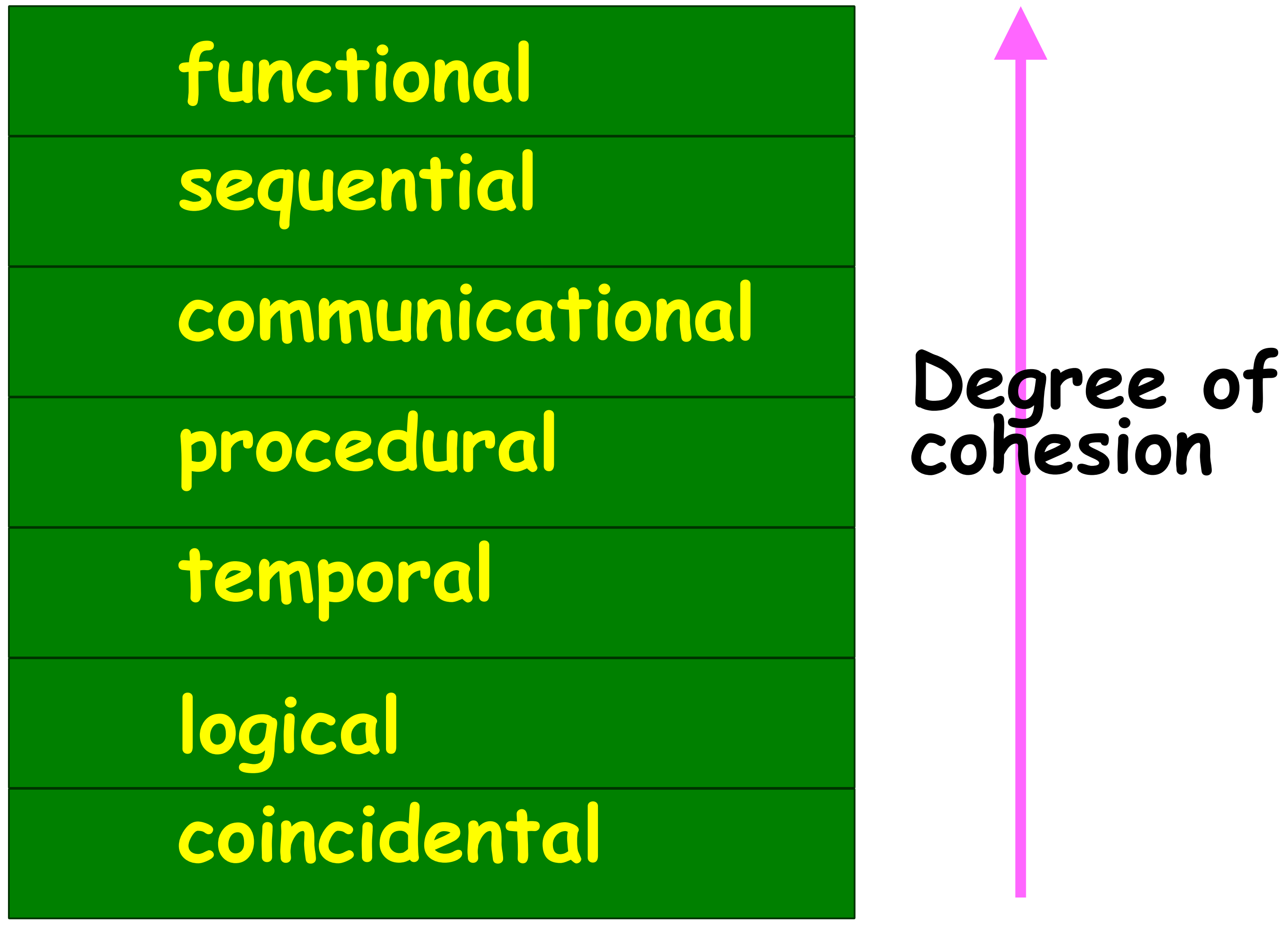


Sequential cohesion

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



Classification of Cohesiveness



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.

Functional Cohesion - example

```
Module AAA{
Issue-Book();

Return-Book();

Query-Book();

Find-Borrower();
};
```

Determining Cohesiveness

- Write down a sentence to describe the function of the module
  - If the sentence is compound (two sentence together)

    Ex: I want to go for a walk, but it started raining
    - 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/setup/shutdown
    - it probably has temporal cohesion.

- An example of a module with coincidental cohesion has been **shown in Figure 4.4(a).** Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

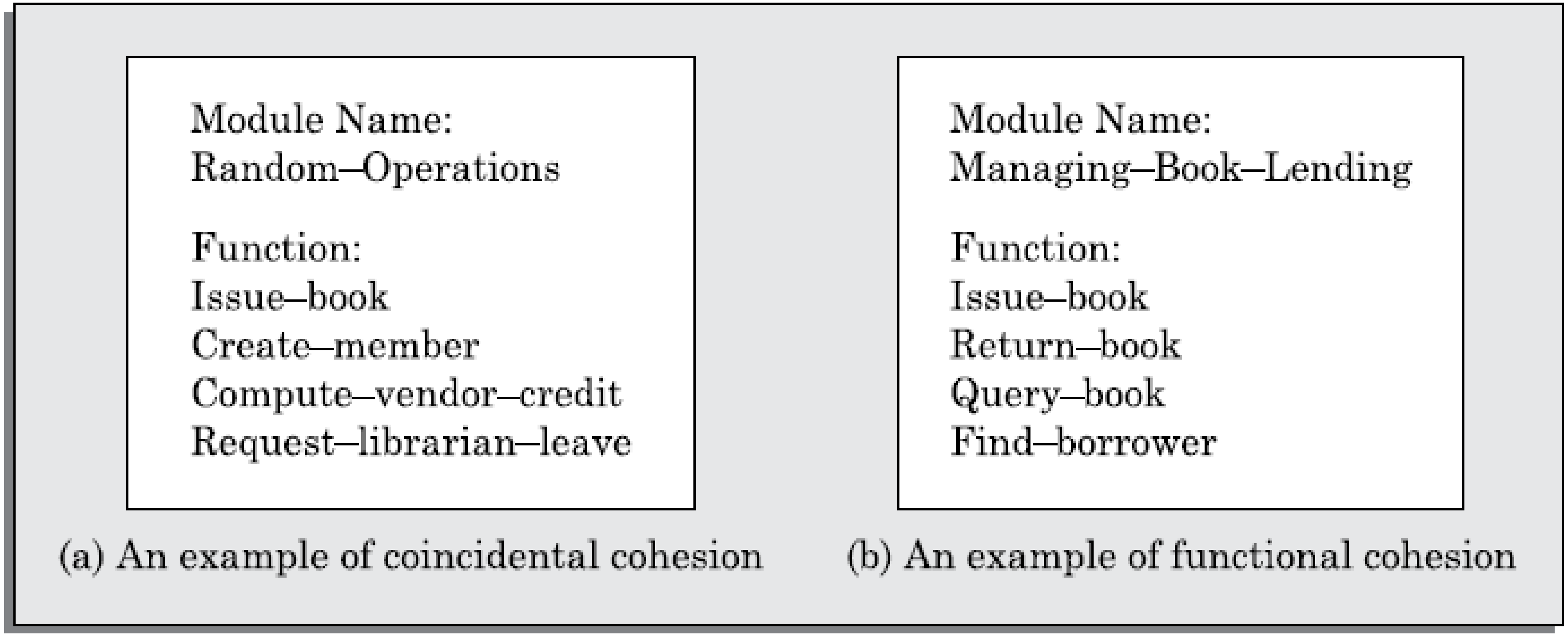


FIGURE 4.4 Examples of cohesion.

4.3.2 Classification of Coupling (from best to worst):

Coupling Type	**Definition**
🟢 Data Coupling	Modules exchange simple data types (e.g., integers). Most desirable.
🟢 Stamp Coupling	Modules exchange composite data types (e.g., structs or records).
🟡 Control Coupling	One module controls the behavior of another (e.g., using flags).
🔴 Common Coupling	Modules share global variables.
🔴 Content Coupling	One module modifies or uses code inside another (e.g., jumps into another module's code). Worst form.

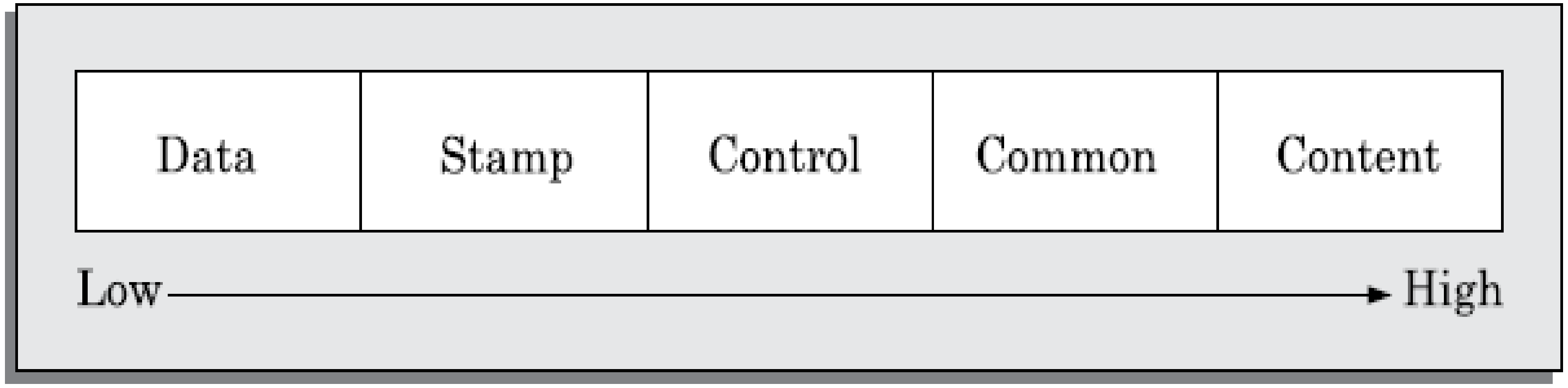
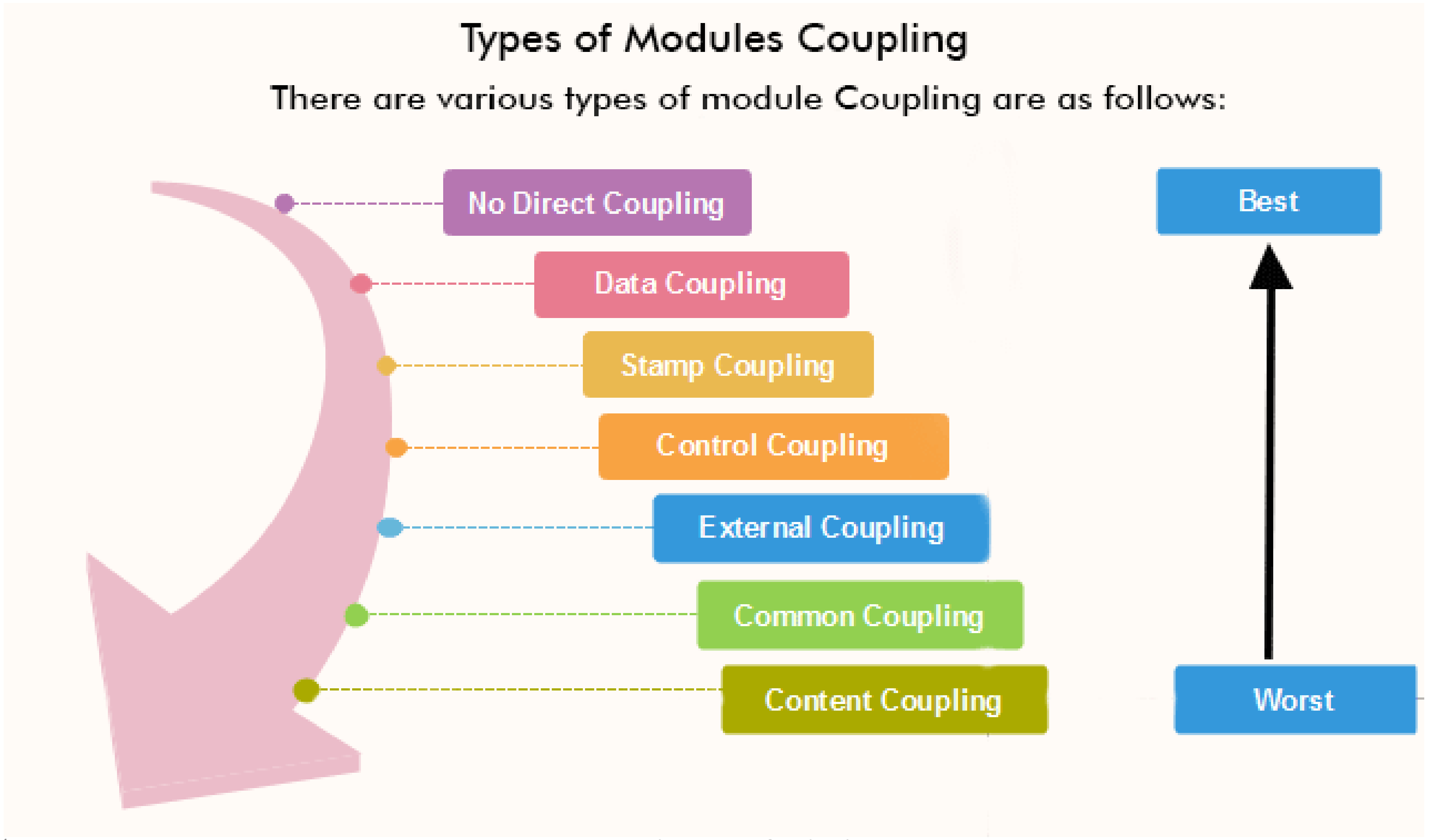
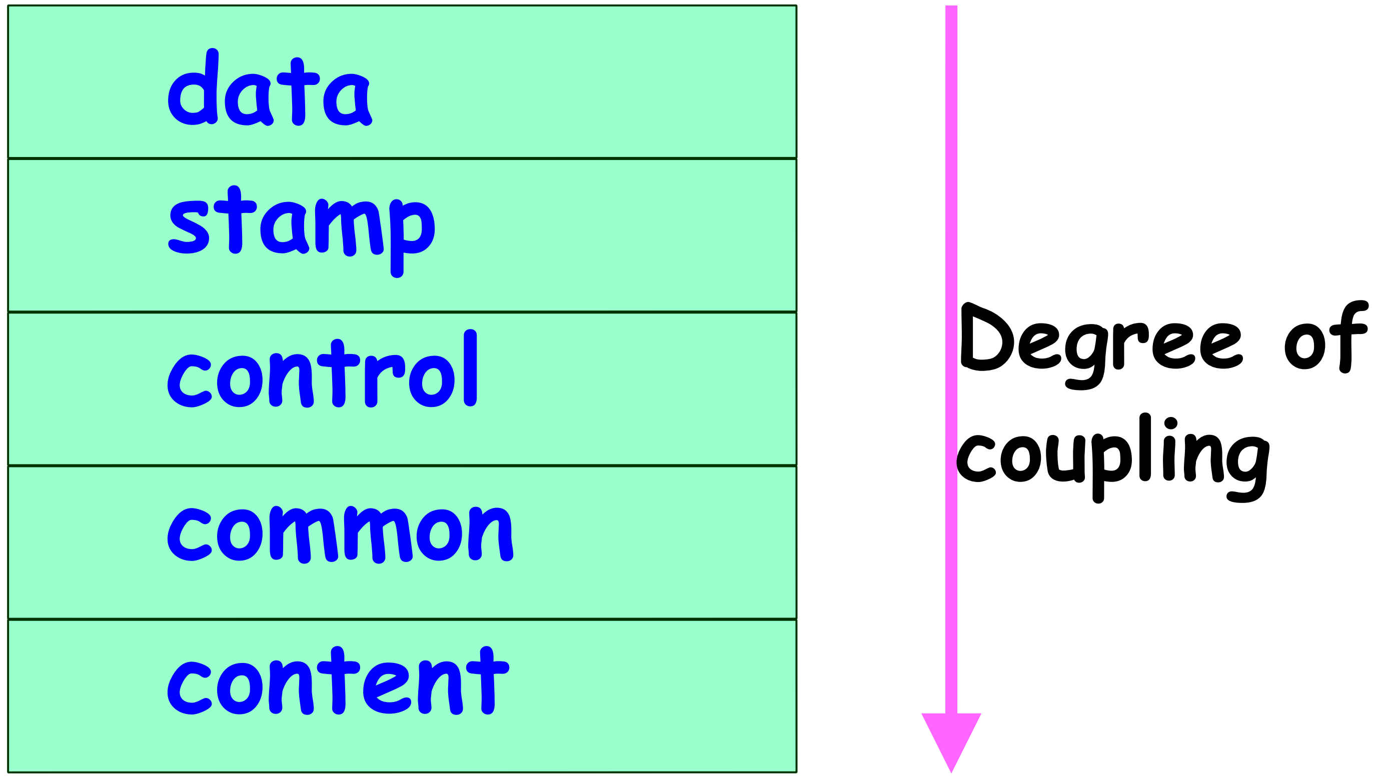


FIGURE 4.5 Classification of coupling.

