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..)

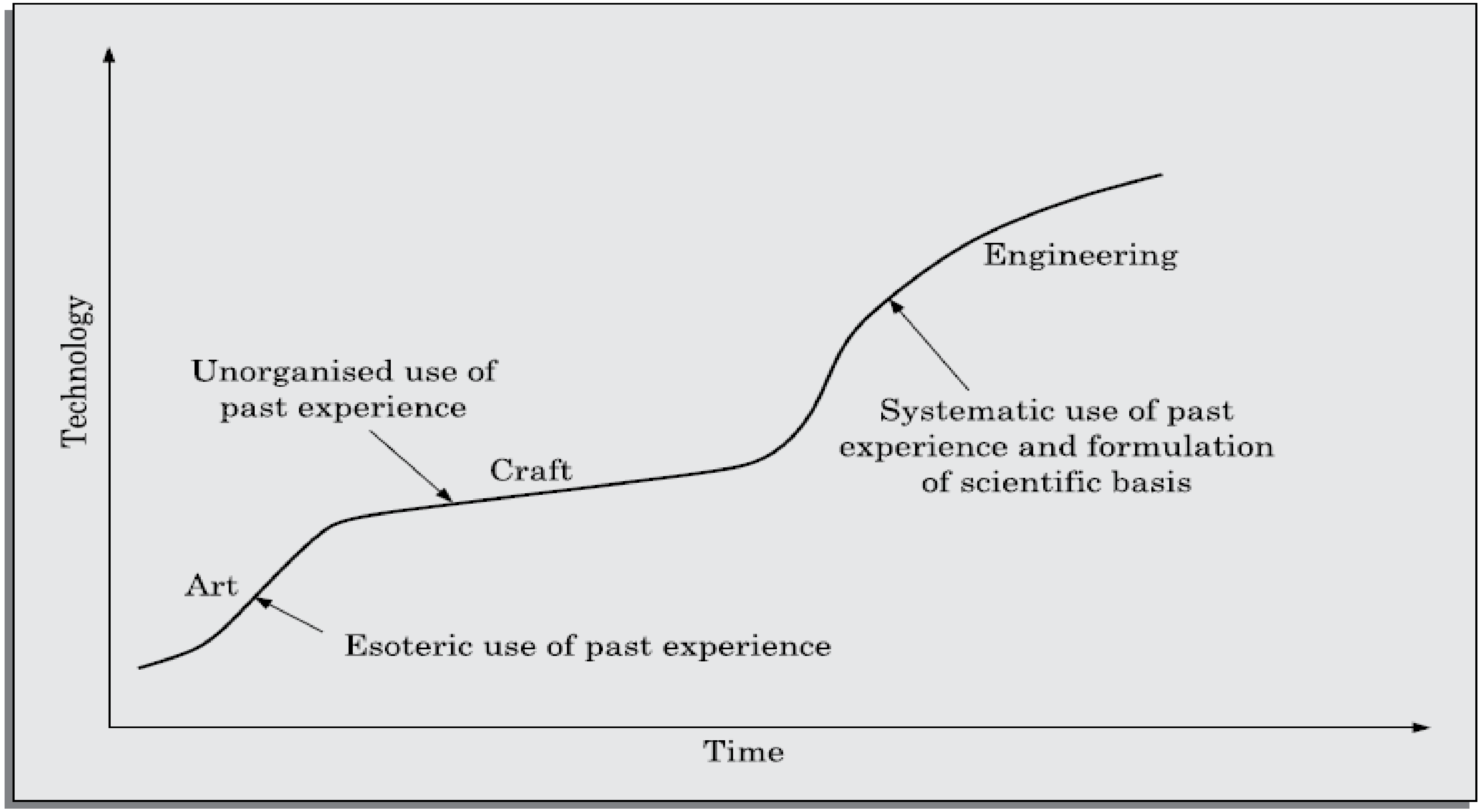

Technology
Engineering
Unorganised use of past experience
Craft
Systematic use of past experience and formulation of scientific basis
Art
Esoteric use of past experience
Time

FIGURE 1.1 Evolution of technology with time.

1.1.3 A Solution to the Software Crisis

- 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**.

Causes:

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- Outdated or inefficient development methods

Solution:

- 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).

