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**).

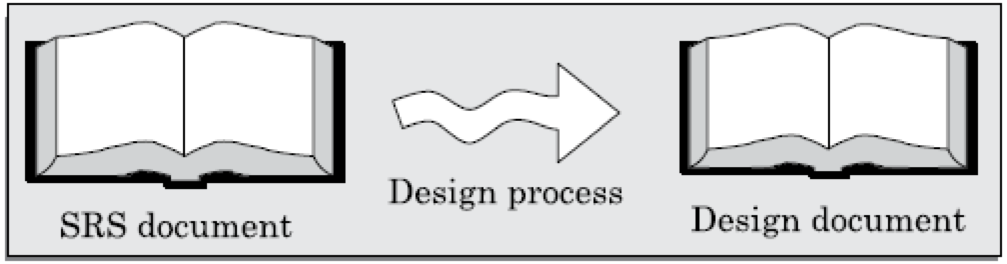


FIGURE 4.1 The design process.

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.

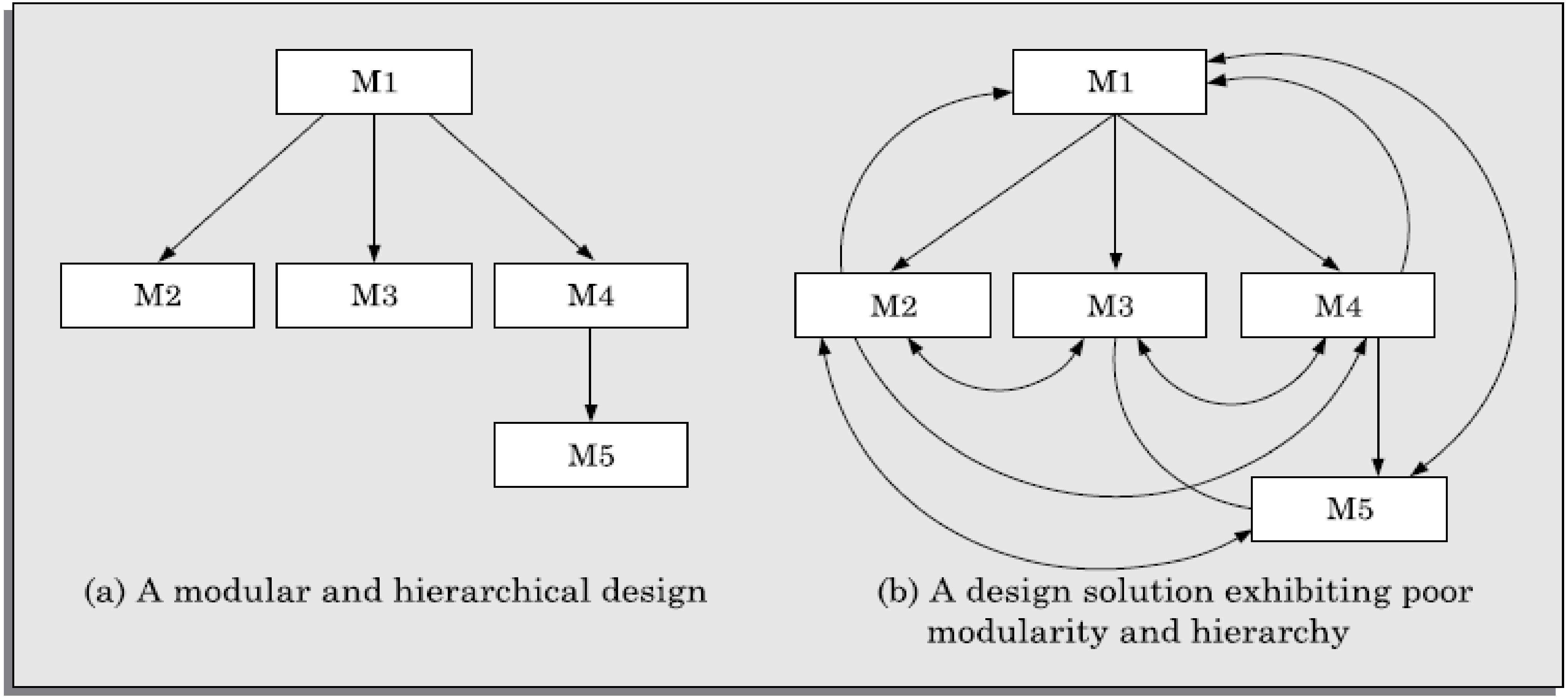


FIGURE 4.2 Two design solutions to the same problem.

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).

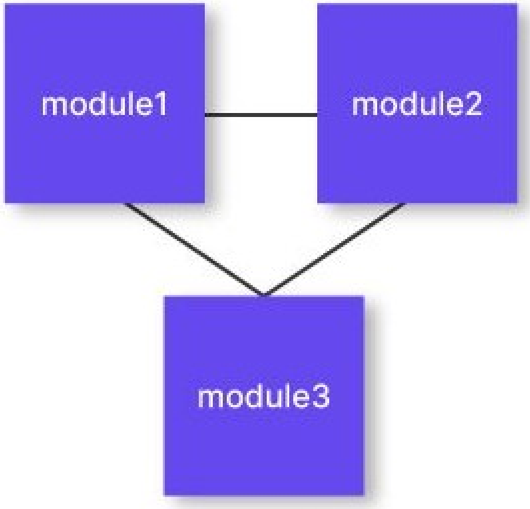
Benefits:

- 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



Coupling

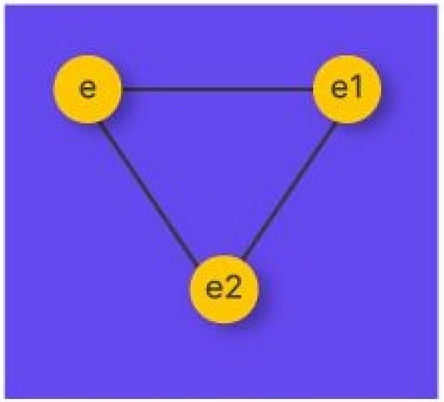
🔹 Overview

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).

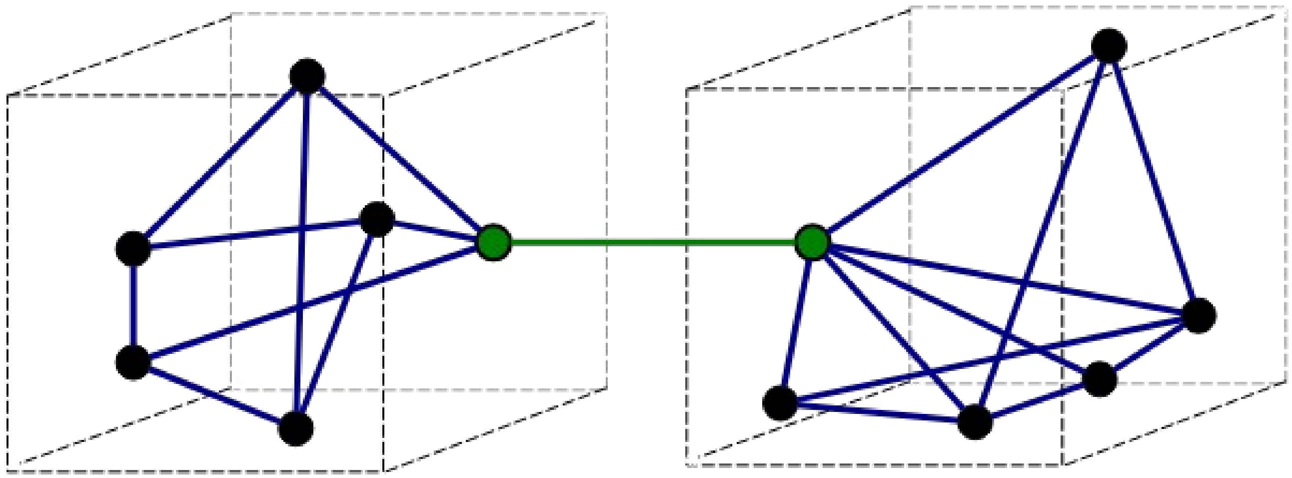
✅ Coupling – *Interdependence between modules*

- 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).