4.3.2 Classification of Coupling (from best to worst):



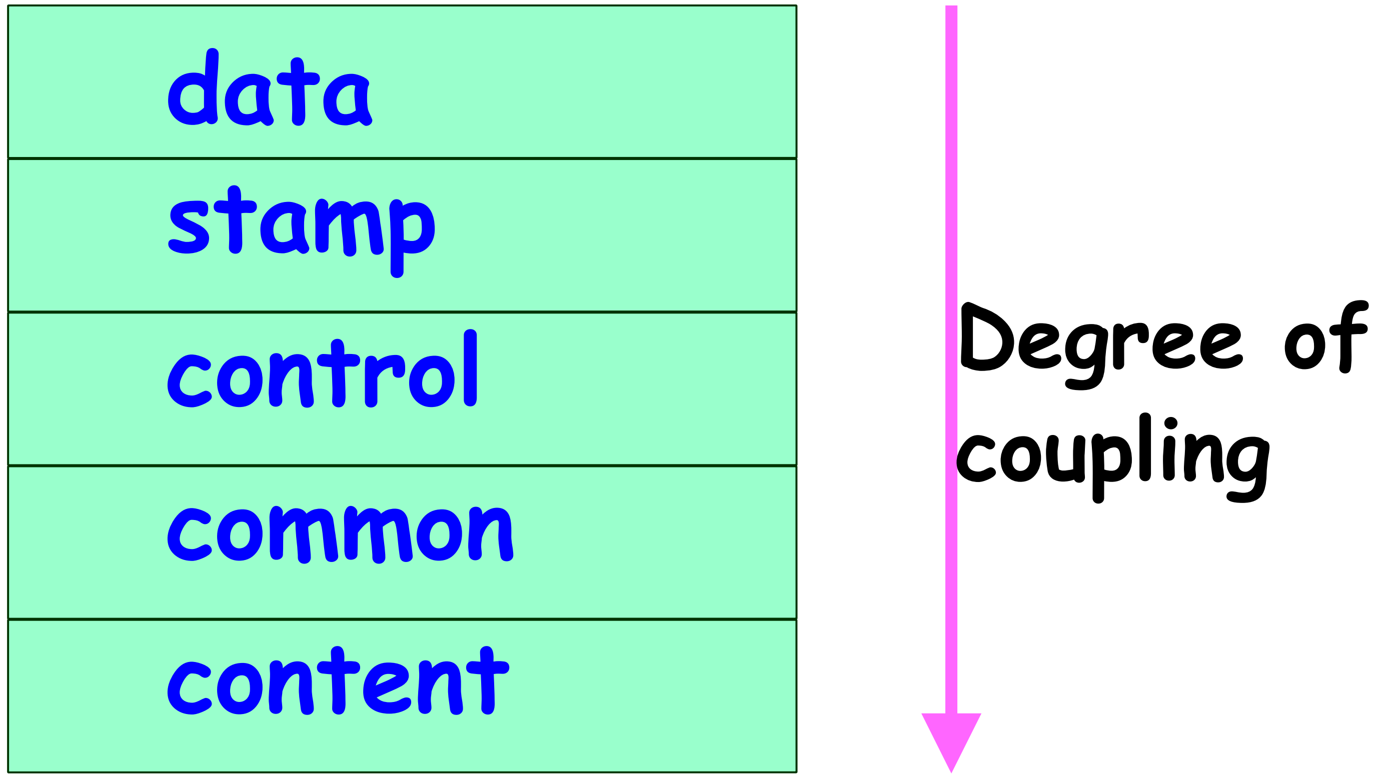
Classes of coupling



Data coupling

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

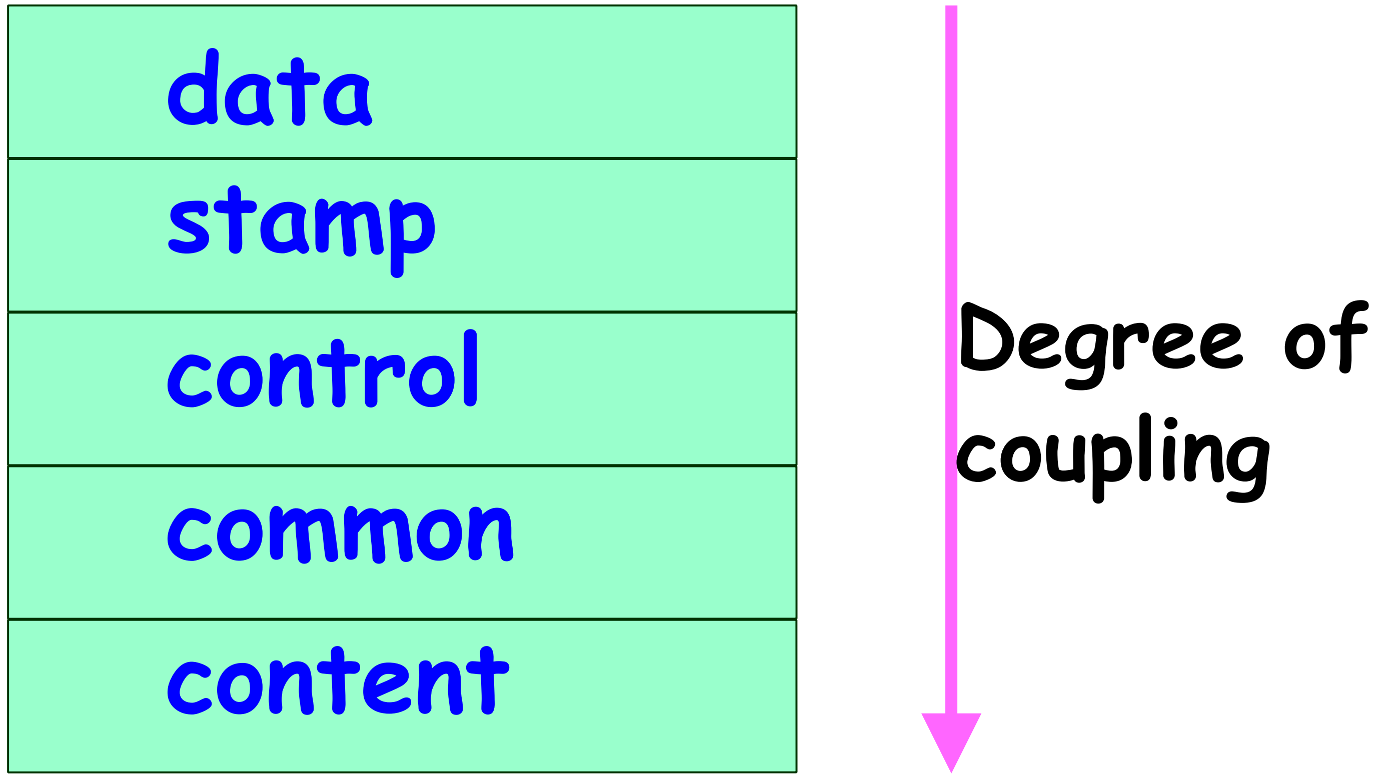
Classes of coupling



Stamp coupling

- Two modules are stamp coupled,
  - if they communicate via a composite data item
    - an array or structure in C.

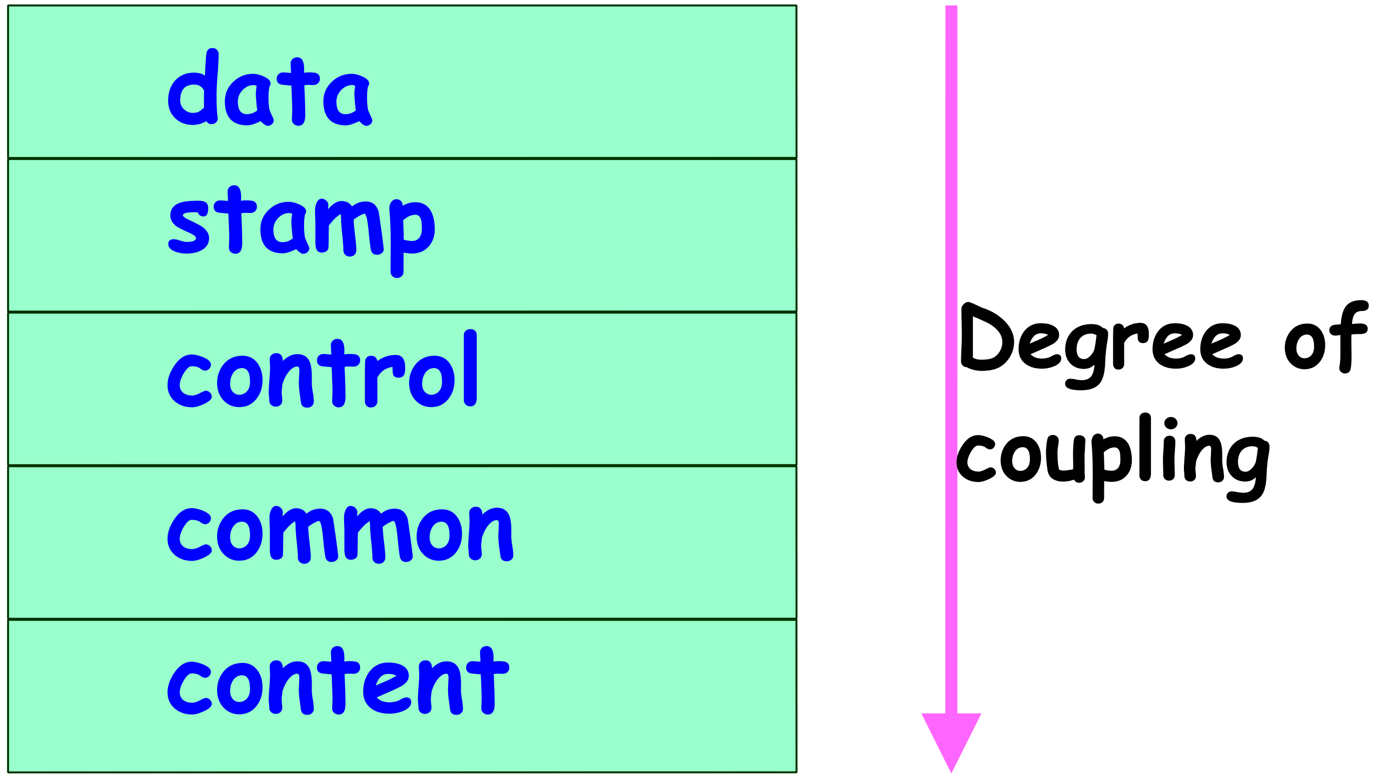
Classes of coupling



Control coupling

- Data from one module is used to direct
  - order of instruction execution in another.
- Example of control coupling:
  - a flag set in one module and tested in another module.

Classes of coupling

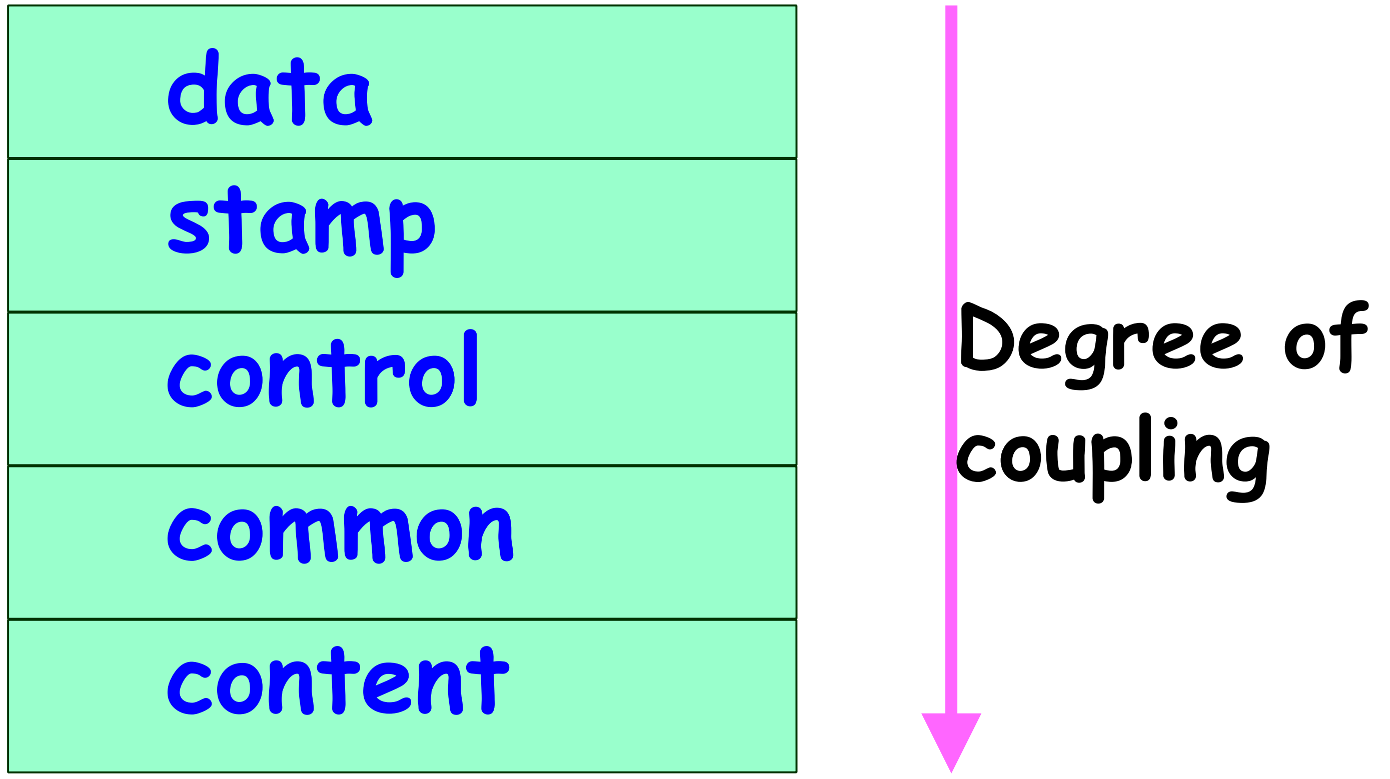


Common Coupling

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

This means, different modules access and modify the same global variables.

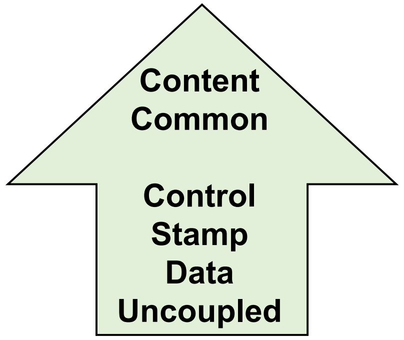
Classes of coupling

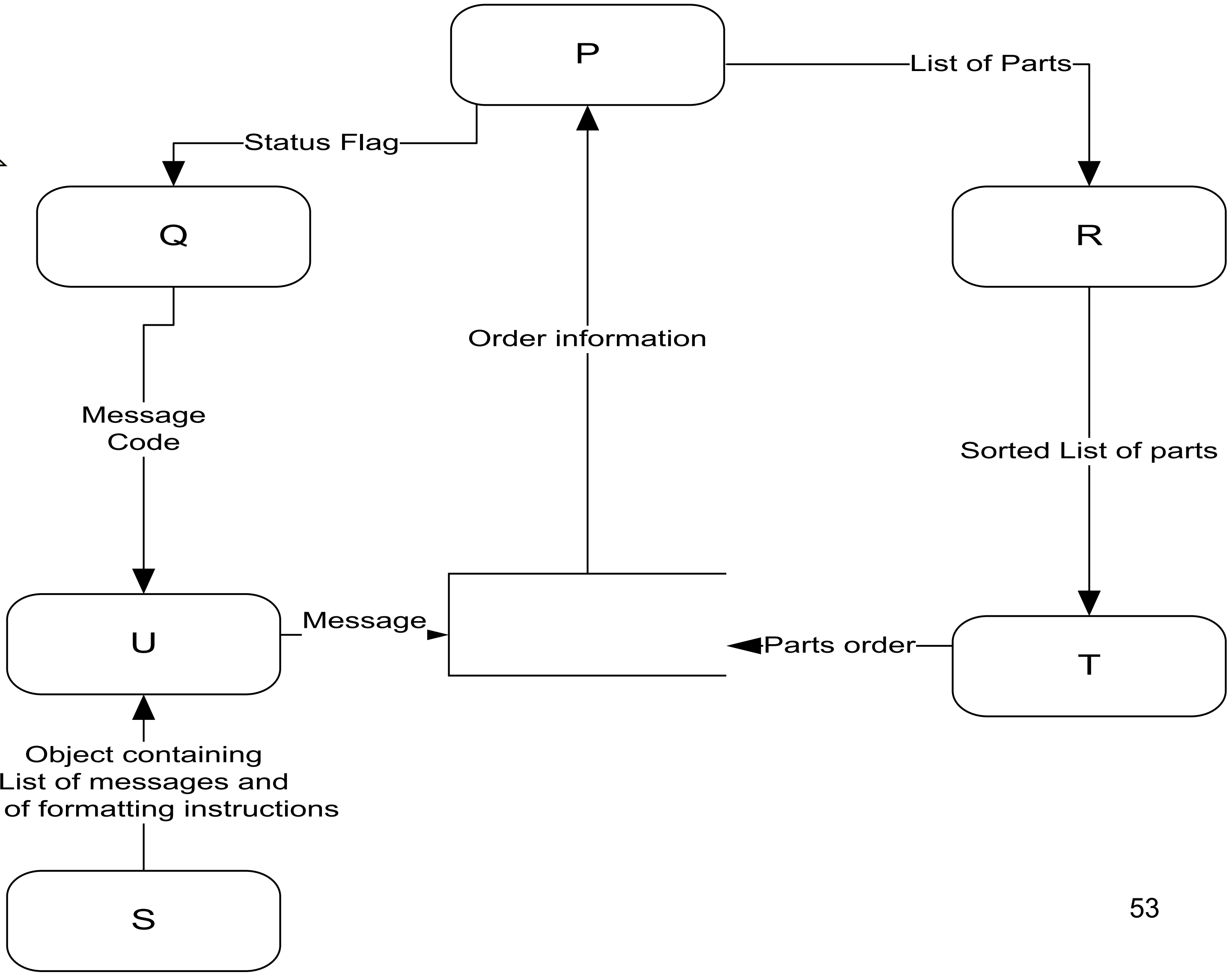


Content coupling

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

Exercise: Define Coupling between Pairs of Modules





Coupling between Pairs of Modules

	Q	R	S	T	U
P					
Q					
R					
S					
T					

Coupling between Pairs of Modules

	Q	R	S	T	U
P	Control	stamp		Common	Common
Q					Data
R					
S					Stamp
T		Stamp			Common

⚠️ **Higher coupling** leads to:

- ➢Complex dependencies
- ➢Reduced modularity
- ➢Difficult debugging and maintenance

Aspect	**Good Practice**	**Poor Practice**
Cohesion	Functional, Sequential	Coincidental, Logical
Coupling	Data, Stamp	Common, Content

A **good design** should strive for:

- **High cohesion**: Each module is focused and meaningful.
- **Low coupling**: Modules are independent and loosely connected.