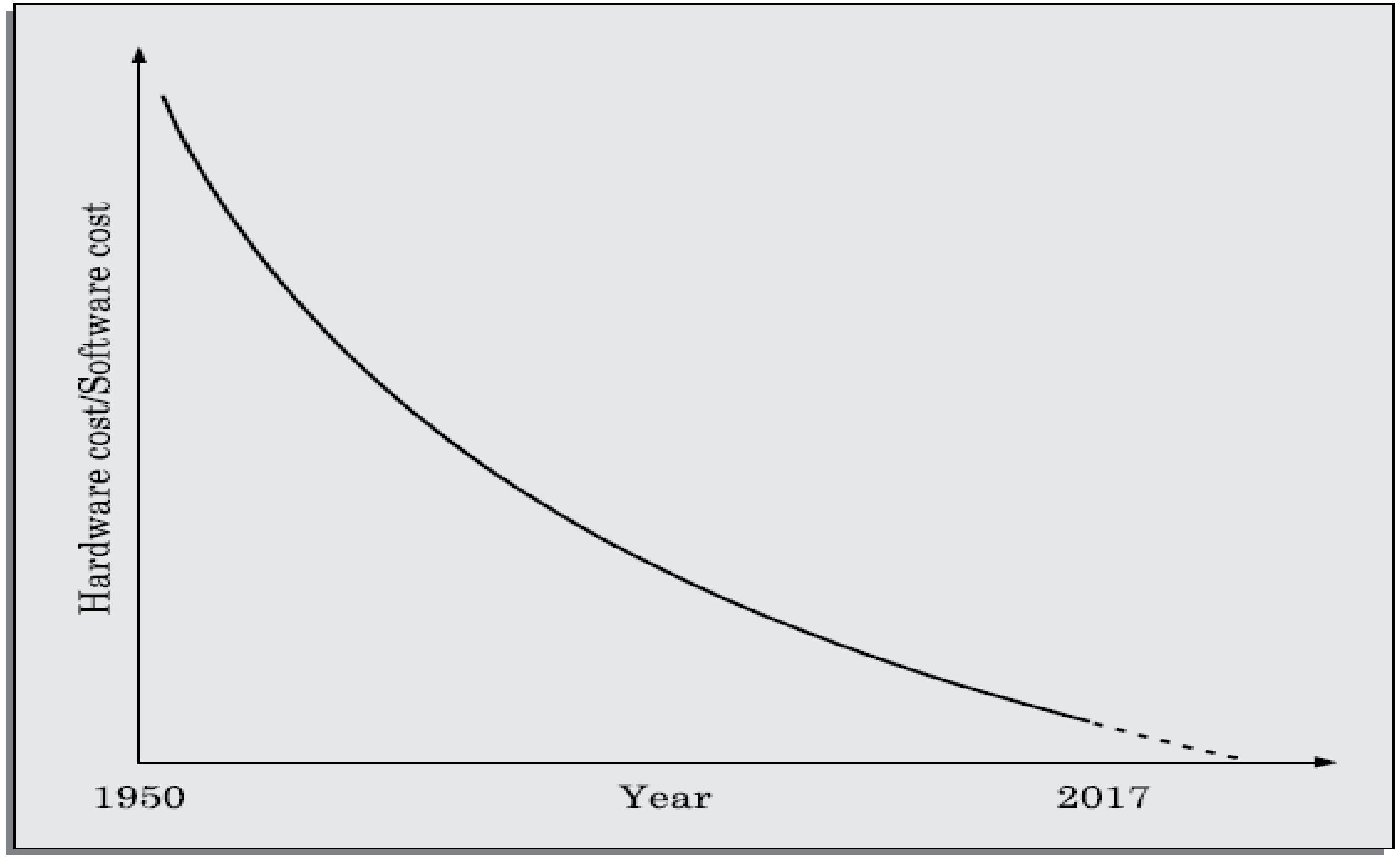


FIGURE 1.2 Relative changes of hardware and software costs over time.