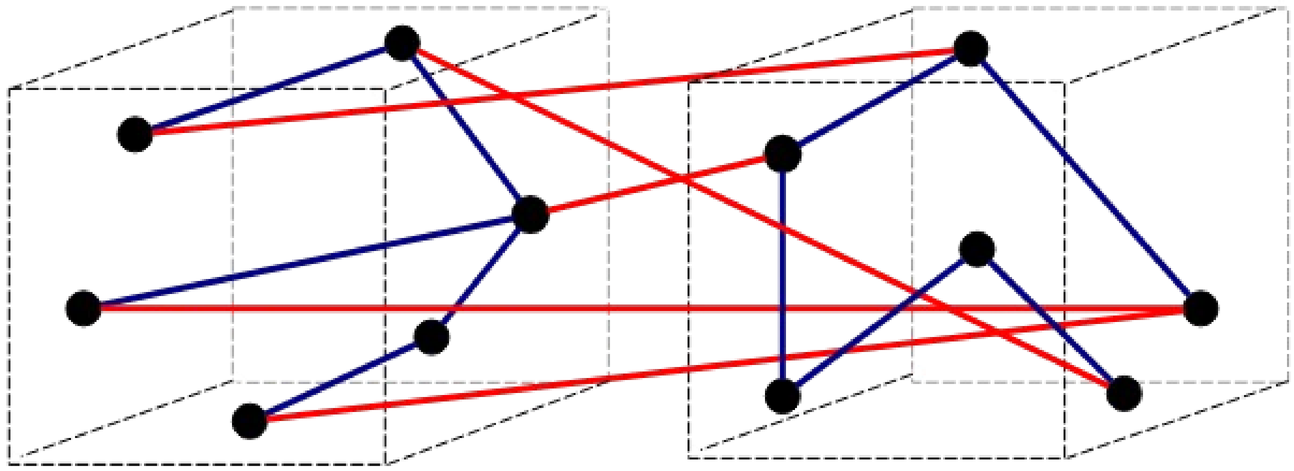


Cohesion

4.3 COHESION AND COUPLING



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

✳️ **<u>Coupling indicates:</u>**

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

✅ **<u>Cohesion – *Internal strength of a module*</u>**

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.

✳️ **<u>Cohesion indicates:</u>**

- 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).

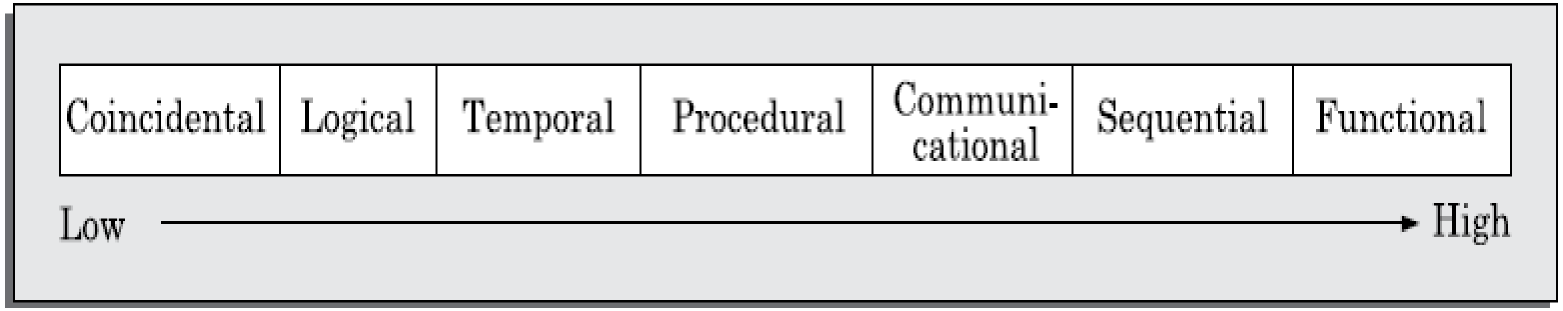
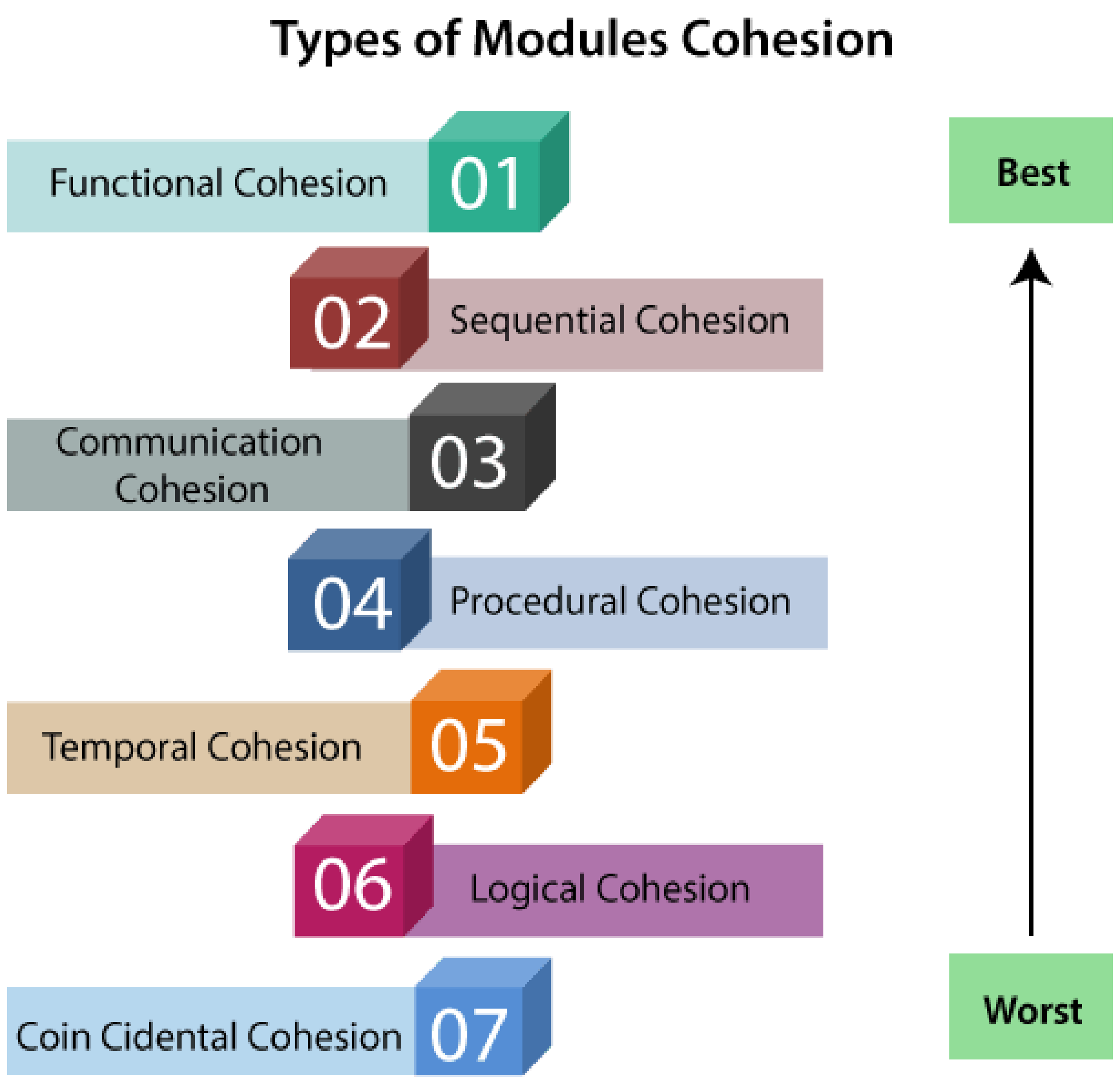
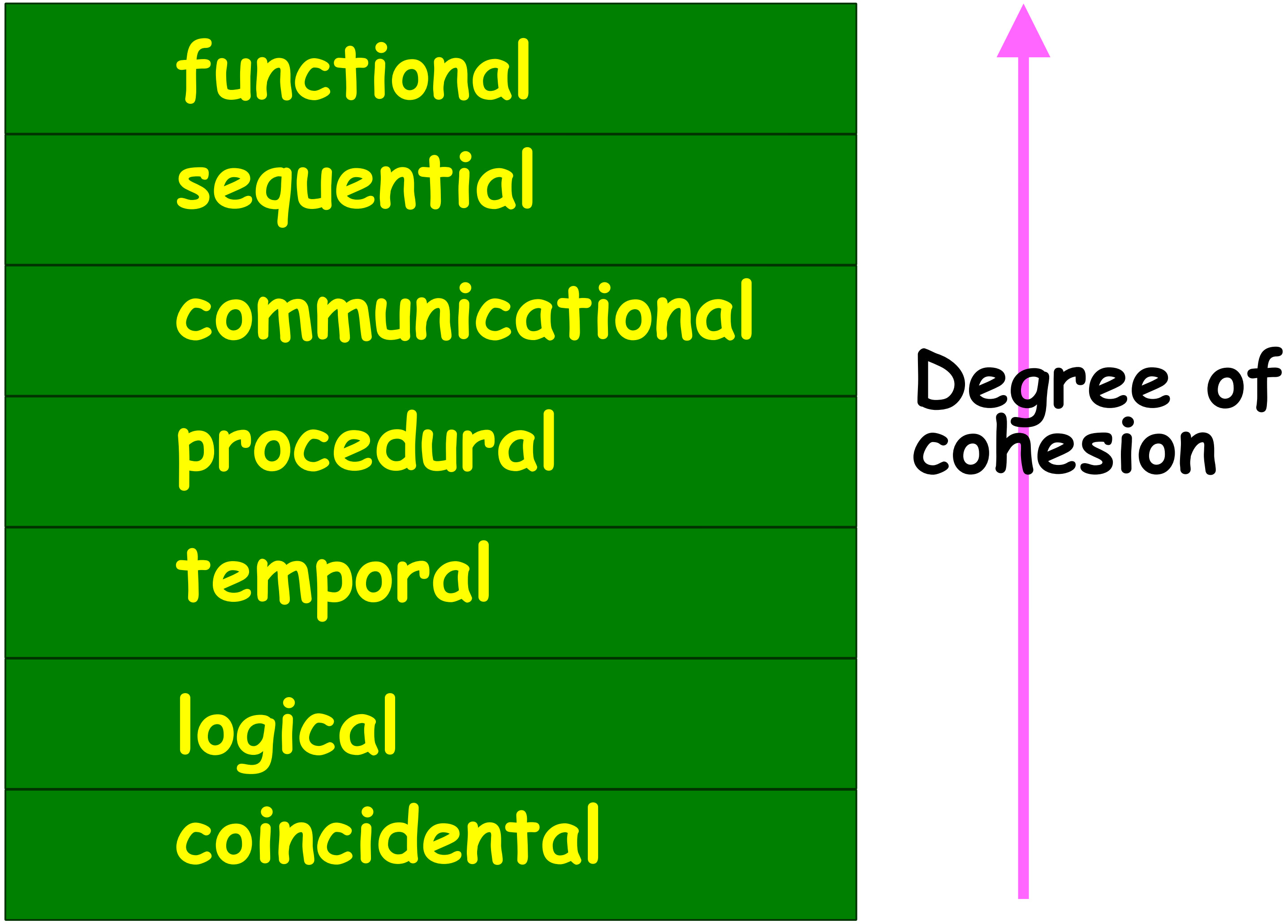