🔑 **Functionally independent modules are key to better software quality, reusability, maintainability, and debugging.**

4.4 Layered Arrangement of Modules

🔹 What is a Layered Design?

- A **layered design** organizes software modules based on their **control hierarchy**—that is, how modules **call** each other.
- In a **layered structure**, a module can **only call modules in the layer directly below** it. It should **not** call:
  - Modules from the **same layer**, or
  - Modules from **higher layers**.

🔸 Visual Representation:

- A **tree-like diagram** called a **structure chart** (Module - 5) is commonly used to show control hierarchy.

✅ Benefits of a Layered Design

- **Improved Understandability**:
  - To understand any module, you only need to examine the modules it directly uses (i.e., those below it).

- **Easier Debugging**:
  - If a module fails, the issue is usually in one of the **modules it calls** (i.e., **lower layers**).
  - Reduces the time and effort required to isolate errors.

- **Control Abstraction**:
  - Lower-level modules are **hidden** from higher layers.
  - Each module only focuses on its own responsibility and the modules it directly controls.

🔹 Types of Modules in a Layered Design

Layer	Type of Module	Responsibilities
Top Layer	Manager Module	Controls lower modules; delegates tasks
Middle Layers	Intermediate Modules	Performs some tasks and calls further lower modules
Bottom Layer	Worker Modules	Perform complete tasks on their own; don’t call other modules

✅ Important Terminologies

1. **Superordinate and Subordinate Modules**
    - A **superordinate** module controls or calls another module.
    - A **subordinate** module is controlled (called) by another.
2. **Visibility**
    - A module **A** can *see* (i.e., call) module **B** only if **B is in the layer below A**.
3. **Control Abstraction**
    - Higher-layer modules are **not visible** to lower-layer modules.
    - Modules only call the **immediate lower layer**.
4. **Depth and Width**
    - **Depth**: Number of **layers** in the hierarchy.
    - **Width**: Number of **modules at each level**.

- Example: In Figure 4.6(a), the design has depth = 3 and width = 3.

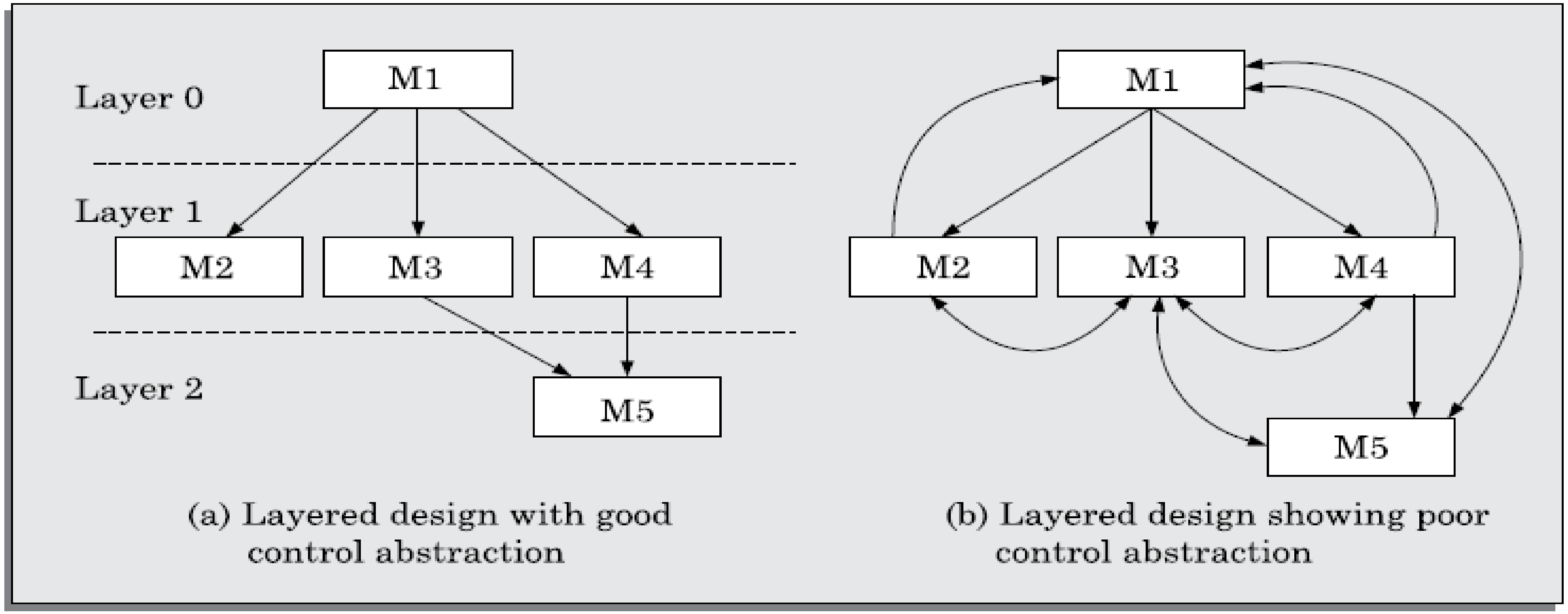

Layer 0
Layer 1
Layer 2
M1
M2
M3
M4
M5
(a) Layered design with good control abstraction
(b) Layered design showing poor control abstraction

FIGURE 4.6 Examples of good and poor control abstraction.

✅ Fan-In and Fan-Out

Term	Meaning	Good/Bad
Fan-Out	Number of modules a module directly calls	Low is good. High fan-out (>7) may indicate poor cohesion.
Fan-In	Number of modules that directly call a given module itself	High is good. Indicates code reuse.

🚫 Non-layered Design (Bad Example)

- All modules **call each other freely**.
- **Harder to debug**, because errors can come from **any module**.
- **No clear hierarchy**, making it harder to understand the structure.

Feature	**Layered Design**	**Non-layered Design**
Structure	Hierarchical, top-to-bottom	Flat, unorganized
Module Calls	Only downward (to lower layers)	Calls can be in any direction
Understandability	Easier	Harder
Debugging	Simplified (trace downwards)	Complex (trace across all modules)
Control Abstraction	Present	Absent
Fan-out	Should be low (≤7)	Often high (bad)
Fan-in	Higher is better (indicates reuse)	May be low

- **A layered module design is a hallmark of a good, maintainable, and scalable software system. It supports abstraction, reuse, and reduces debugging complexity.**

4.5 Approaches to Software Design

There are **two major approaches** to software design:

- **Function-Oriented Design (FOD)** – Traditional, structured approach
- **Object-Oriented Design (OOD)** – Modern, modular approach

- These are **complementary**, not competing. OOD is increasingly used for large-scale systems, while FOD remains a stable, well-established technique.

5.5.1 Function-Oriented Design (FOD)

◆ **Key Characteristics:**

•**Top-Down Decomposition**:

Begin with high-level functions and refine them into smaller sub-functions.

Example: create-new-library-member →

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

•**Centralized System State**:

Shared global data is accessible across multiple functions.

E.g., member-records are accessed by create-member, delete-member, etc.

🔸 Popular Function-Oriented Methods:

- Structured Design (Constantine & Yourdon, 1979)
- Jackson's Structured Design (1975)
- Warnier-Orr Methodology (1977, 1981)
- Step-wise Refinement (Wirth, 1971)
- Hatley and Pirbhai's Methodology (1987)

🔹 5.5.2 Object-Oriented Design (OOD)

- **🔸 Core Concepts:**
  - **Objects** = Data + Methods
  - **Each object** manages its own data (private), accessed only via its methods.
  - **No global data** — data is **distributed** among objects (decentralised state).
  - Objects interact using **message passing**.

🔸 Abstraction via ADTs (Abstract Data Types):

Concept	**Meaning**
Data Abstraction	Internal data details are hidden; access via defined methods only.
Data Structure	Collection of primitive data items arranged logically.
Data Type	Anything that can be instantiated (e.g., int, float, or a class)

Benefits of Using ADTs in OOD:

- **Encapsulation (Data Hiding)**: Errors are isolated; access is controlled via methods.
- **High Cohesion + Low Coupling**: Each object is modular and self-contained.
- **Improved Understandability**: Abstraction simplifies complexity.

🔸 Example Comparison – Fire Alarm System

- **Function-Oriented Design**
  - **Global Data**:

```
BOOL detector_status[MAX_ROOMS];
int detector_locs[MAX_ROOMS];
```

- **Functions**:
  interrogate_detectors(), ring_alarm(), reset_sprinkler()…

Object-Oriented Design

- **Classes**:
  - class detector { status, location, neighbours; methods: sense_status() }
  - class alarm { location, status; methods: ring_alarm(), reset_alarm() }
  - class sprinkler { location, status; methods: activate_sprinkler() }

Feature	**Function-Oriented Design**	**Object-Oriented Design**
Basic Unit	Function / Module	Object (instance of a class)
Data Access	Global/shared across functions	Private inside objects
State Storage	Centralised	Distributed across objects
Function Grouping	Based on higher-level tasks	Based on data they operate on
Use of Abstraction	Limited	Extensive (via ADTs)
Reusability and Modularity	Lower	Higher
Examples in Real World	issue-book()	book object with issue() method

Note:

- **OOD is not limited to object-oriented languages.**
  - It can be implemented in procedural languages like C, though with more effort.
- Often, **both approaches are used together**:
  - Use **OOD** for overall architecture and object definitions.
  - Use **FOD** (top-down) within individual class methods for internal logic.

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

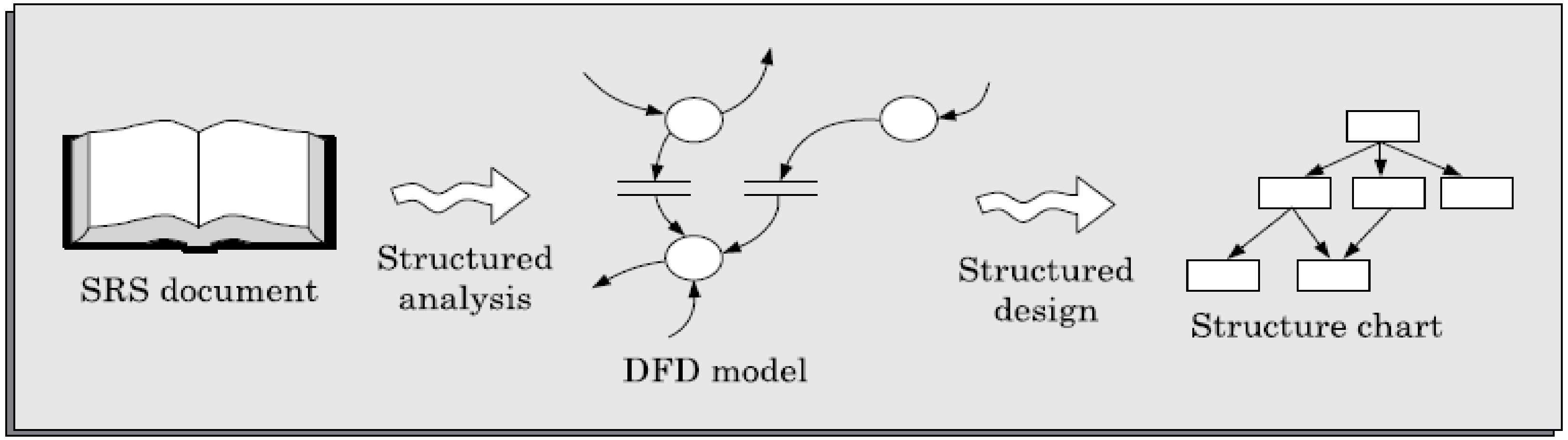


FIGURE 5.1 Structured analysis and structured design methodology.

🔄 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

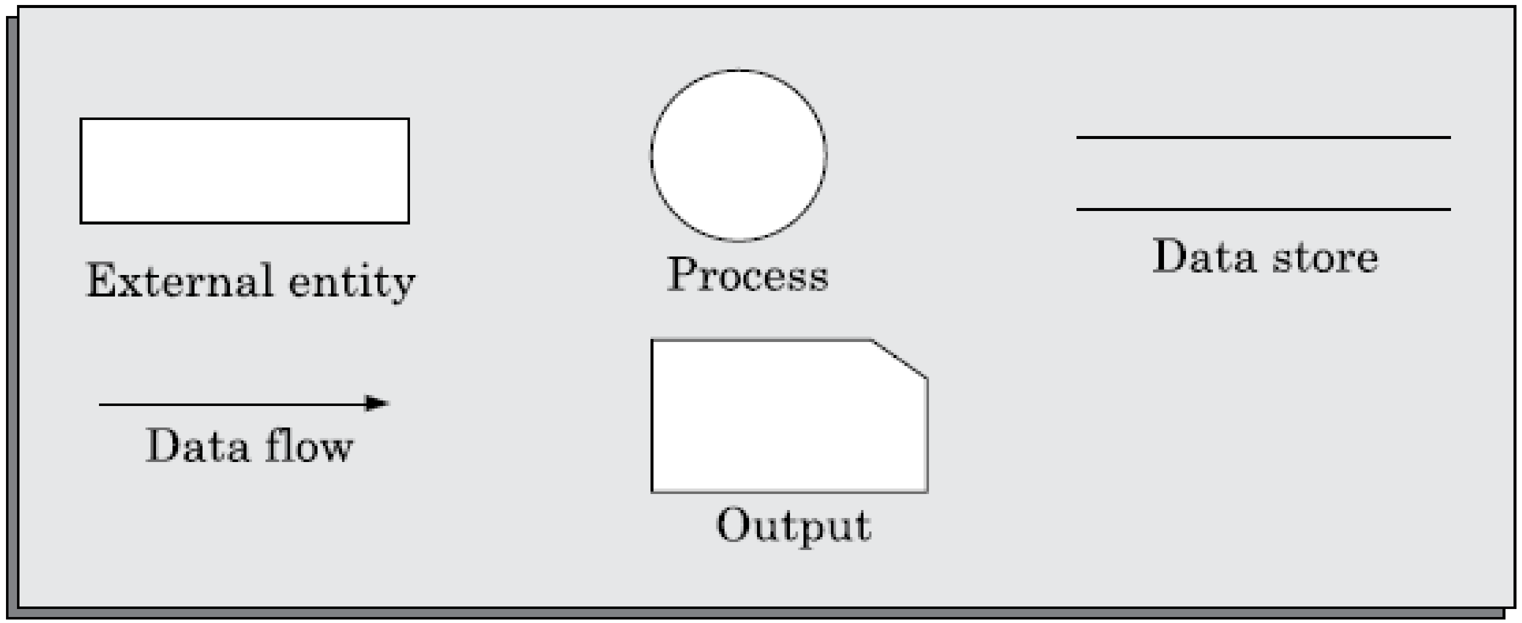


FIGURE 5.2 Symbols used for designing DFDs.

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
‖ (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

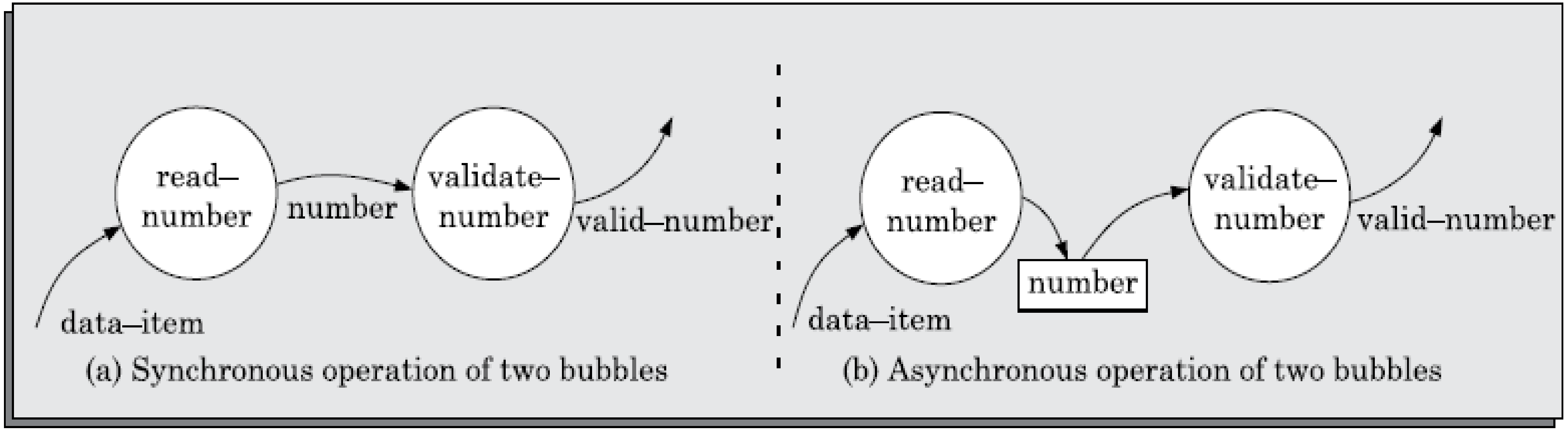


FIGURE 5.3 Synchronous and asynchronous data flow.

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.