- 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.**

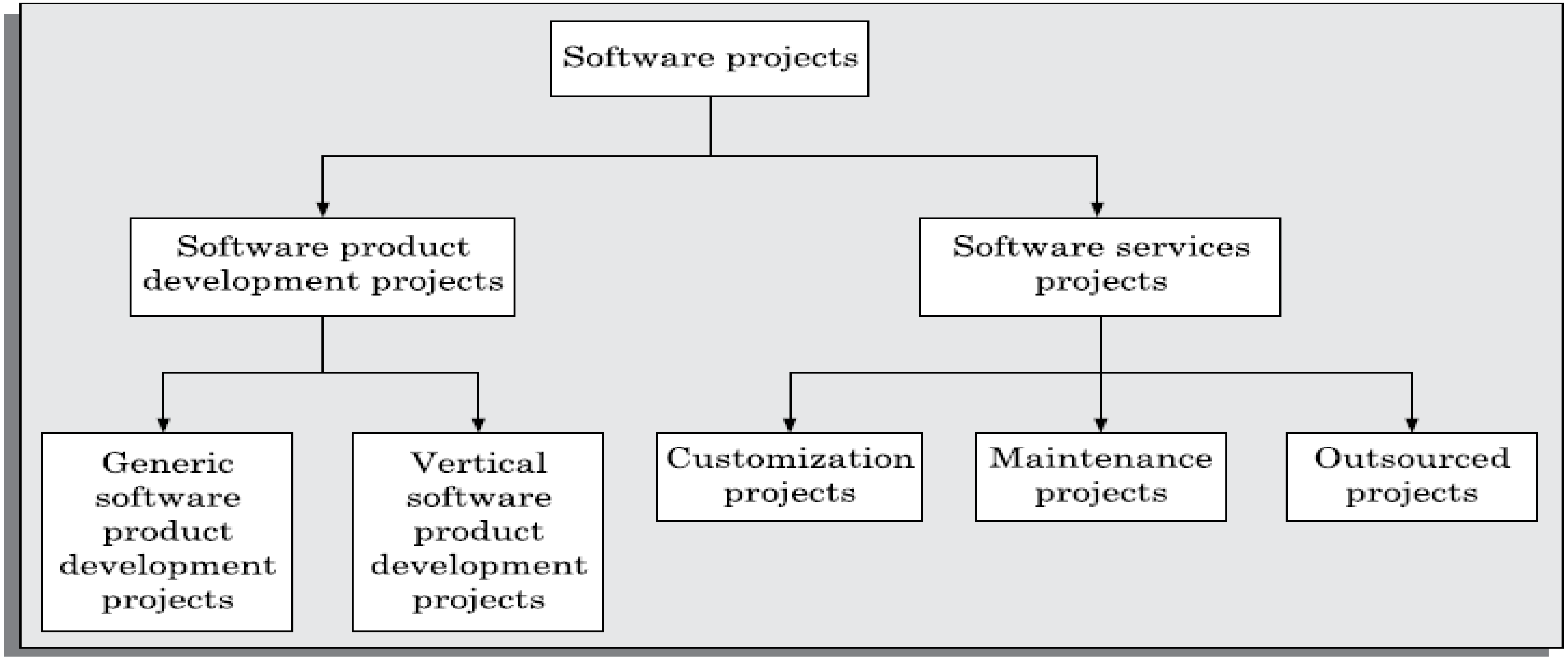

Software projects
Software product development projects
Software services projects
Generic software product development projects
Vertical software product development projects
Customization projects
Maintenance projects
Outsourced projects

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.)

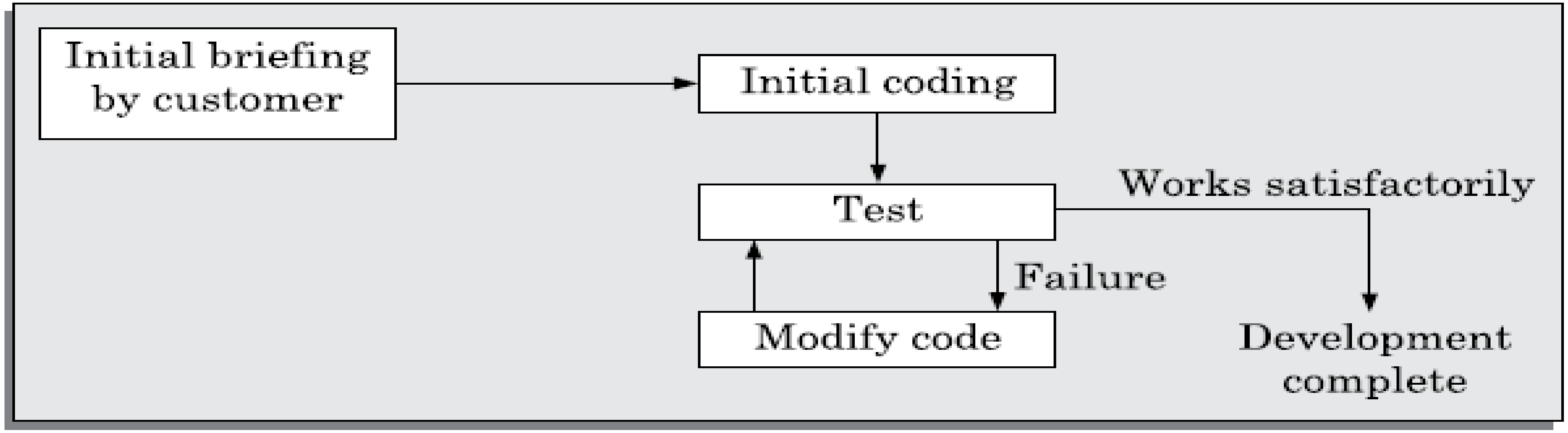


FIGURE 1.4 Exploratory program development.

When It Works (and When It Doesn’t)

- 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.