FIGURE 4.3 Classification of cohesion.

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

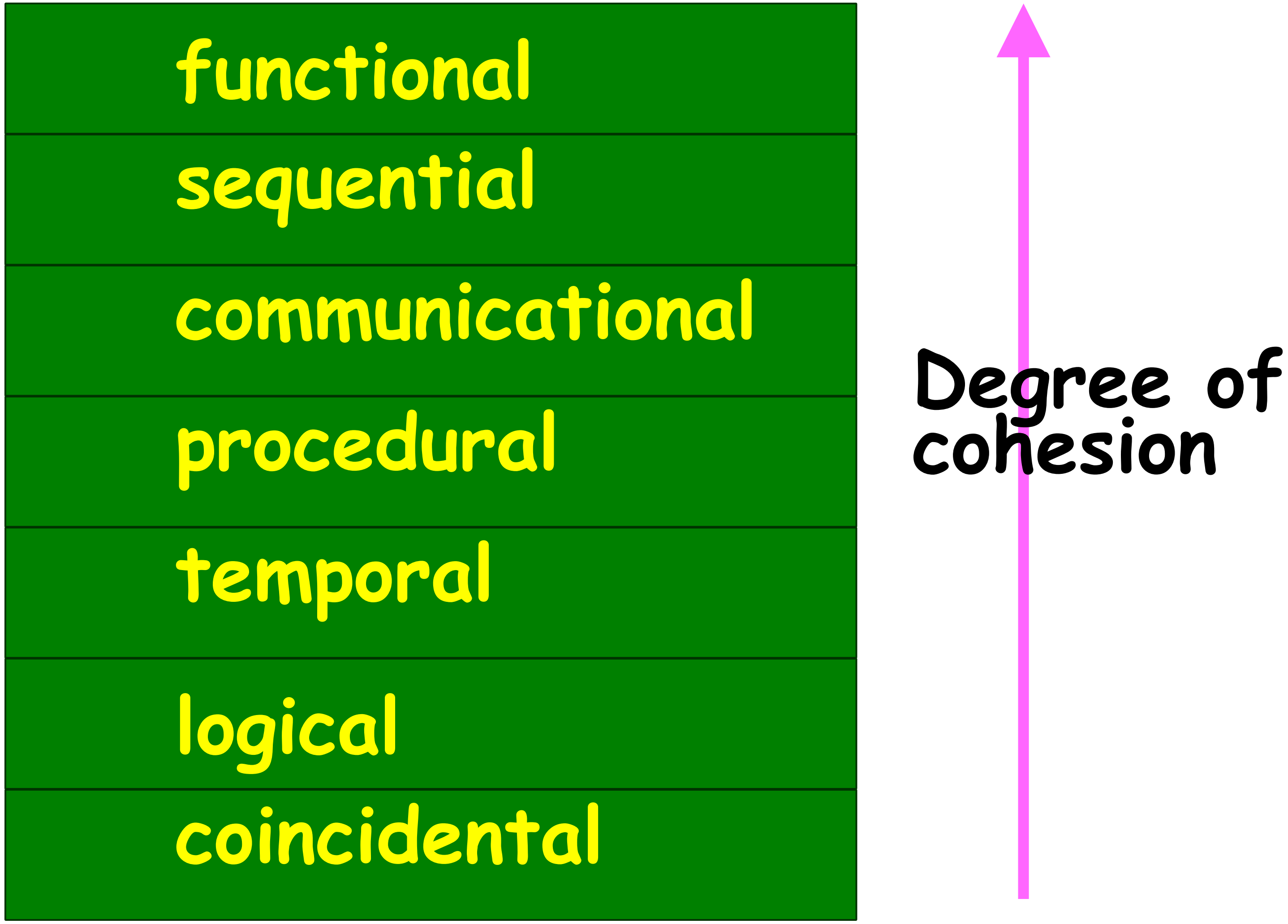
```
Module AAA{

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.

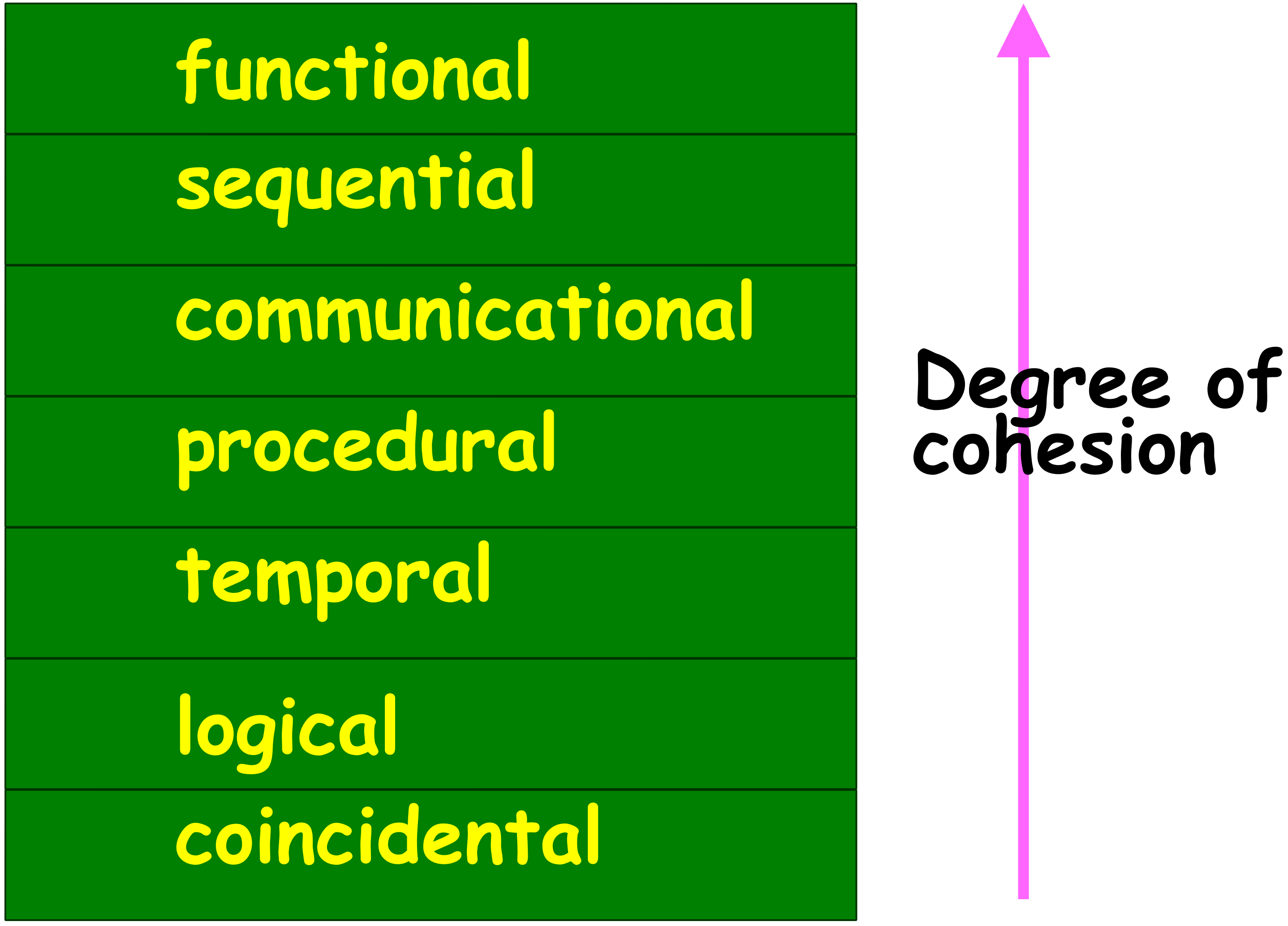
Logical Cohesion

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

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