<u>Level 1 DFD</u>:

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

<u>Level 2 and below</u>:

- 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.**

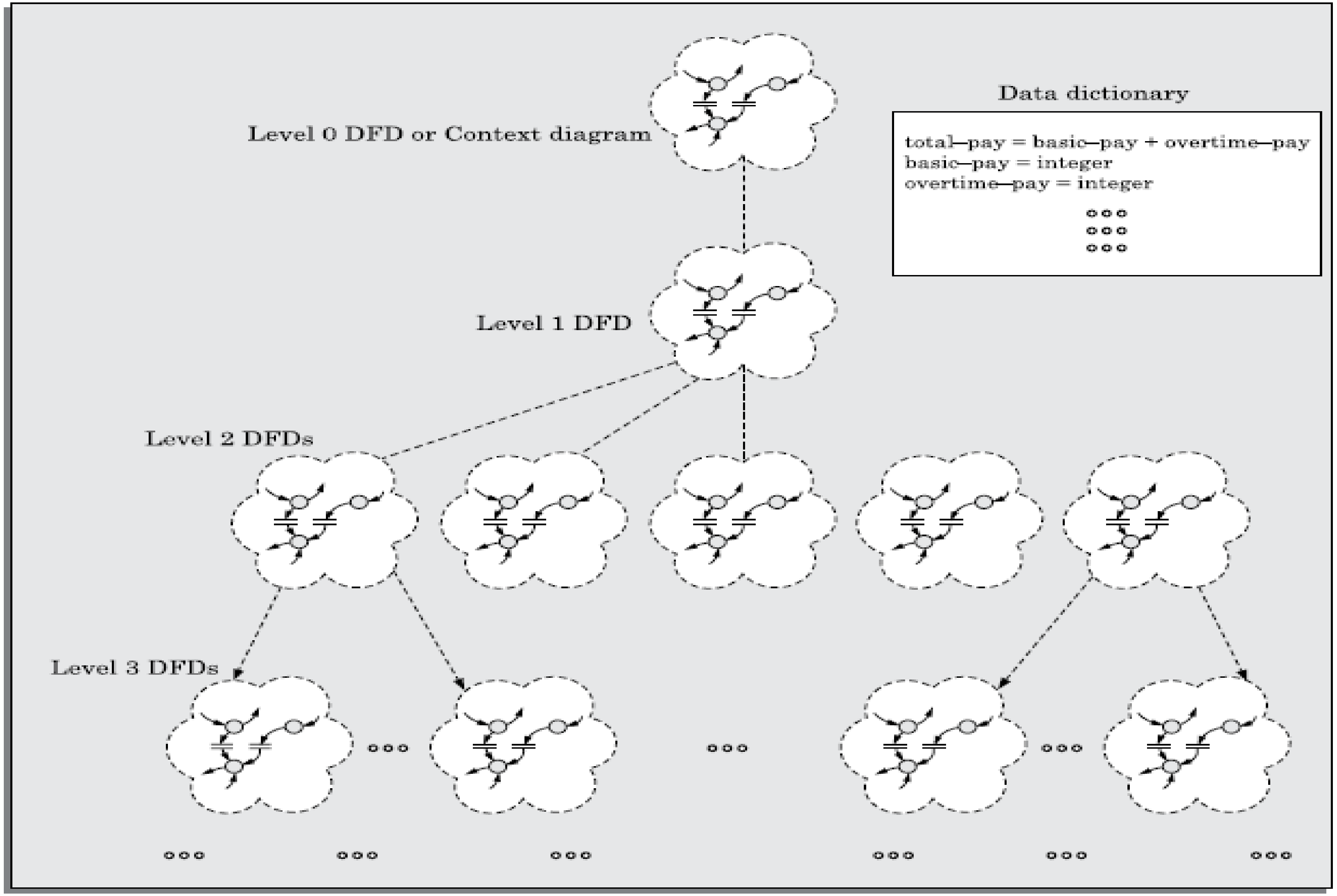


FIGURE 5.4 DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

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

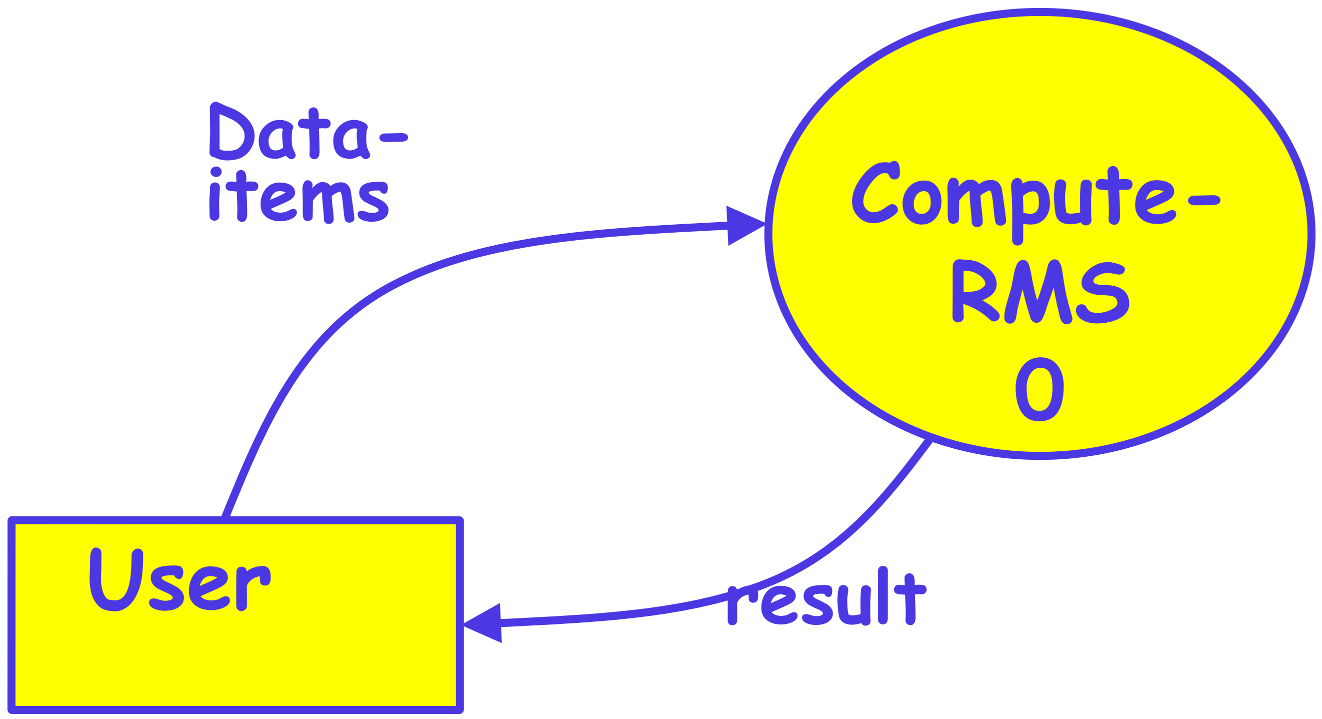
Level 2 DFD (for "Calculate RMS"):

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

<u>Data Dictionary for Example</u>

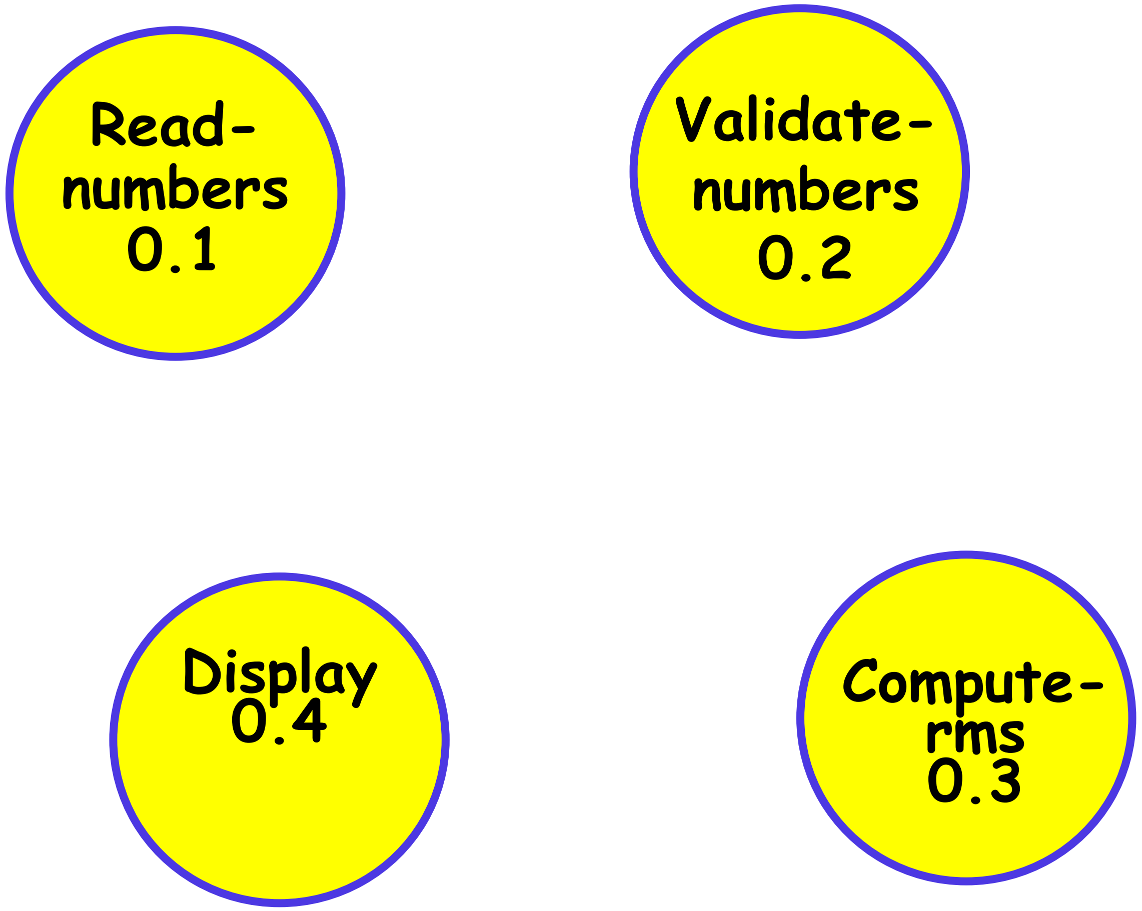
Data Item	**Description**
data-items	3 integers
rms	Floating-point RMS value
valid-data	Same as data-items if valid
a, b, c	Individual integers
asq, bsq, csq	Squares of inputs
msq	Mean of squares