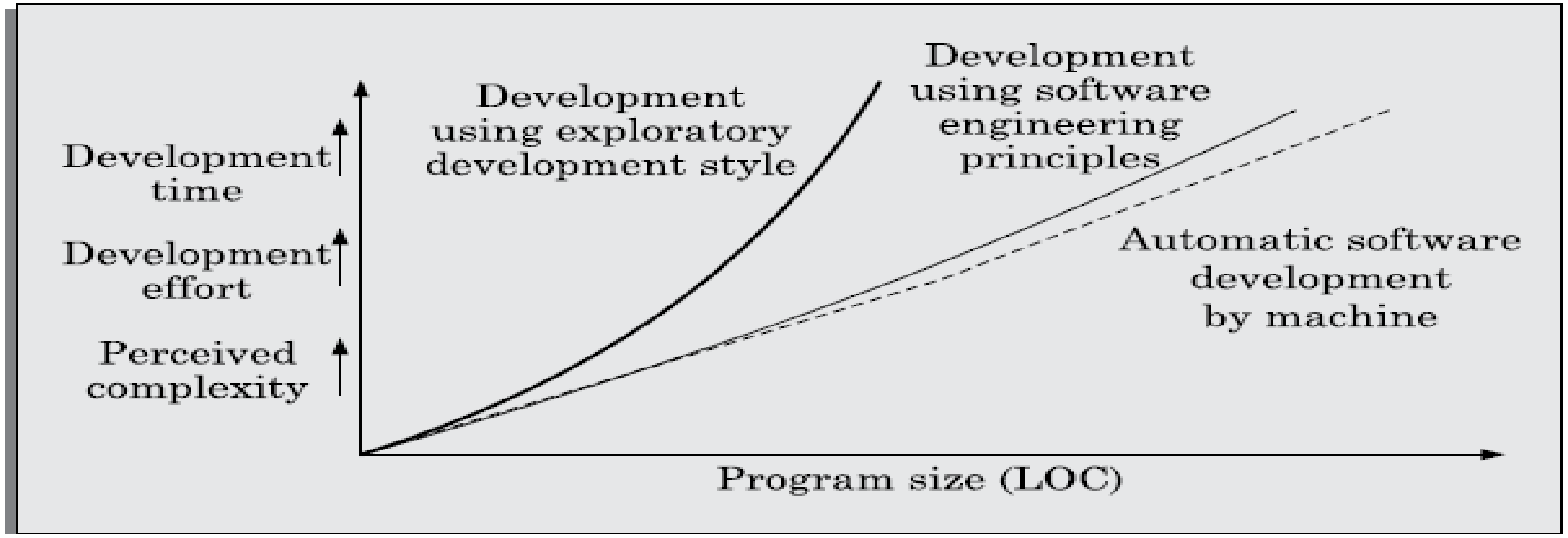


FIGURE 1.5 Increase in development time and effort with problem size.

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.

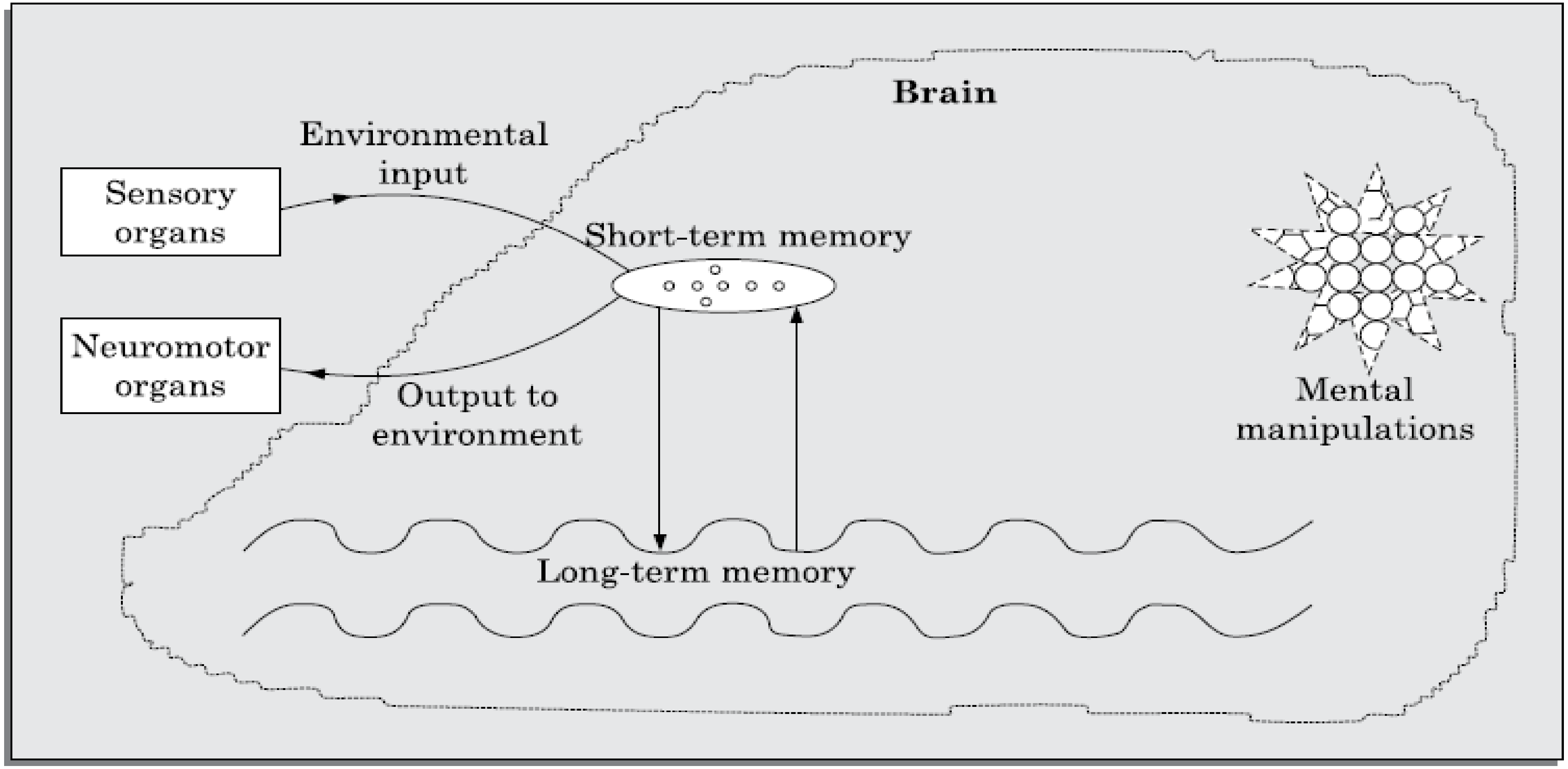


FIGURE 1.6 Human cognition mechanism model.