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

Classification of Cohesiveness



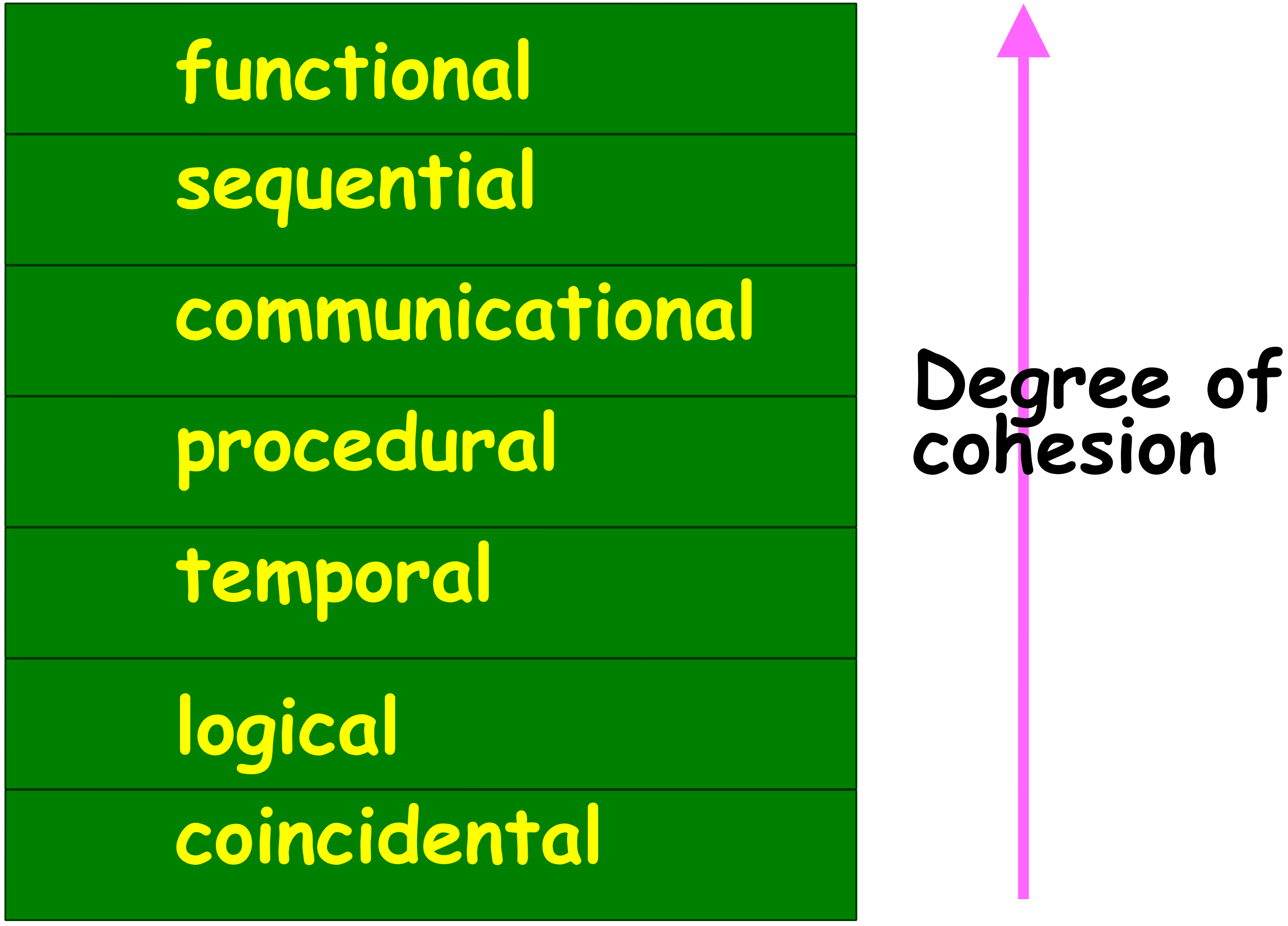
Temporal cohesion

- 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

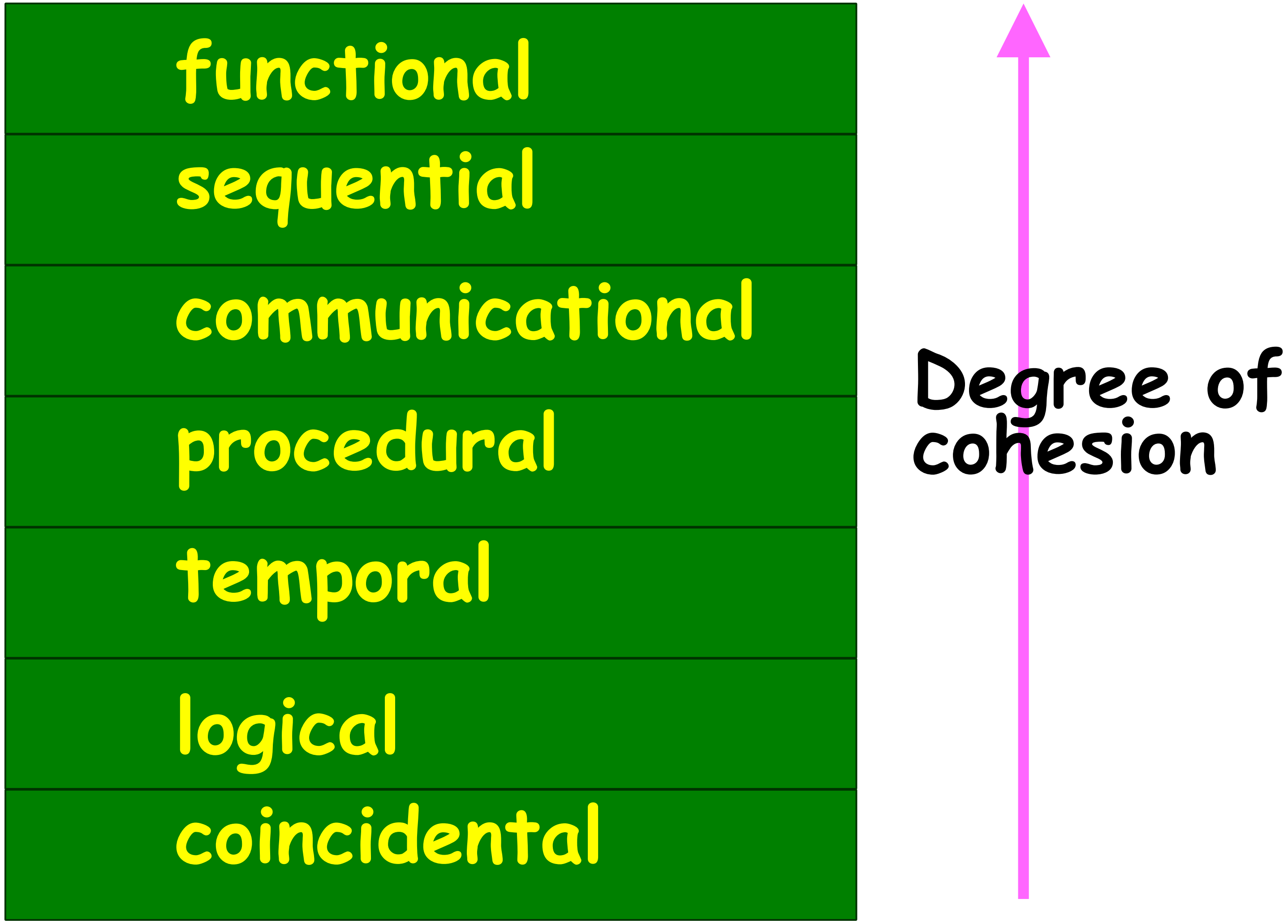
- The set of functions of the module:
  - all part of a procedure (algorithm)