Example 1: RMS Calculating Software

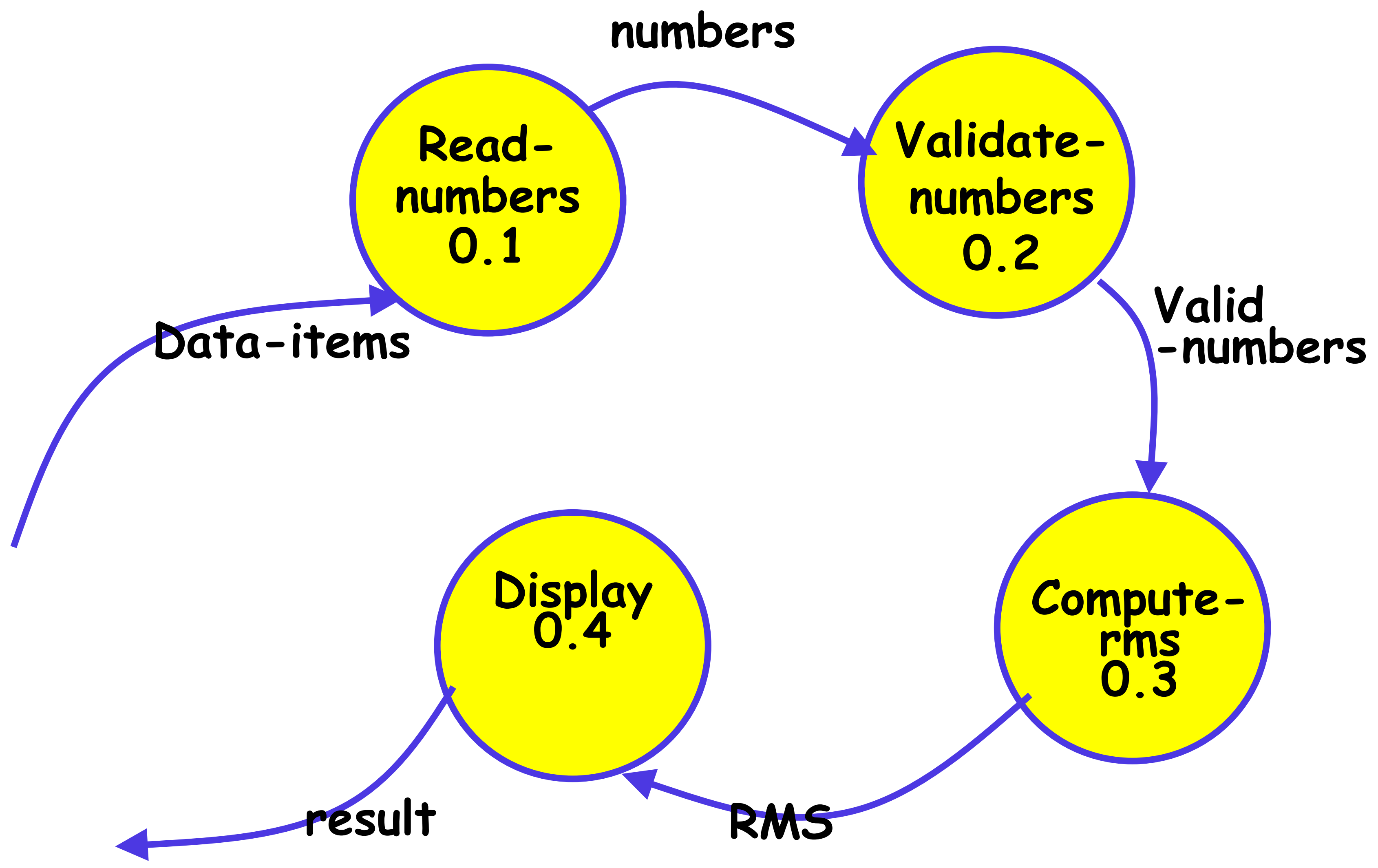


Context Diagram

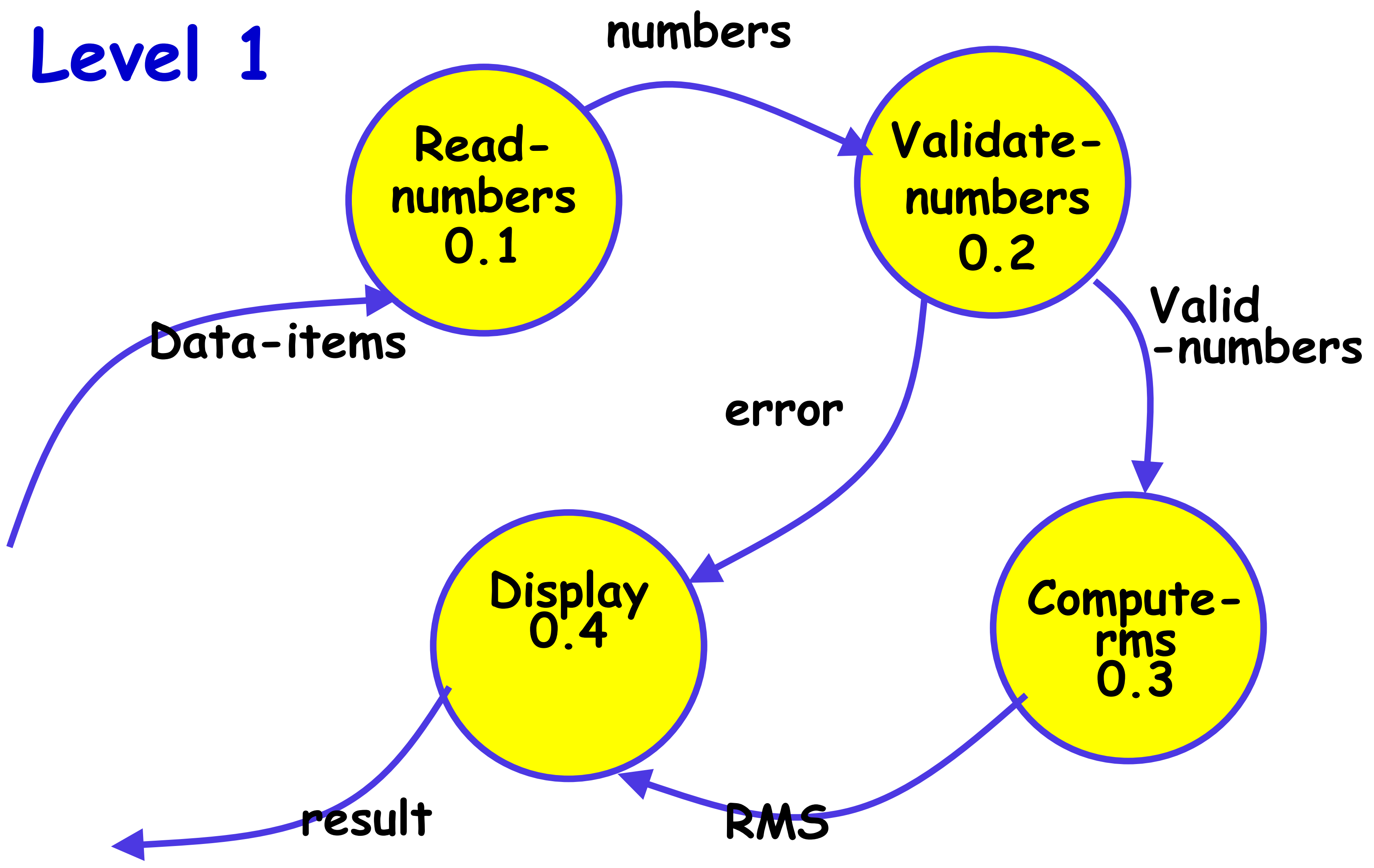
Example 1: RMS Calculating Software



Example 1: RMS Calculating Software

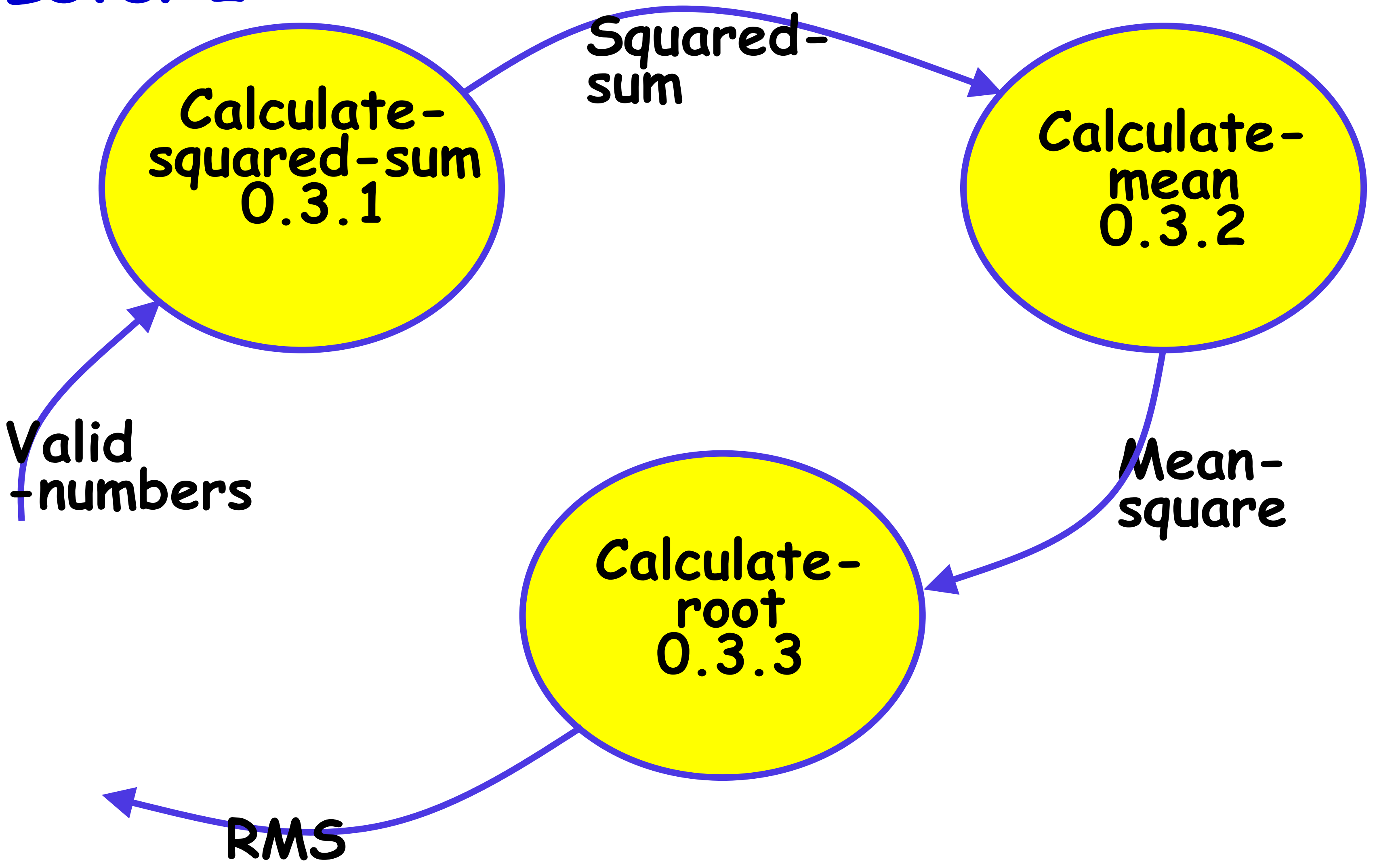


Example 1: RMS Calculating Software

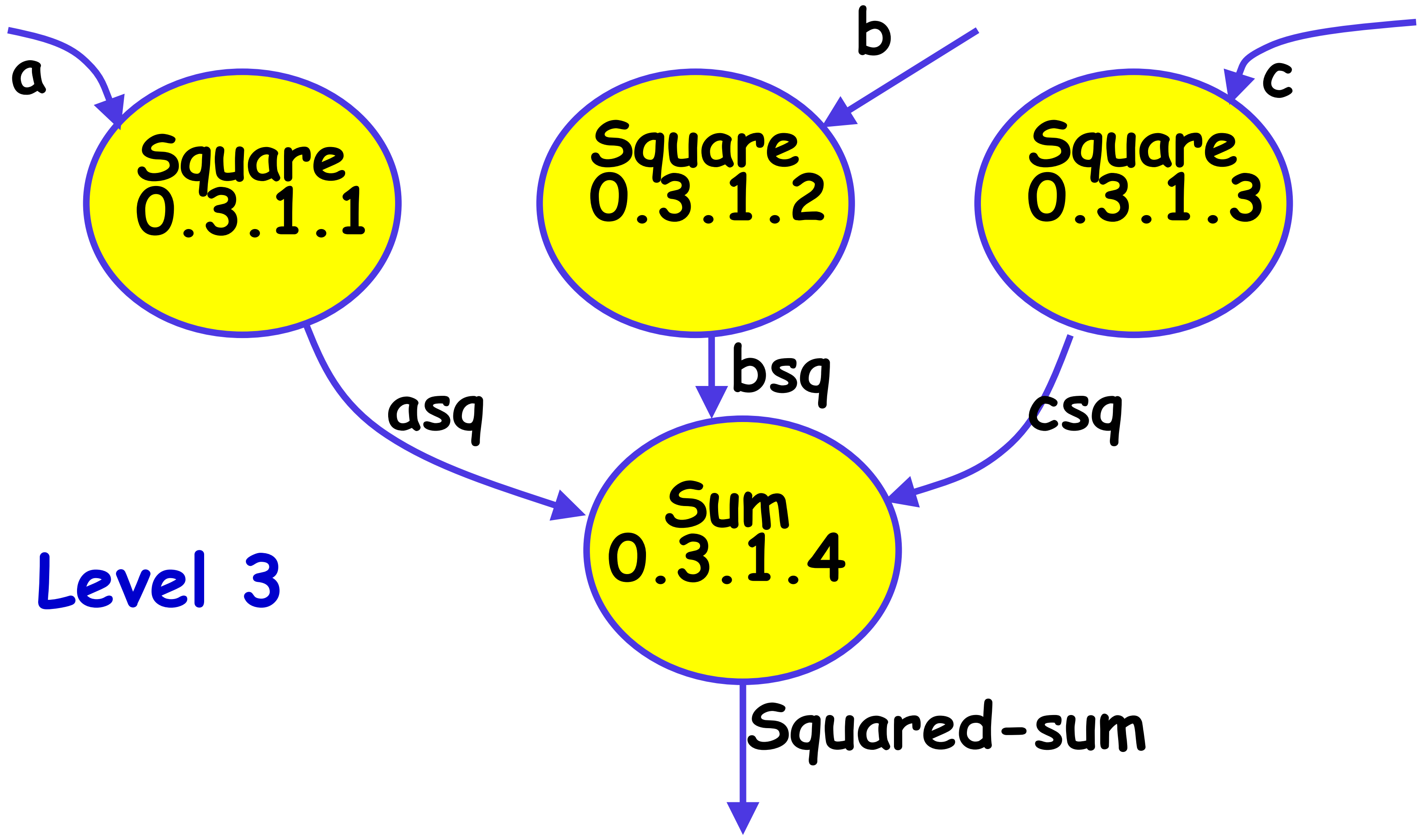


Example 1: RMS Calculating Software

Level 2

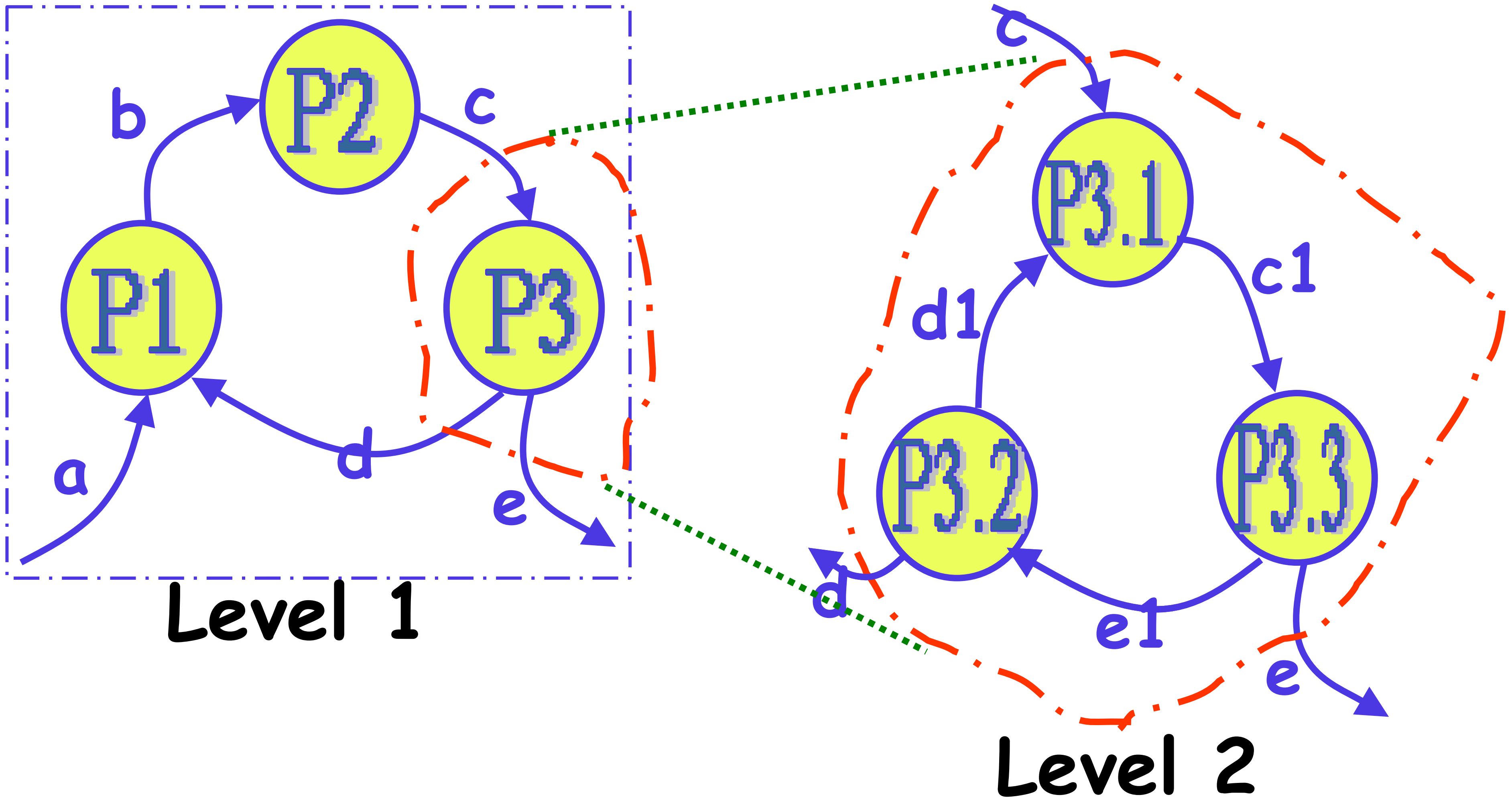


Example: RMS Calculating Software



Level 3

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)

5.4 STRUCTURED DESIGN

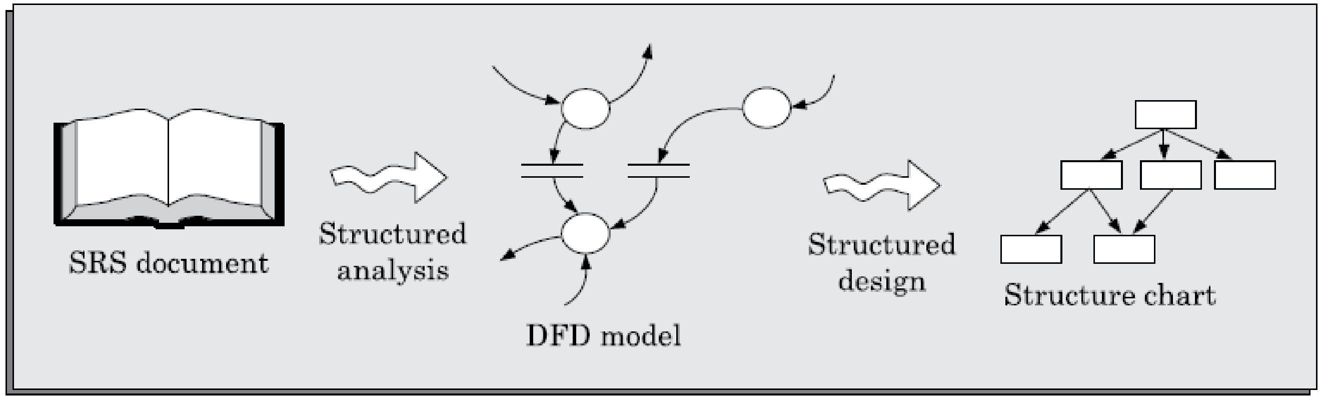
Structured Design transforms the results of **Structured Analysis** (i.e., the **DFD model**) into a **Structure Chart**, which defines the **software architecture**.

✅ What is a Structure Chart?

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.

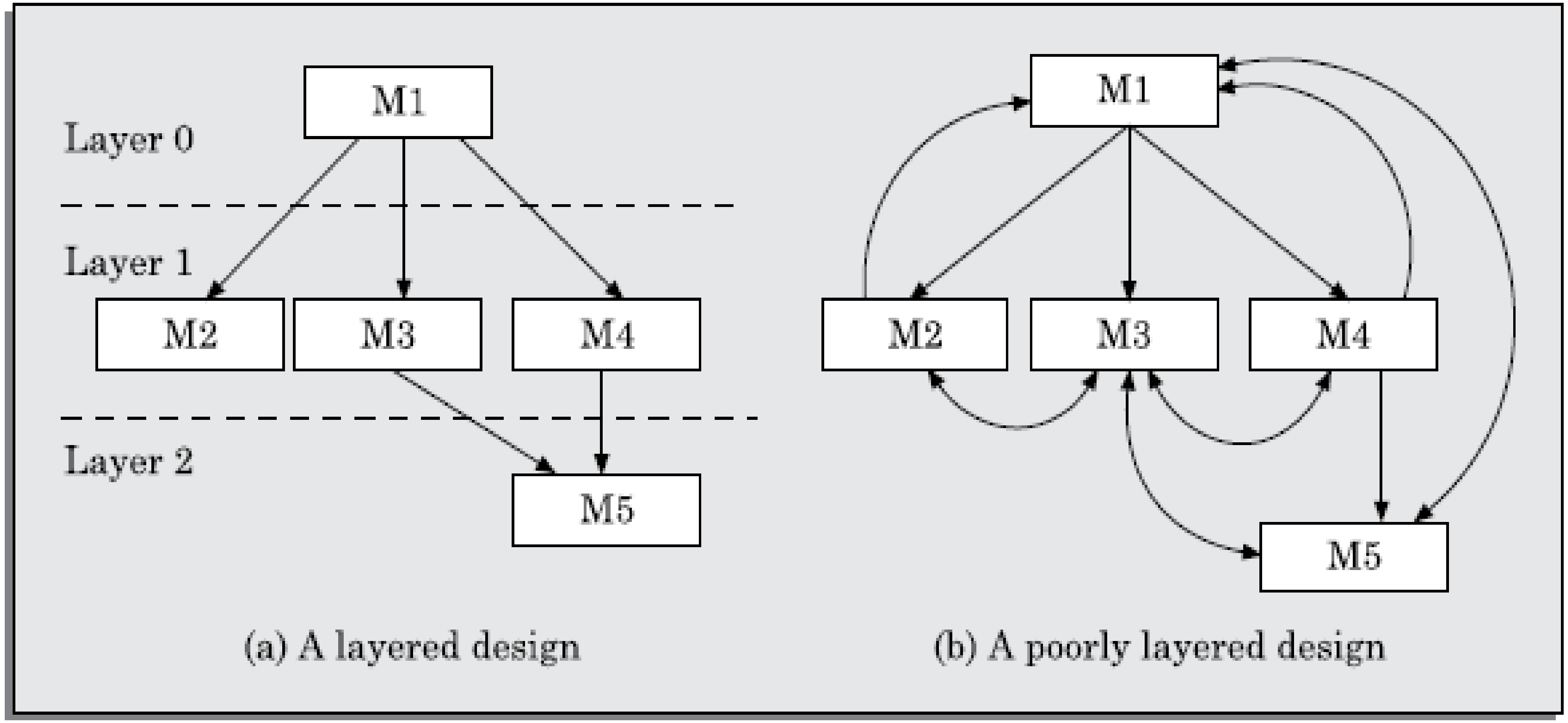


FIGURE 5.5 Examples of properly and poorly layered designs.

📌 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.