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:

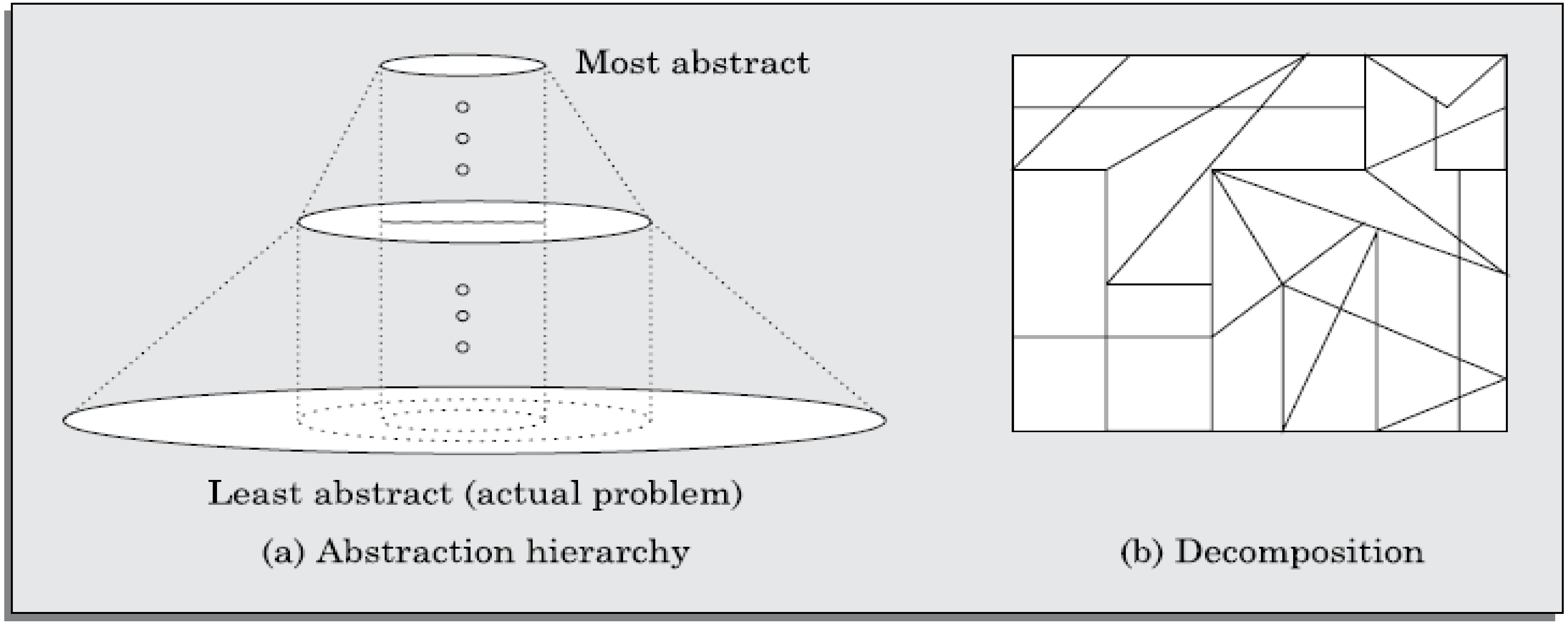


FIGURE 1.7 Schematic representation.

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.