  - certain sequence of steps have to be carried out in a certain order for achieving an objective,
    - e.g. the algorithm for decoding a message.

Procedural Cohesion - example

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

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

Classification of Cohesiveness

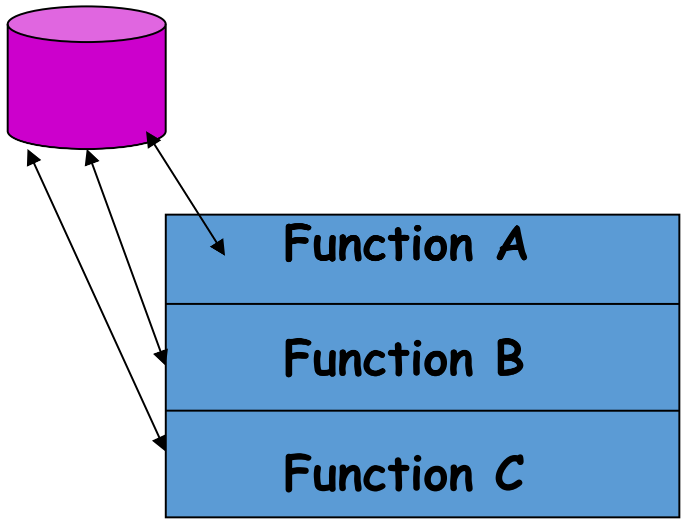