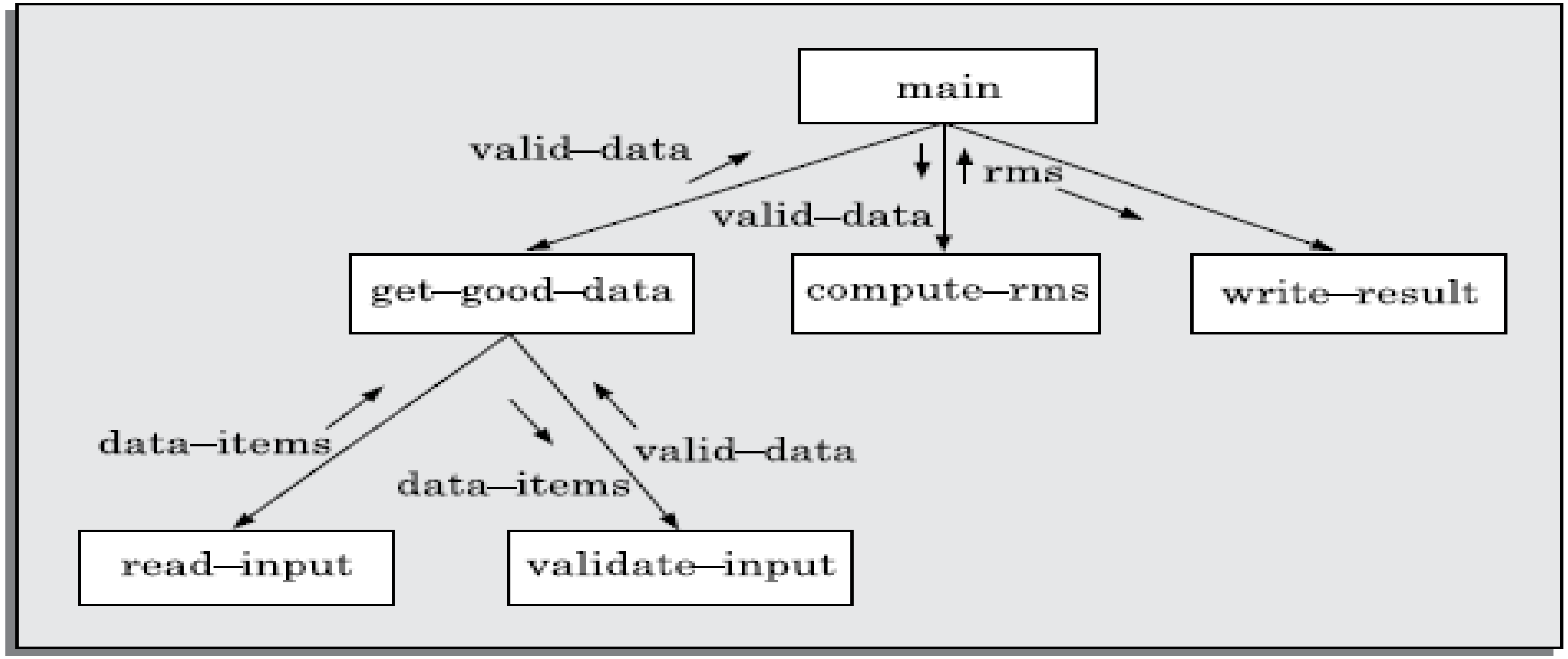
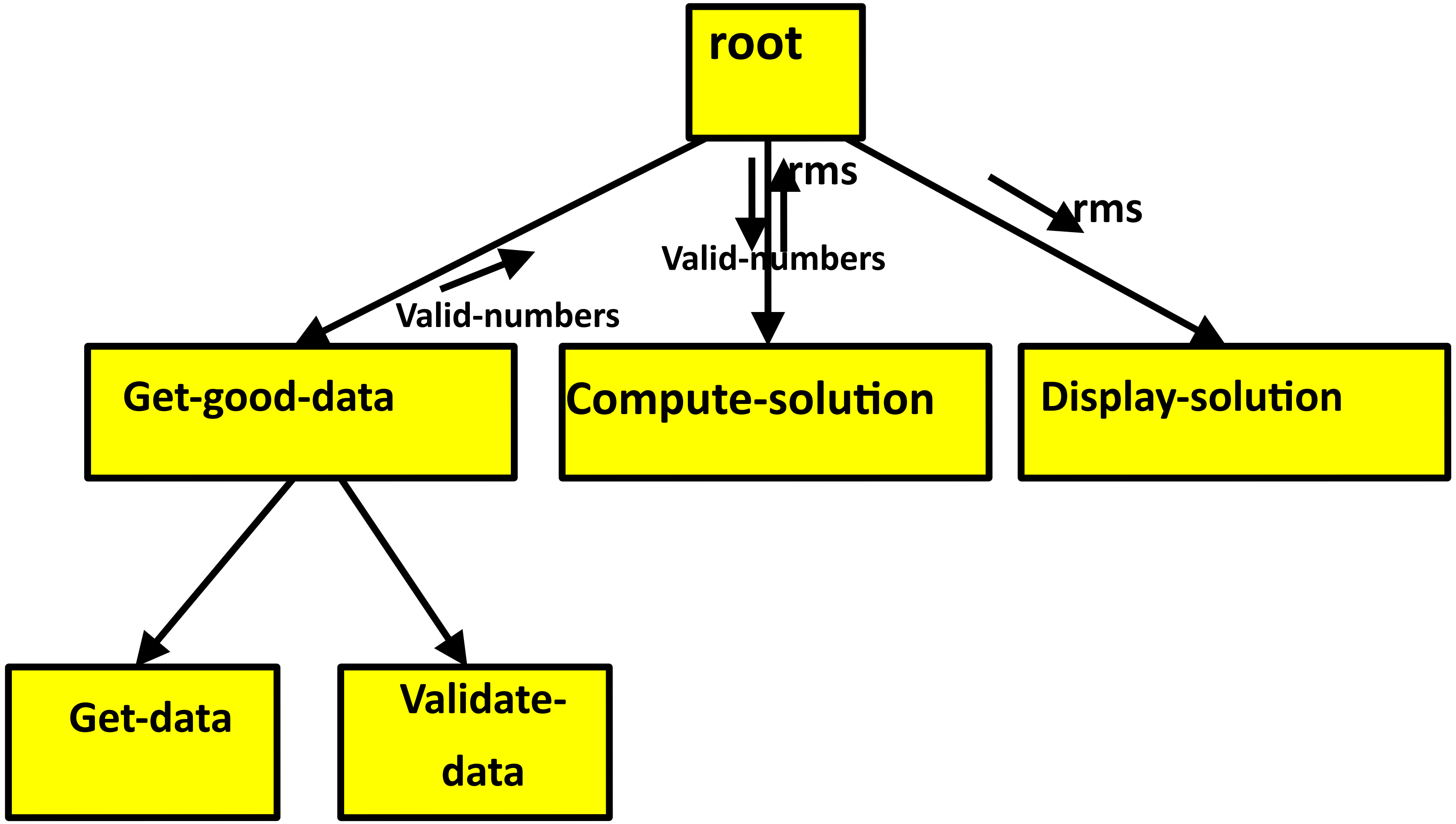


FIGURE 5.6 Structure chart for Problem 5.1.

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:

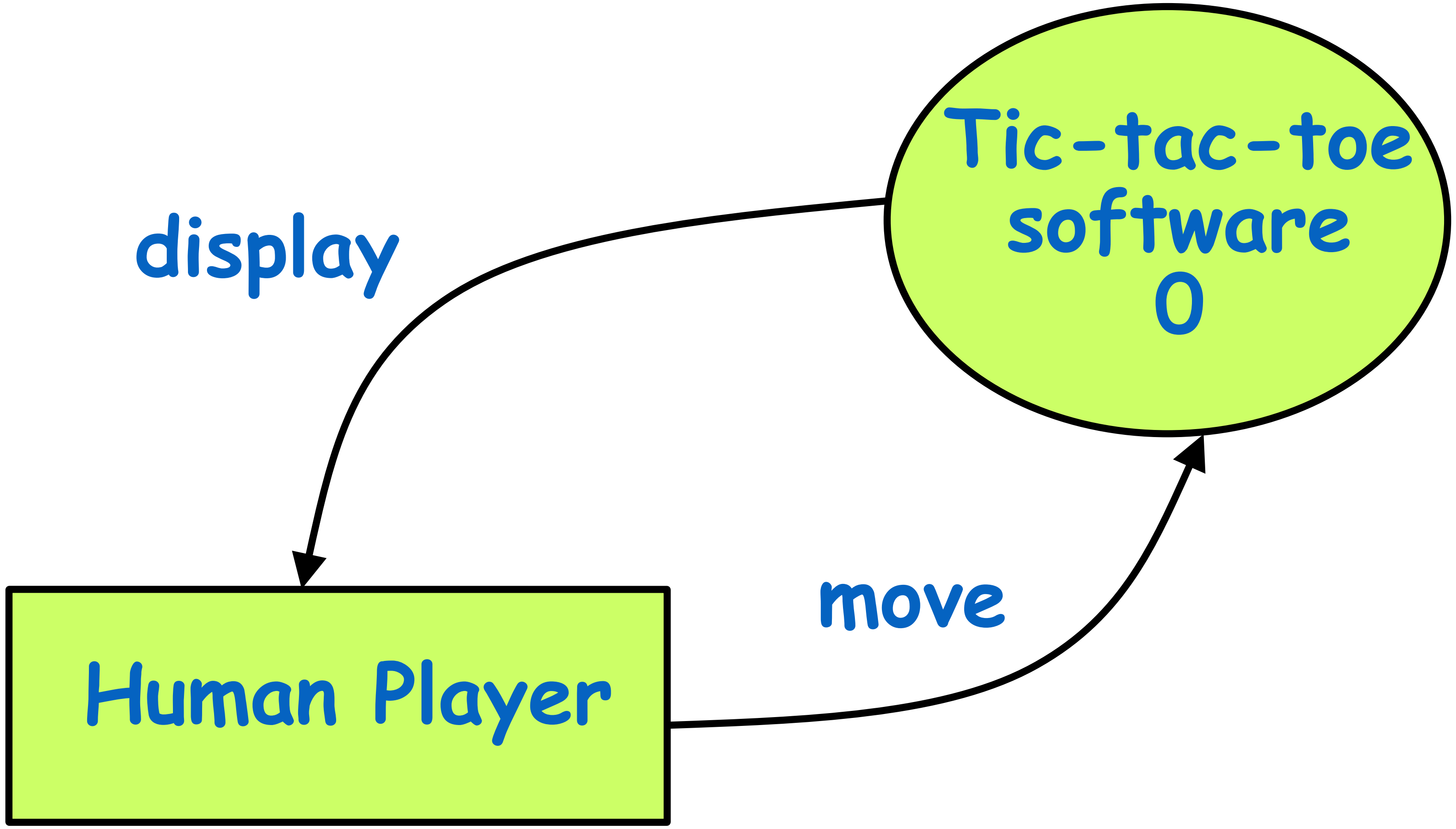
- Arrows represent data flow between modules.
  - For example:
    - valid-data is passed from get-good-data → compute-rms
    - rms is returned to main and passed to write-result

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

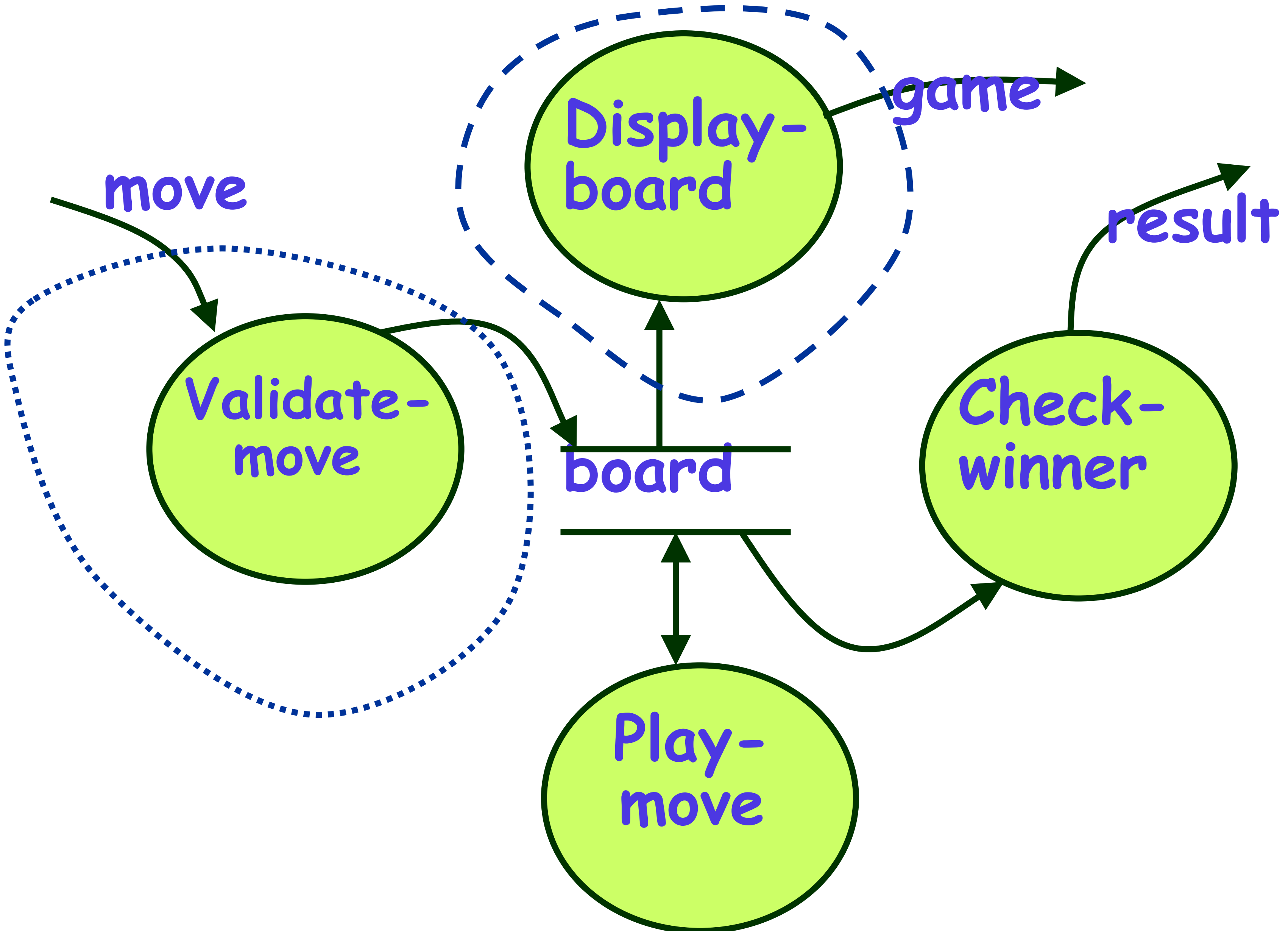
Example 2: Tic-Tac-Toe Computer Game

- 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.

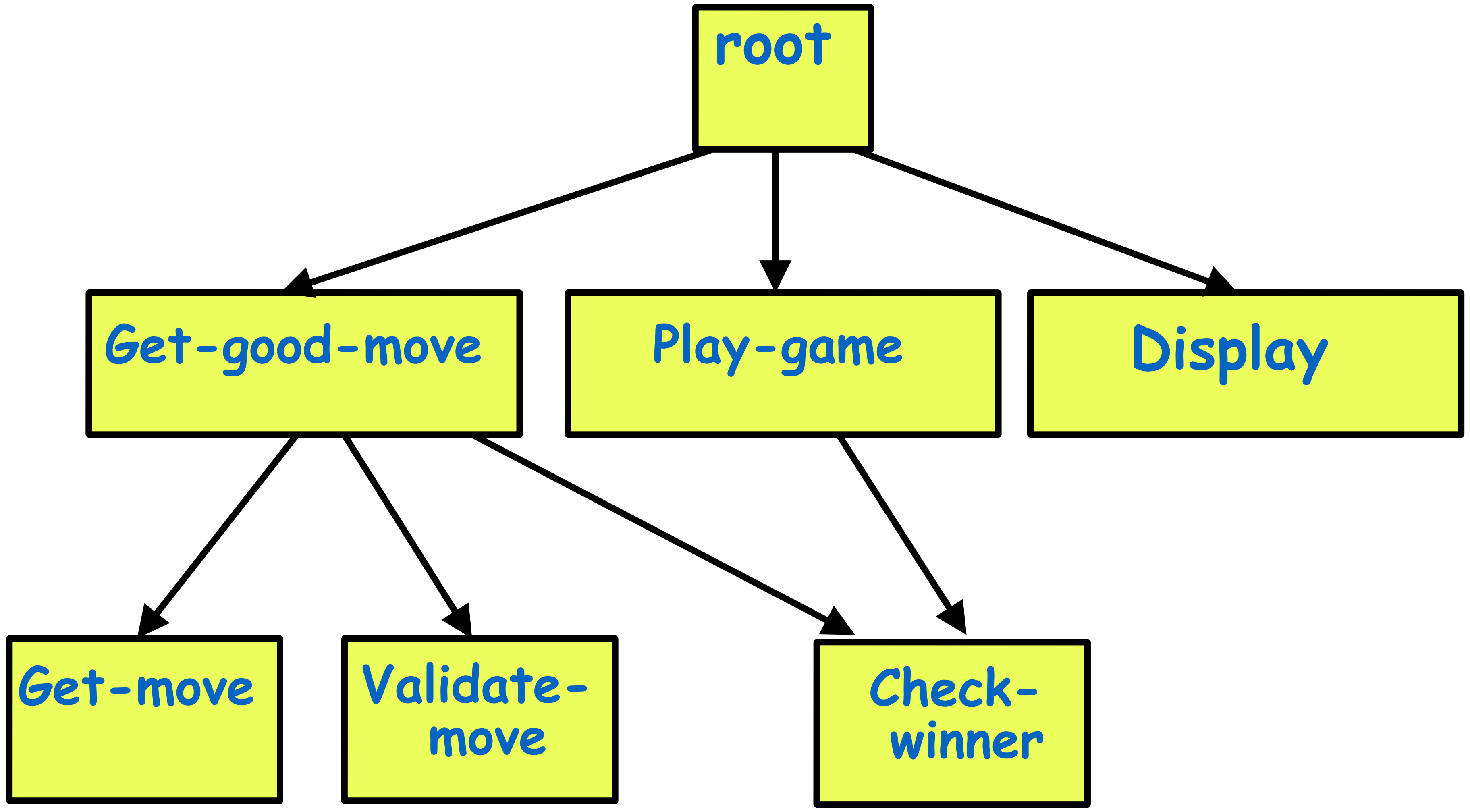
Context Diagram for Example 2



Level 1 DFD


move
Validate-move
Display-board
game
board
result
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:

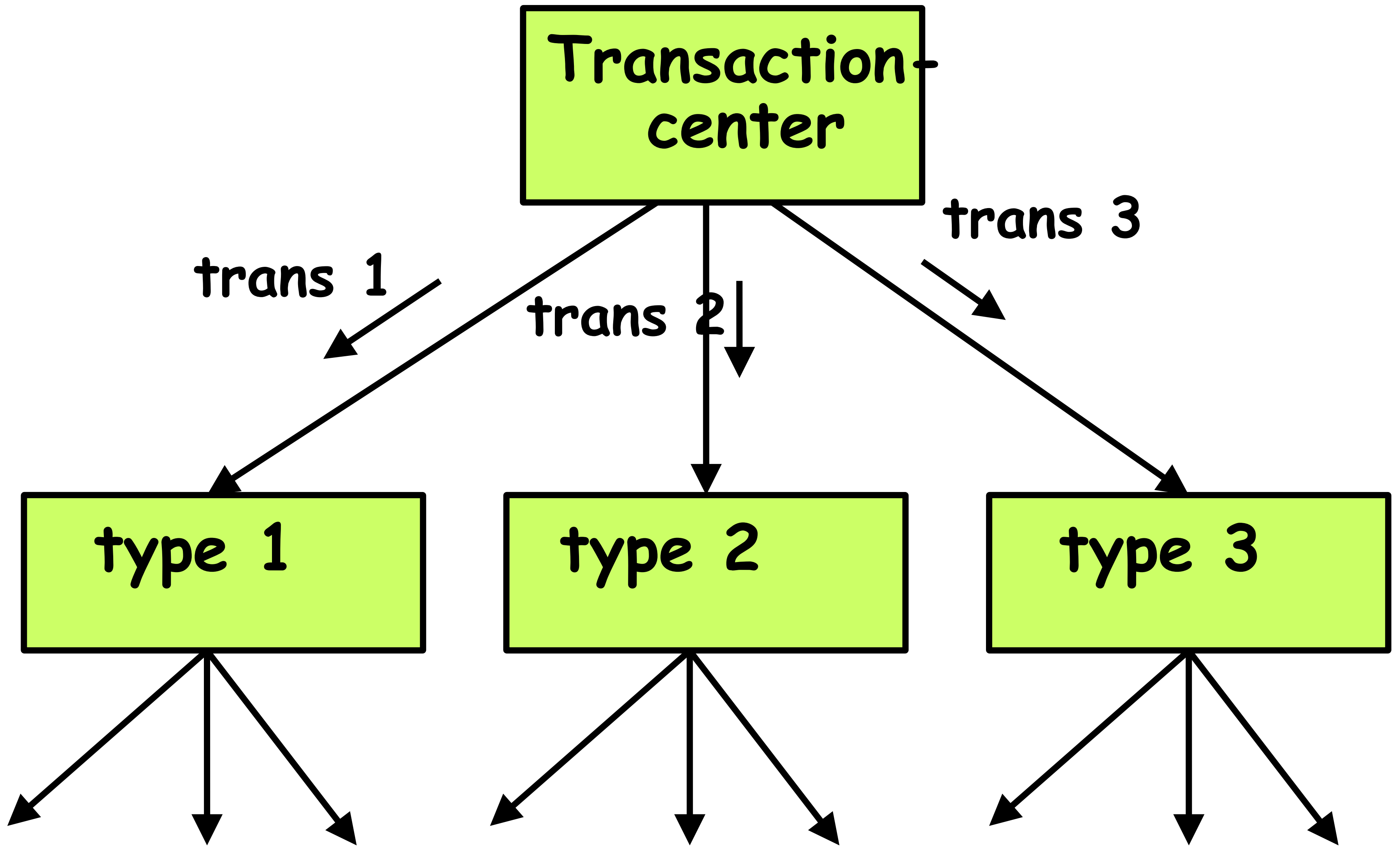
- ➢These help the system know **which transaction logic to execute**.

🔹 Structure Chart Characteristics (Transaction-Based)

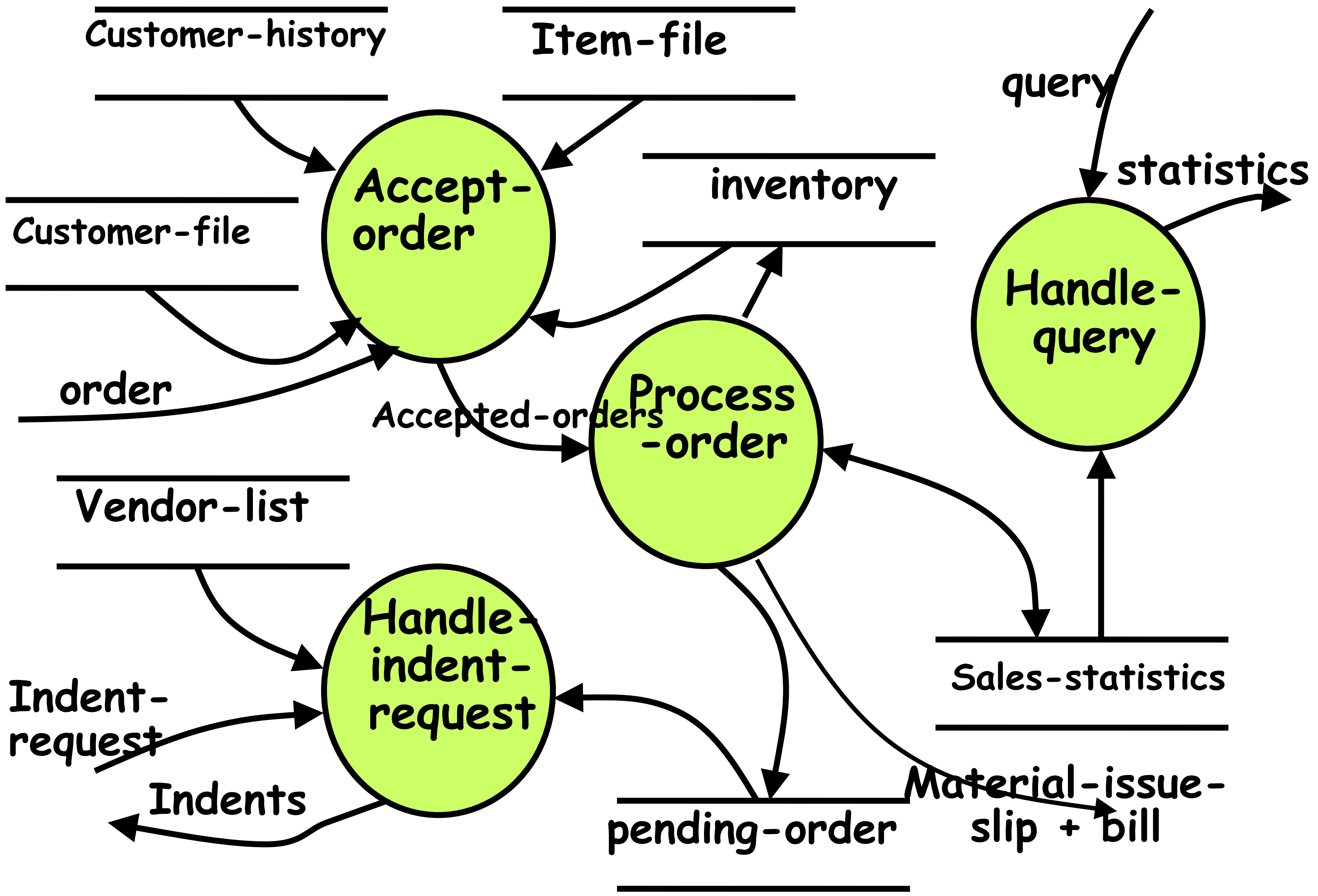
- **Root module**: Controls the flow.
- **Child modules**: Each handles a specific transaction.
- **No duplication**: Shared functionality may be placed in reusable modules.
- **Flexible**: Easy to add new transactions later.