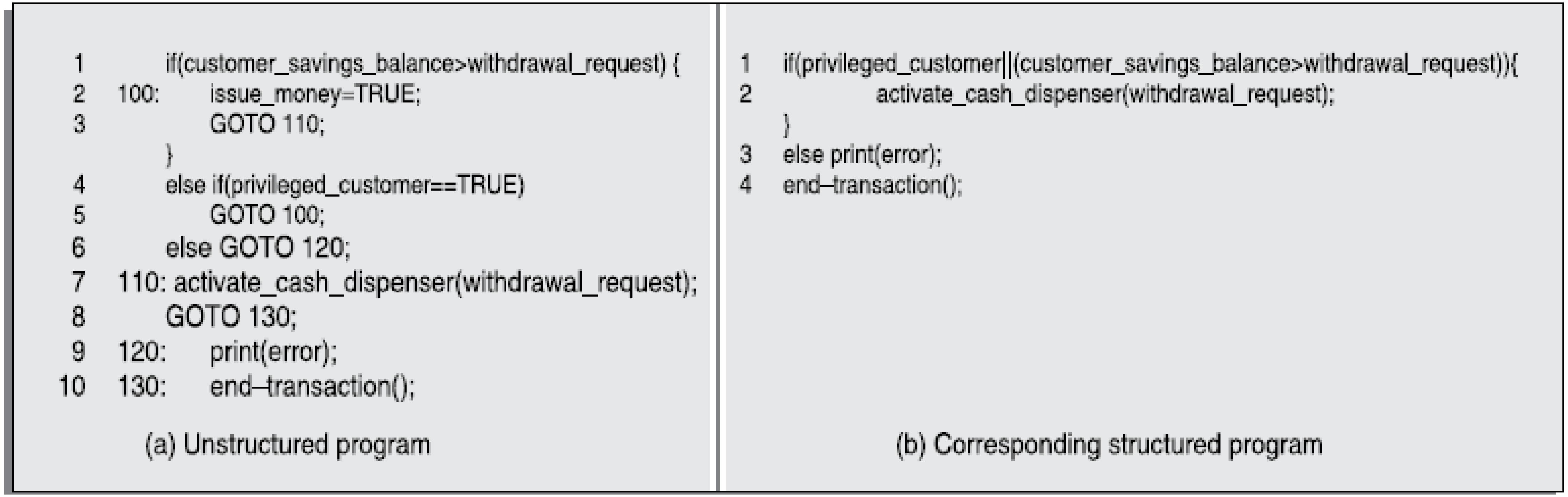
1.4.2 High-Level Language Programming

- 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).

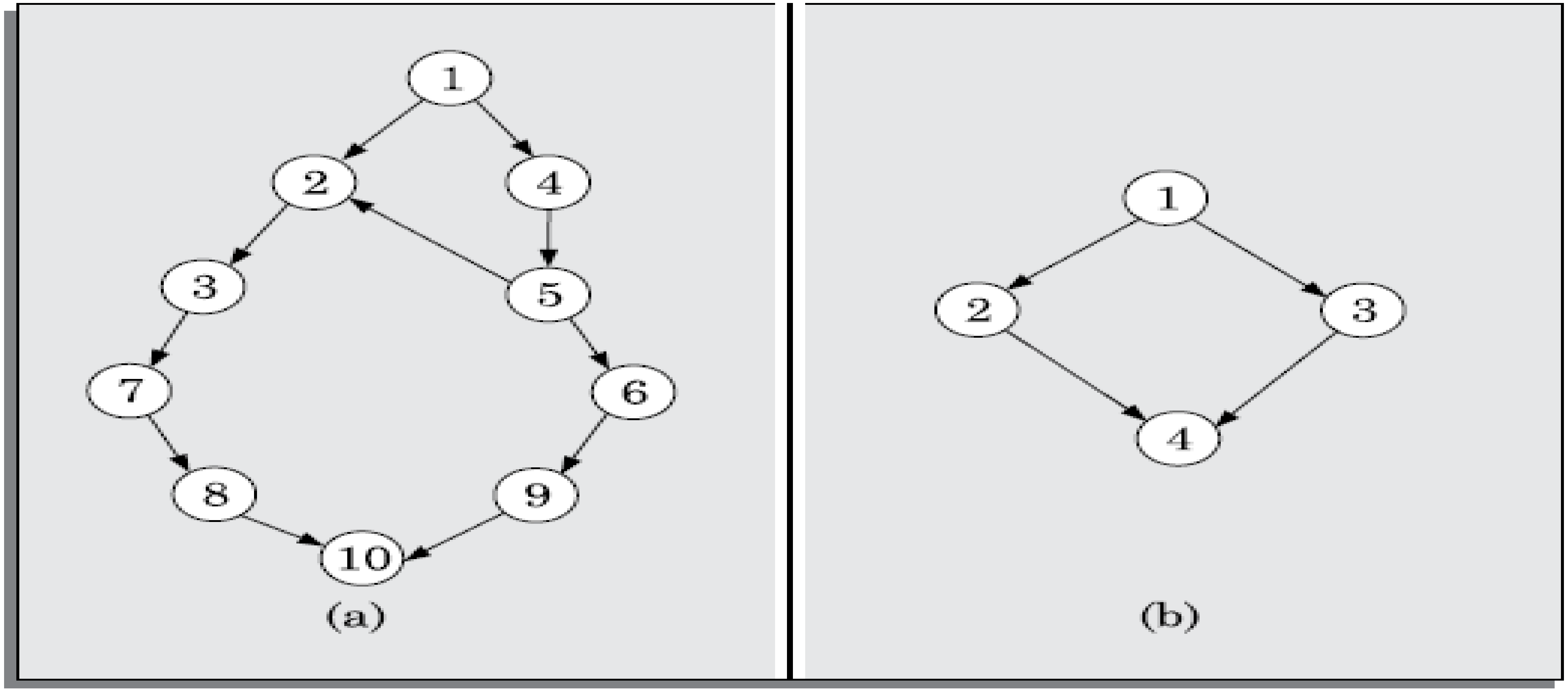


FIGURE 1.10 Control flow graphs of the programs of Figures 1.9(a) and (b).

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.