Communicational cohesion

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

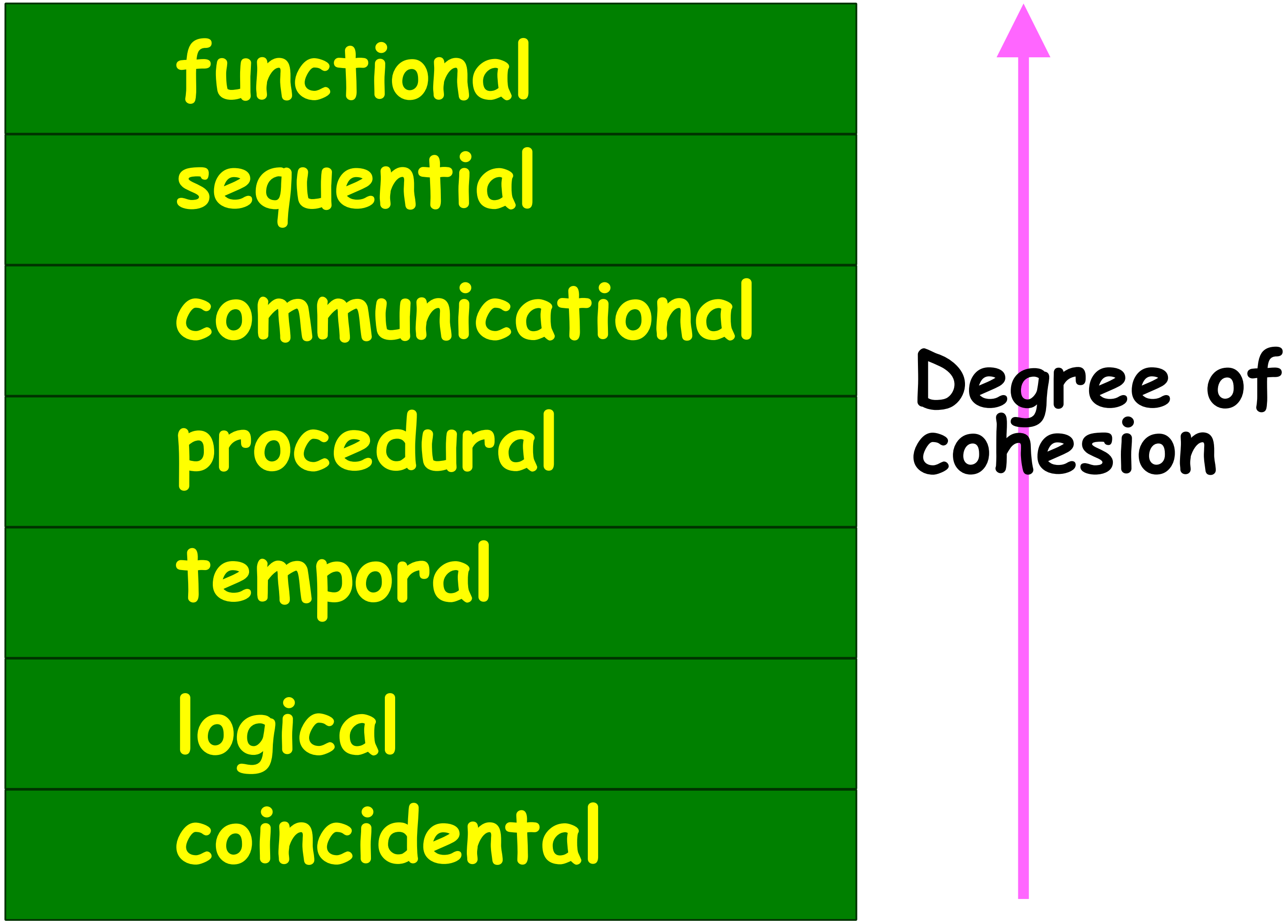
Communicational Cohesion

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



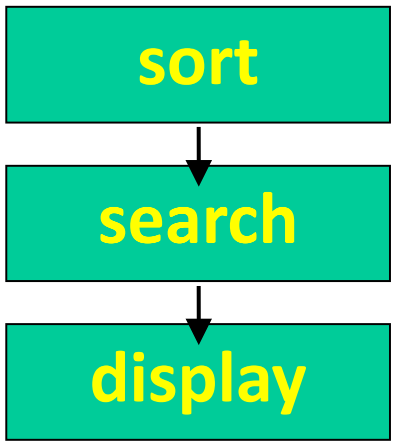
Communicational
Access same data

Classification of Cohesiveness

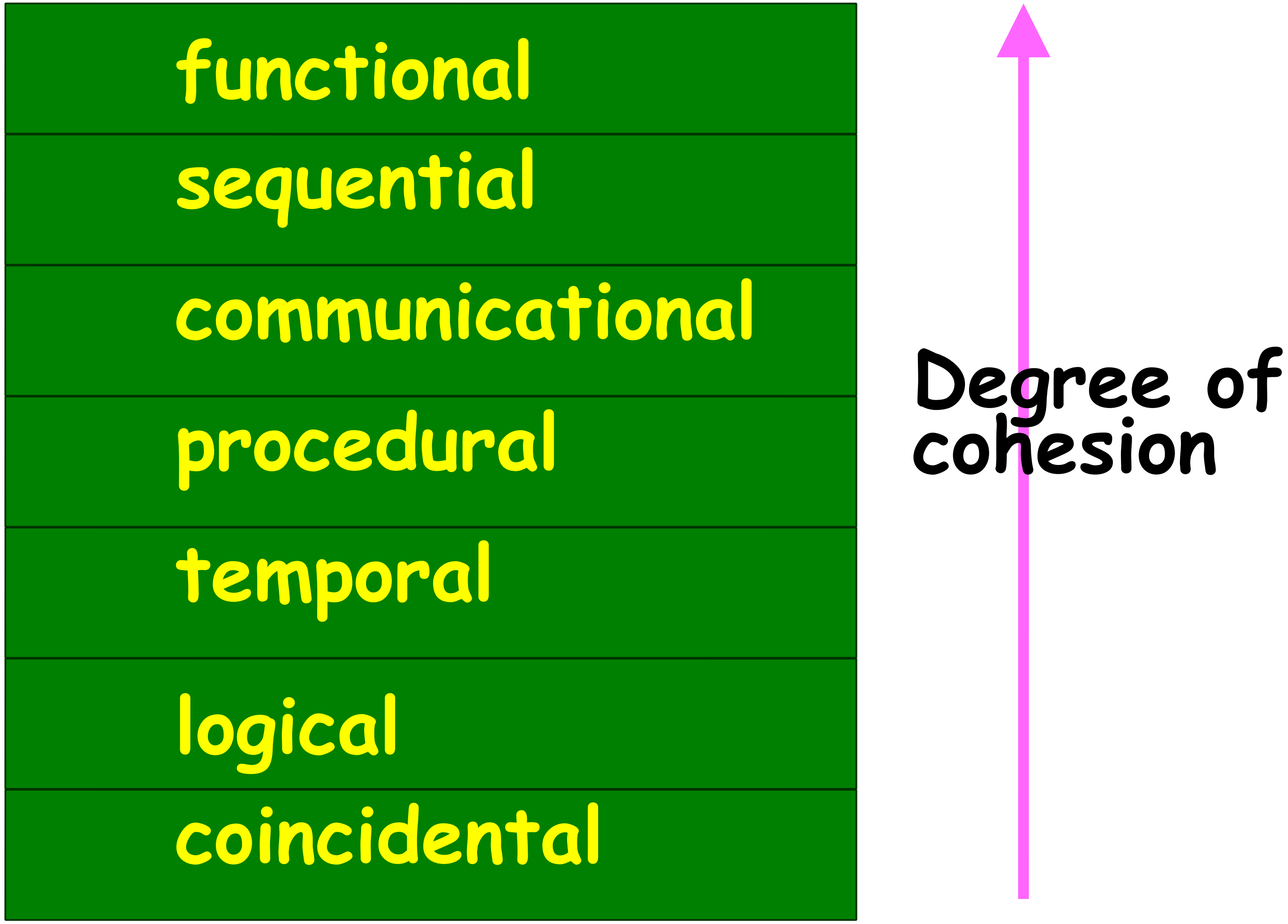