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

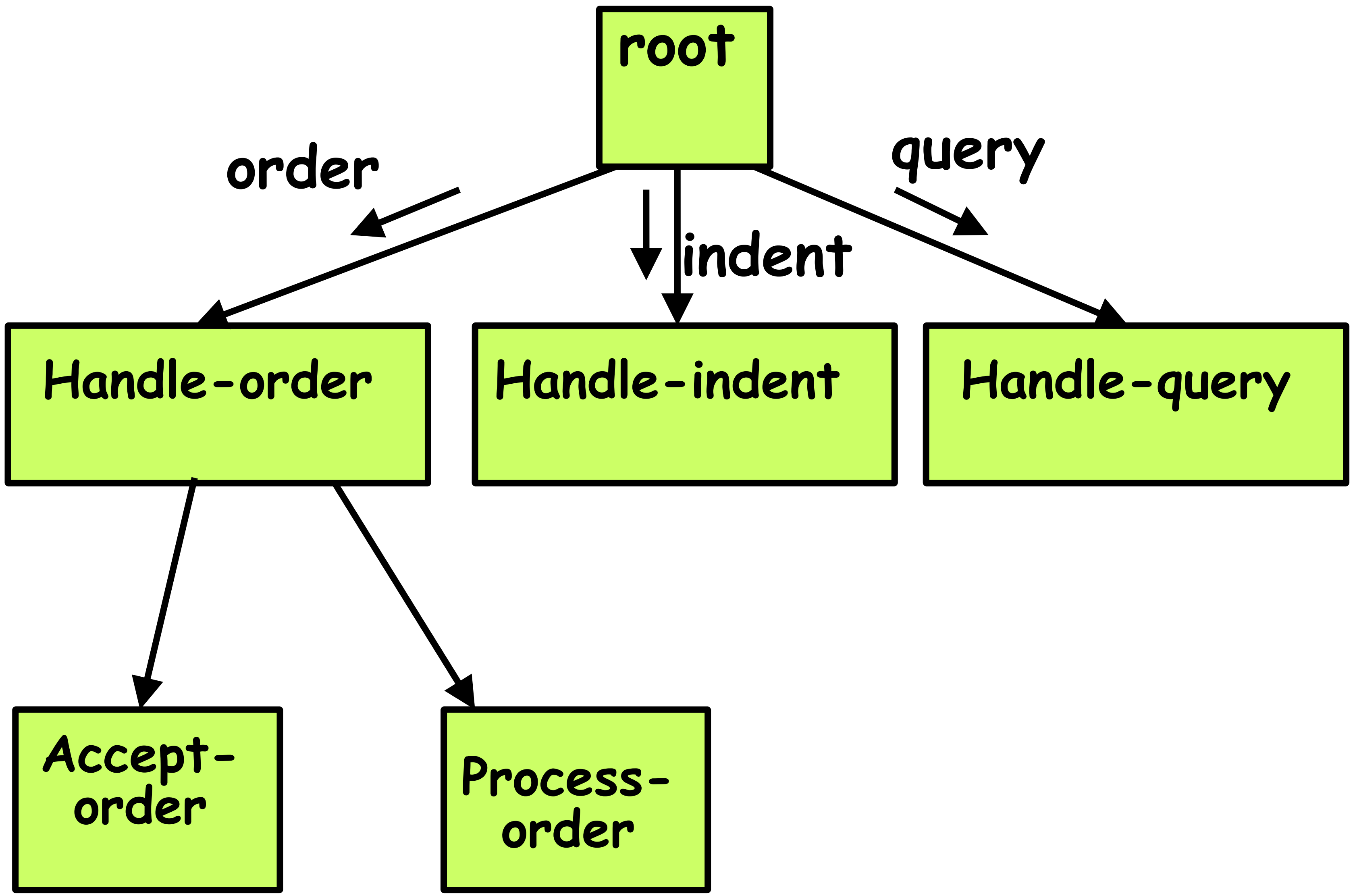
Transaction analysis



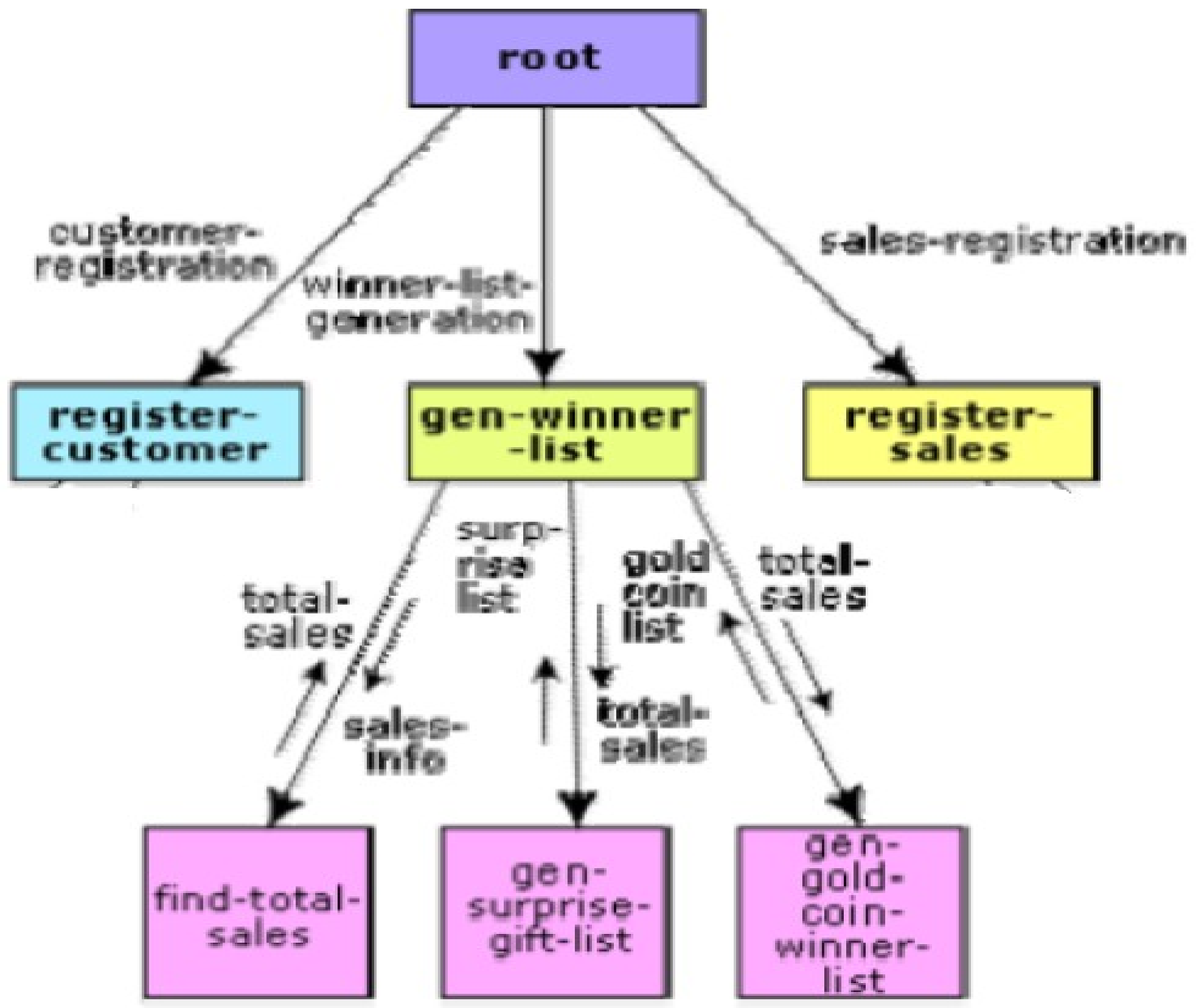
Level 1 DFD for TAS



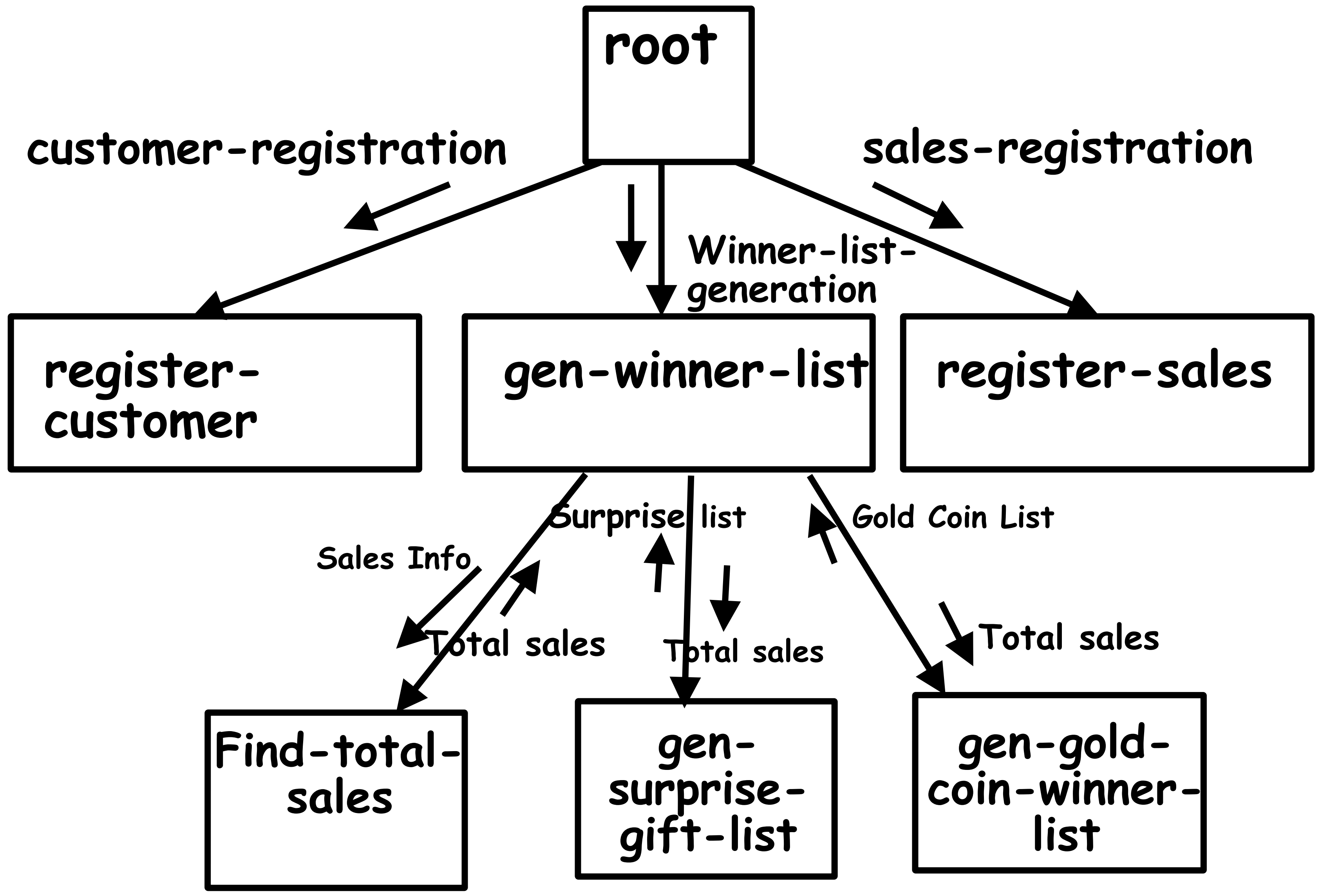
Structure Chart



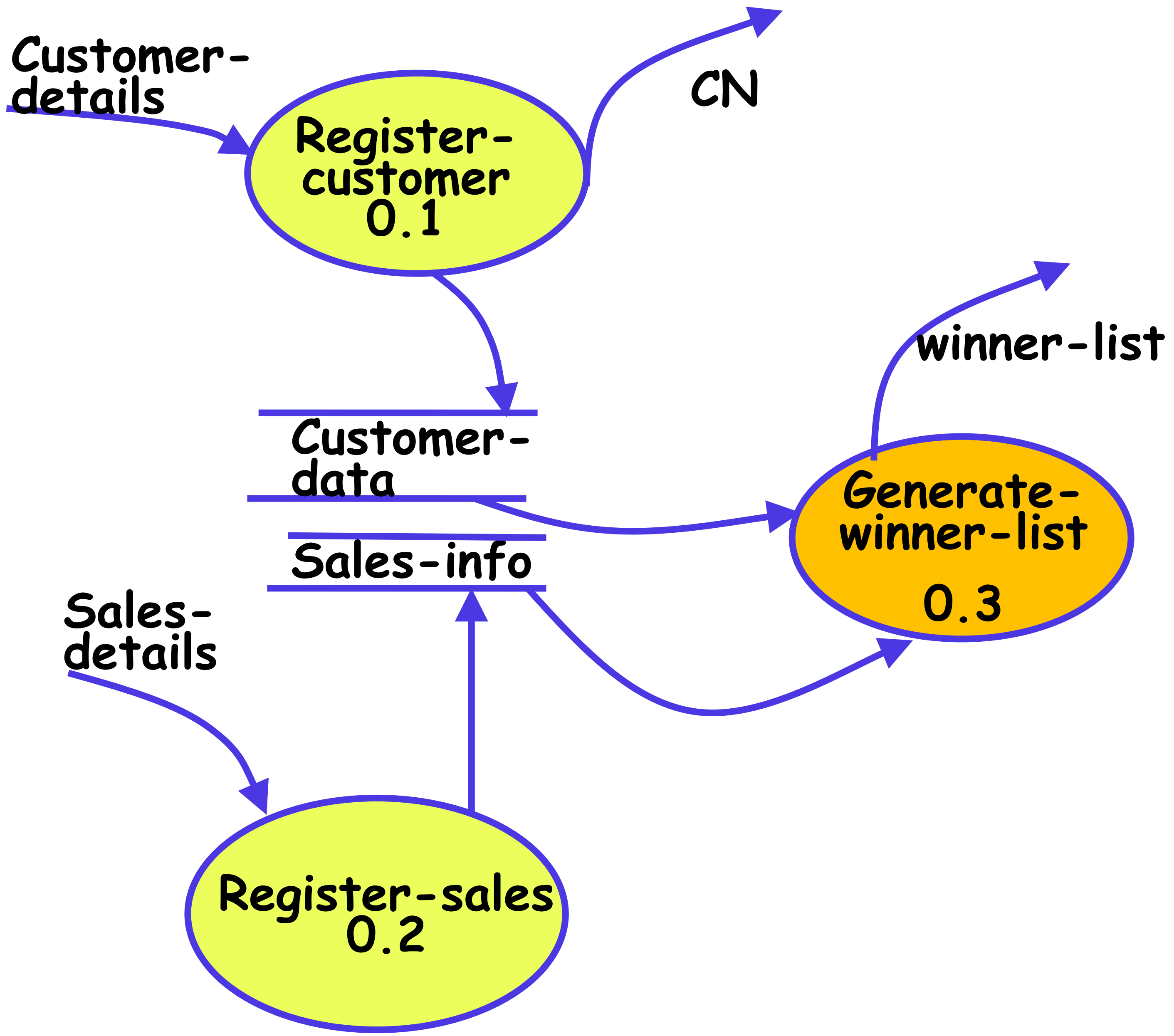
Structure Chart: Supermarket Prize Scheme



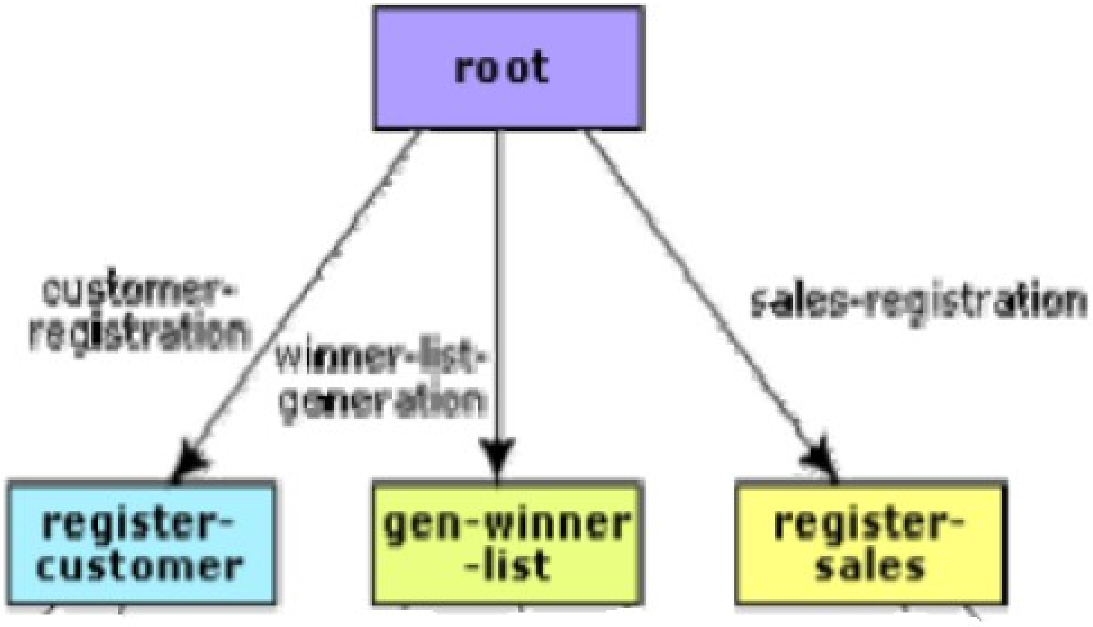
Structure Chart



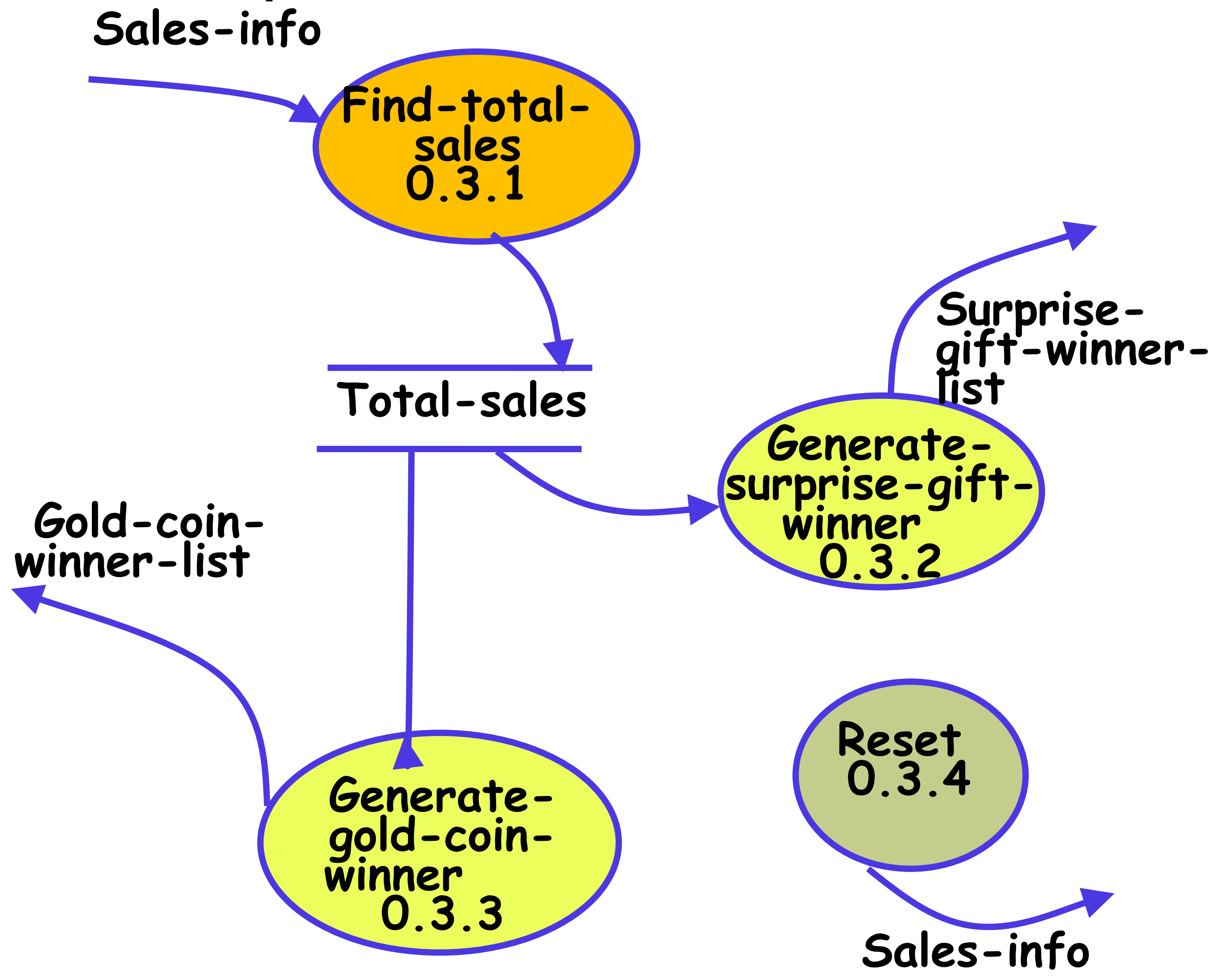
Level 1 DFD: Supermarket Prize Scheme



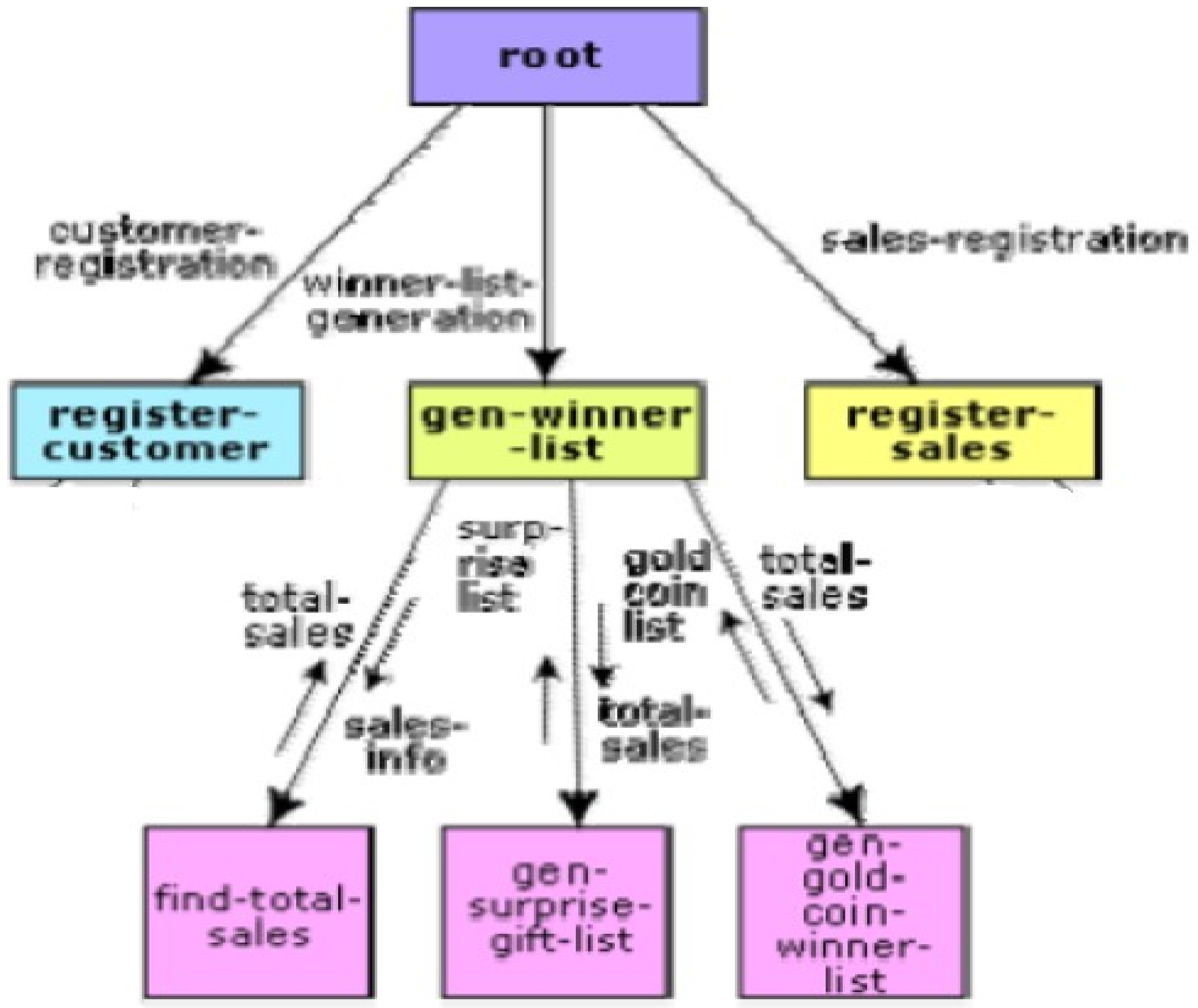
Structure Chart: Supermarket Prize Scheme



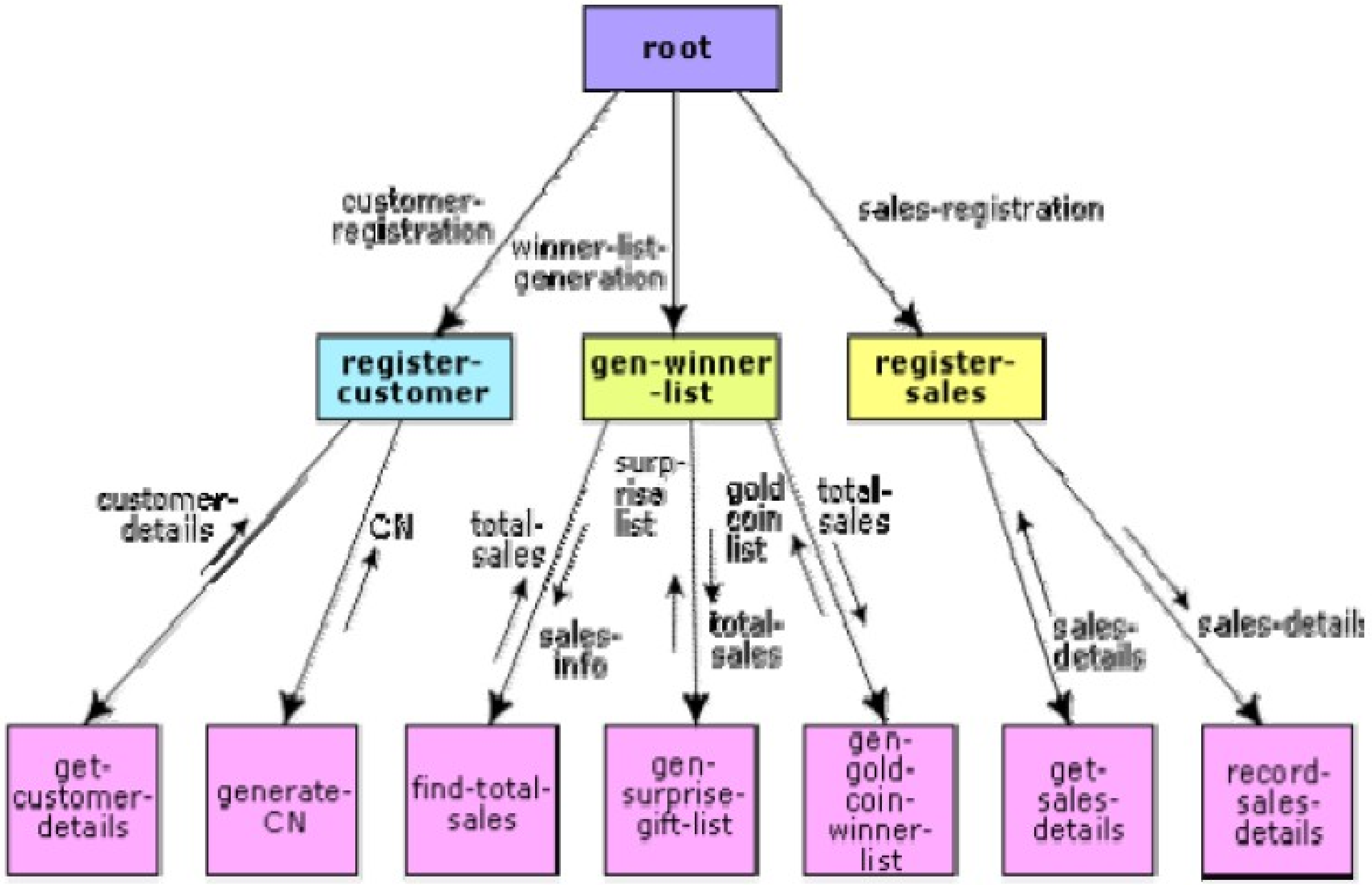
Level 2 DFD: Supermarket Prize Scheme


Sales-info
Find-total-sales 0.3.1
Total-sales
Surprise-gift-winner-list
Generate-surprise-gift-winner 0.3.2
Gold-coin-winner-list
Generate-gold-coin-winner 0.3.3
Reset 0.3.4
Sales-info

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

5.6 DESIGN REVIEW

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