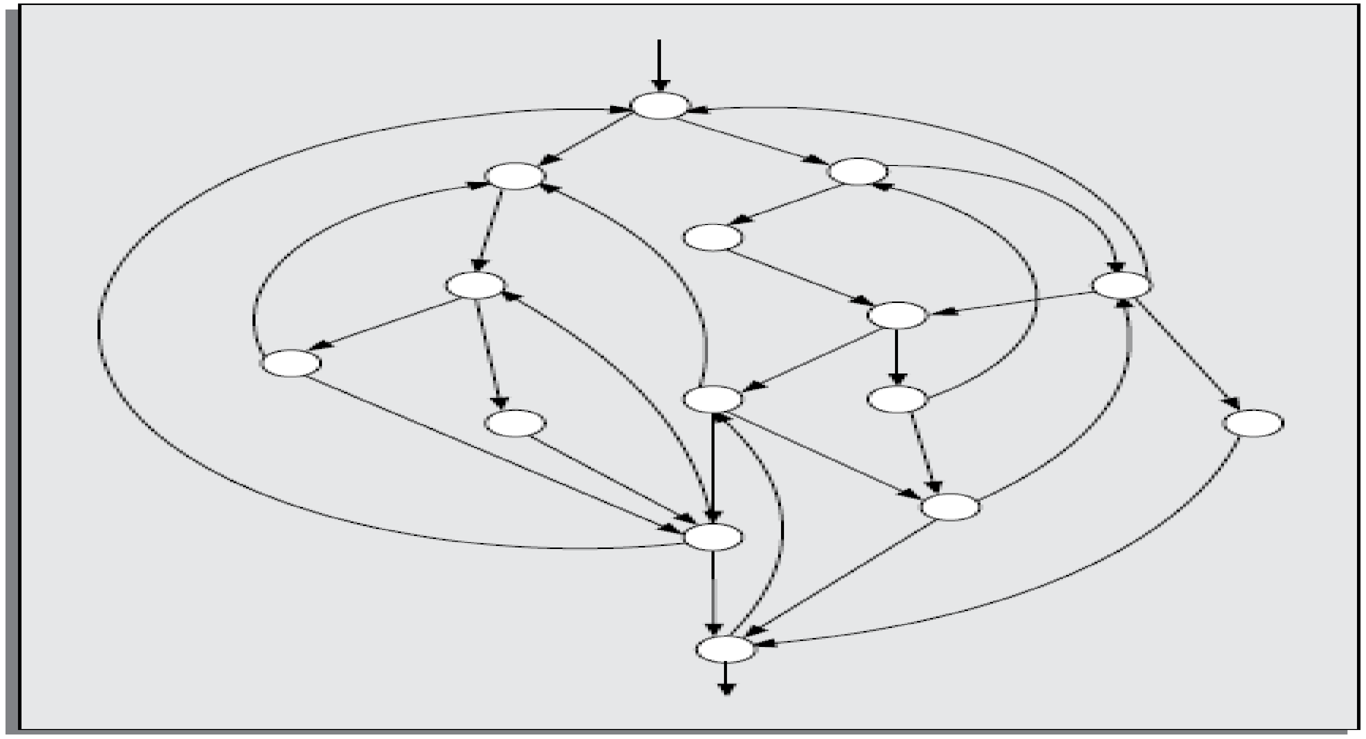


FIGURE 1.11 CFG of a program having too many GO TO statements.

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.