Sequential cohesion

- 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.

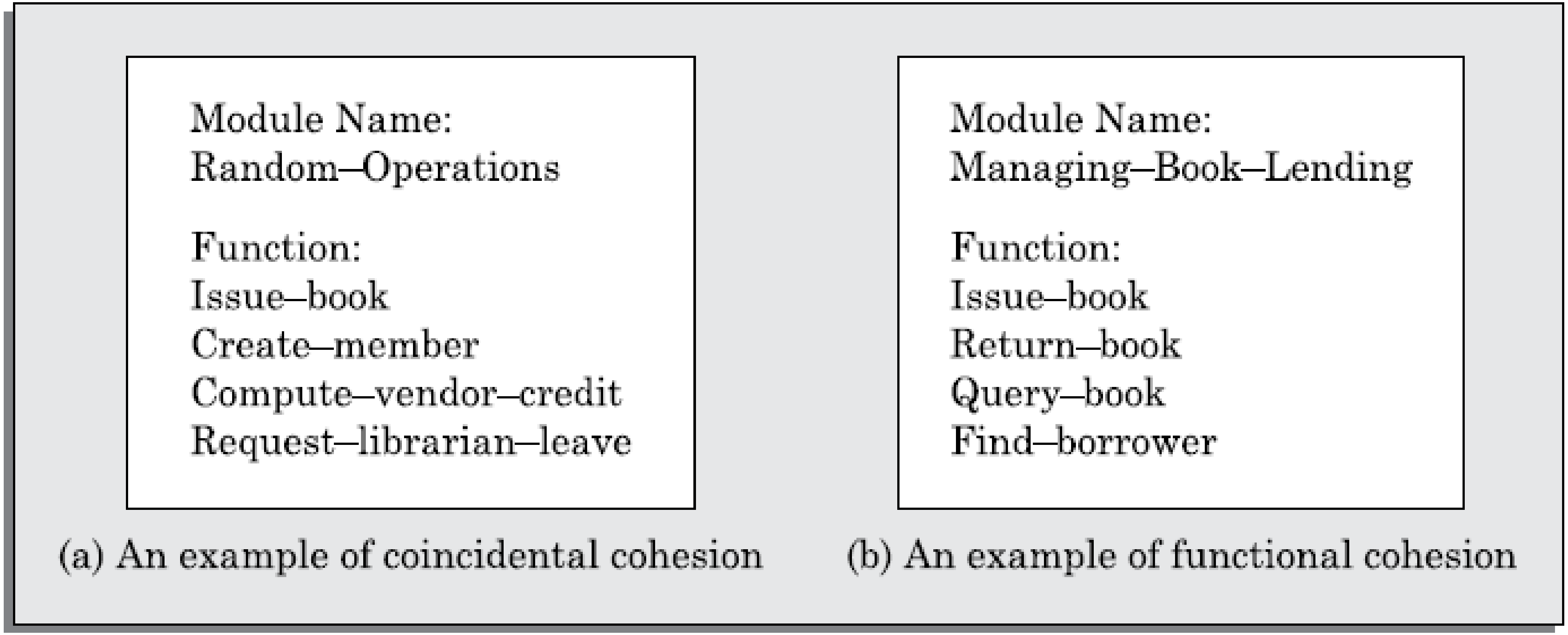


FIGURE 4.4 Examples of cohesion.

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.

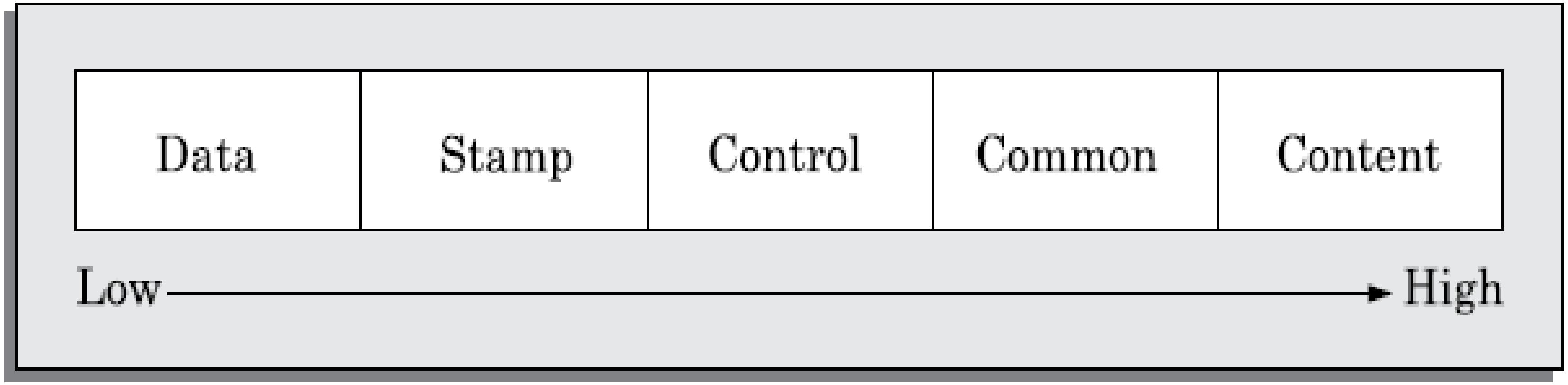
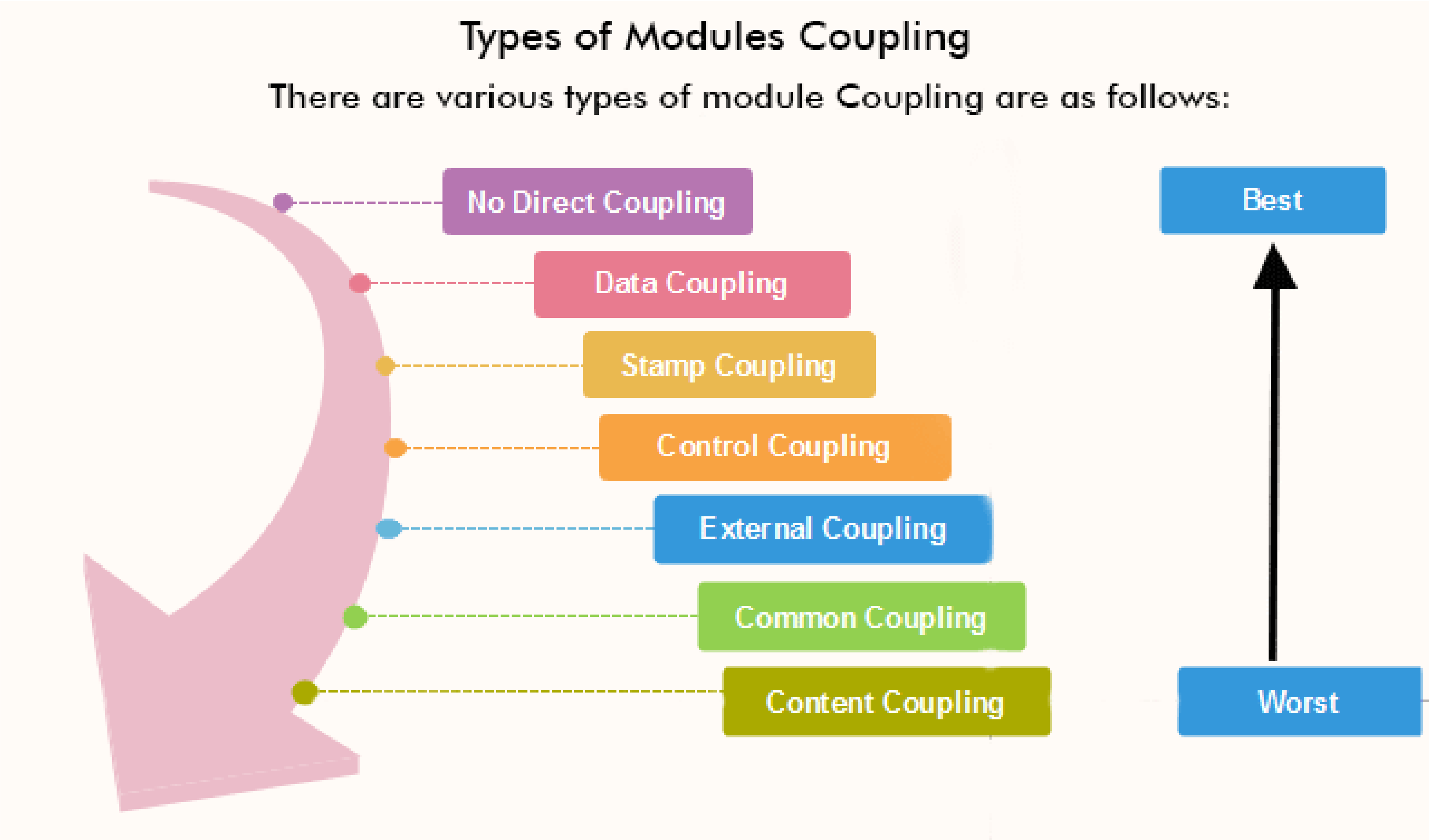
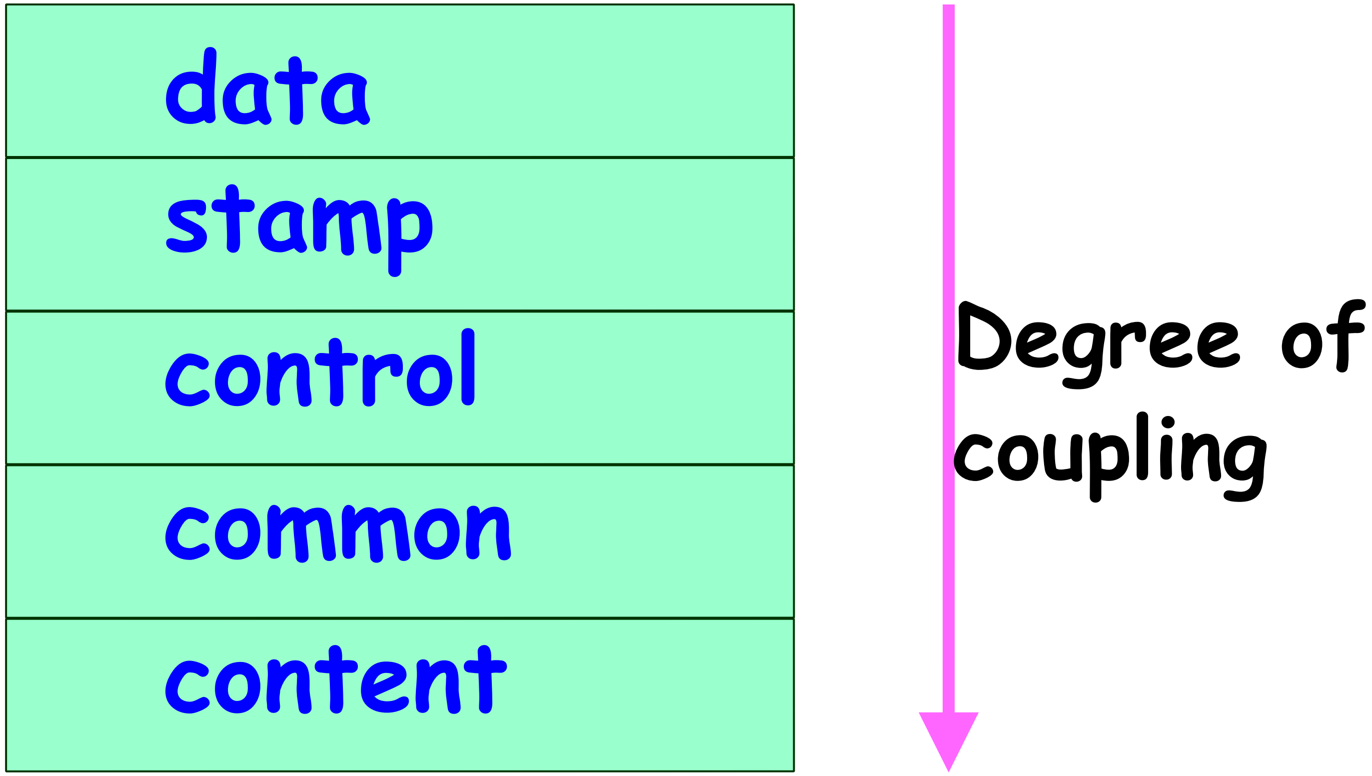