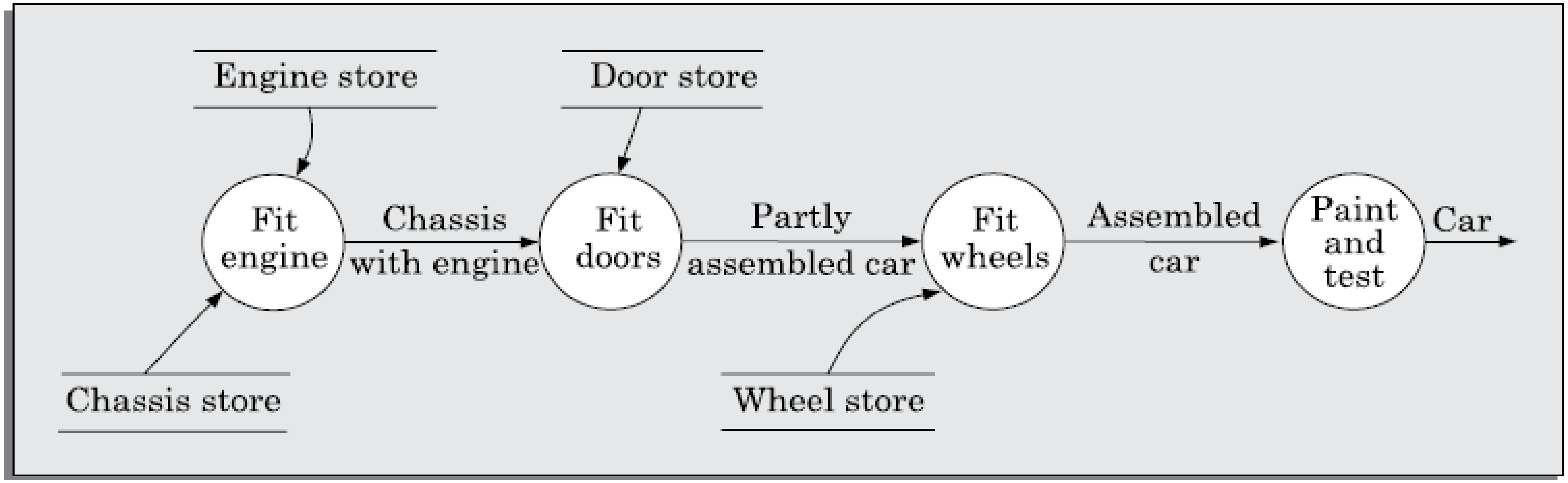


FIGURE 1.12 Data flow model of a car assembly plant.

1.4.6 Object-Oriented Design (OOD)

- 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.

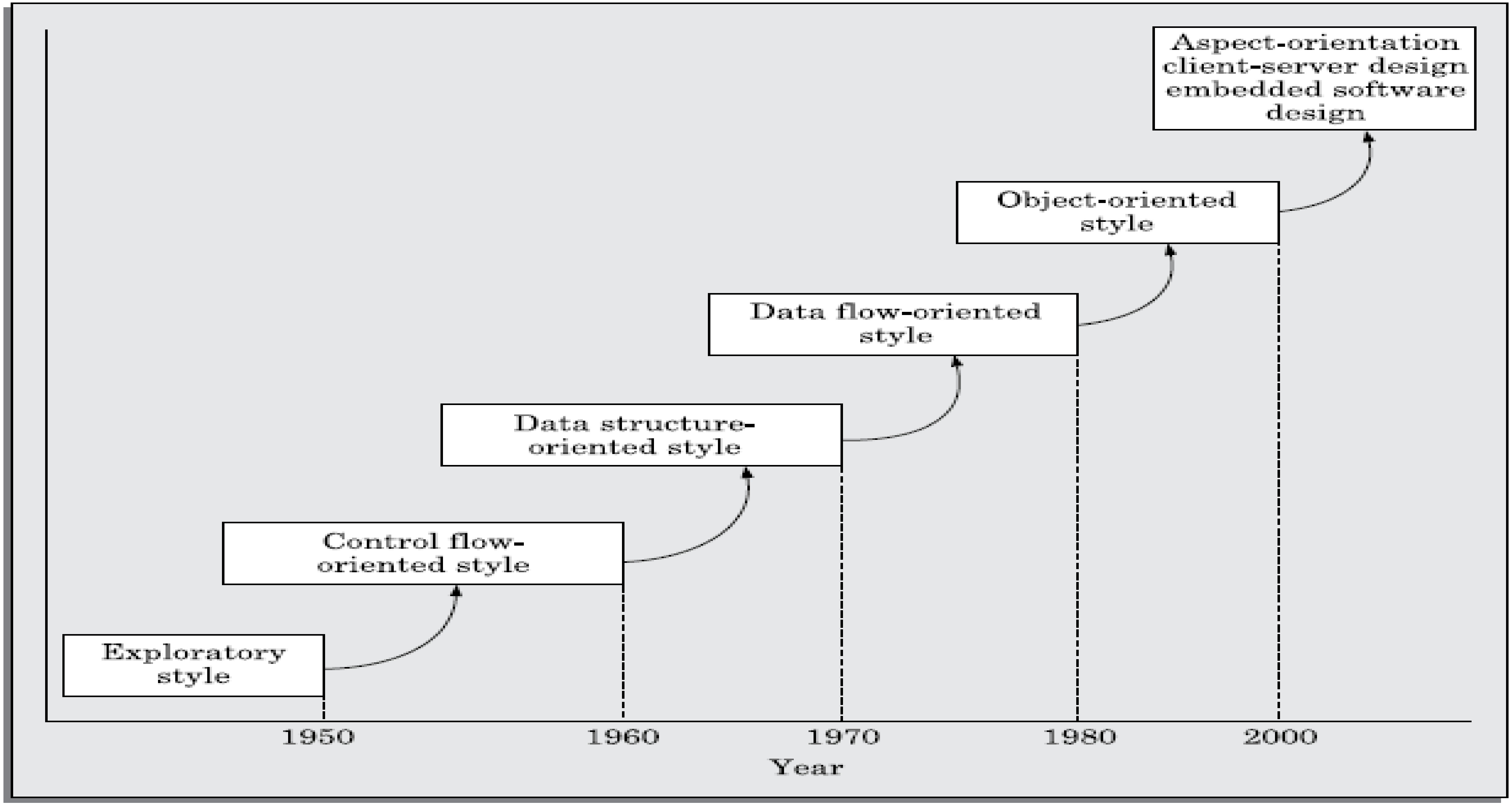

Aspect-orientation client-server design embedded software design
Object-oriented style
Data flow-oriented style
Data structure-oriented style
Control flow-oriented style
Exploratory style
1950
1960
1970
1980
2000
Year

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.

6. Systematic Testing

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

7. Improved Documentation and Visibility

- ➢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.