FIGURE 4.5 Classification of coupling.

4.3.2 Classification of Coupling (from best to worst):



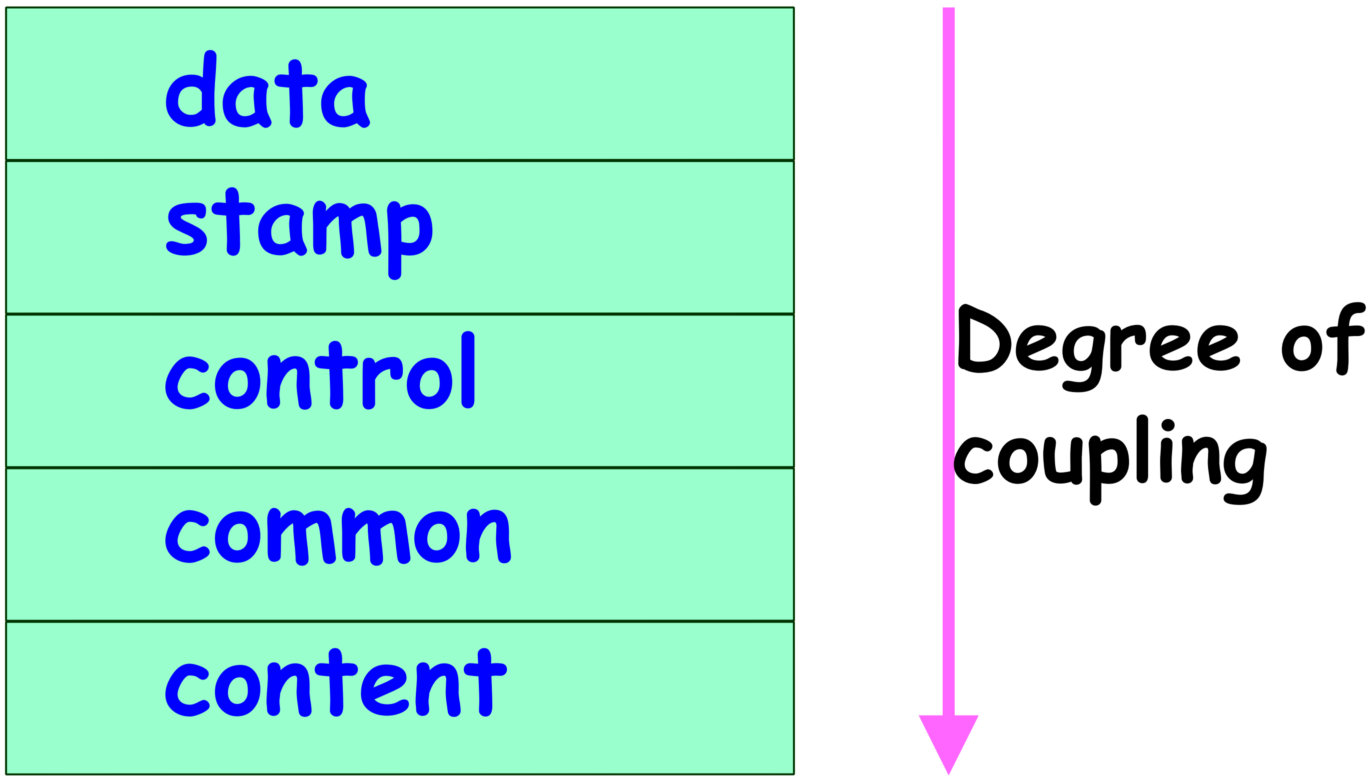
Classes of coupling



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.

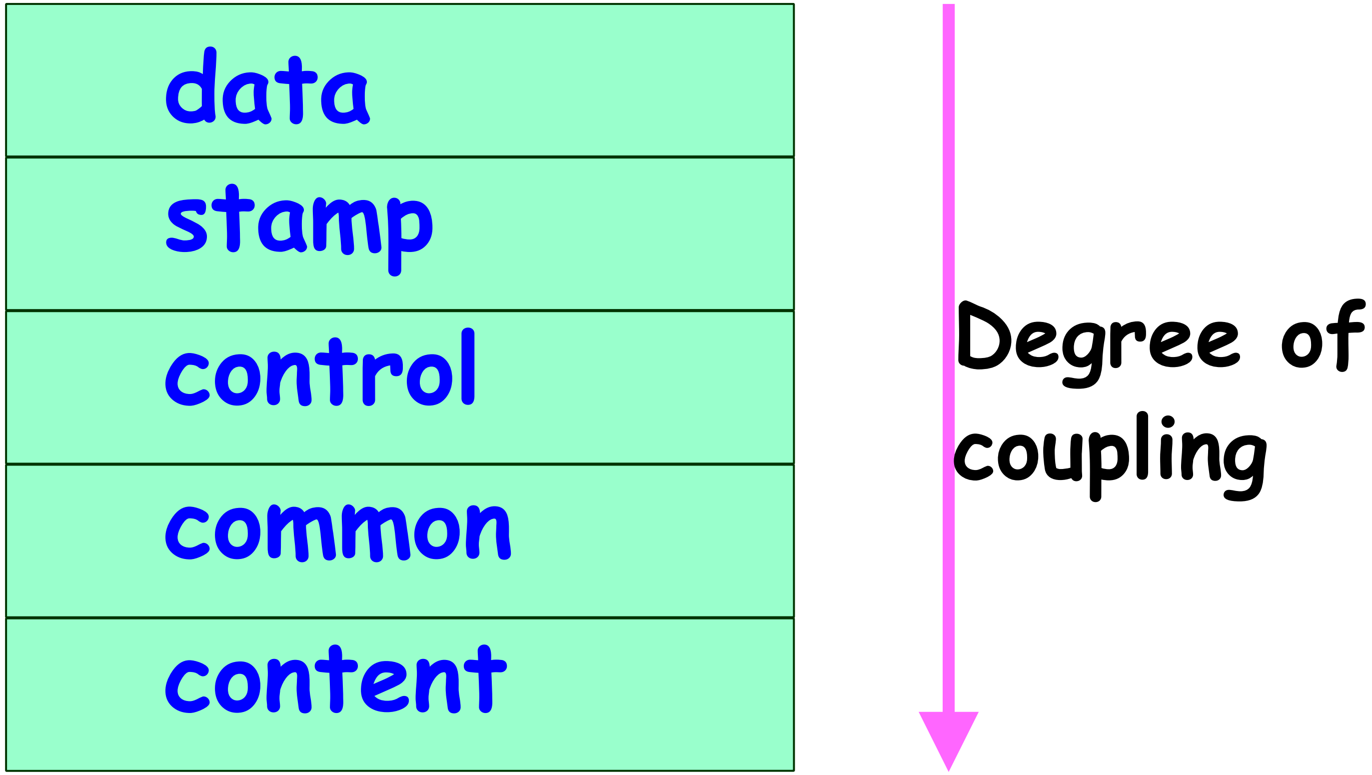
Classes of coupling



Stamp coupling

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

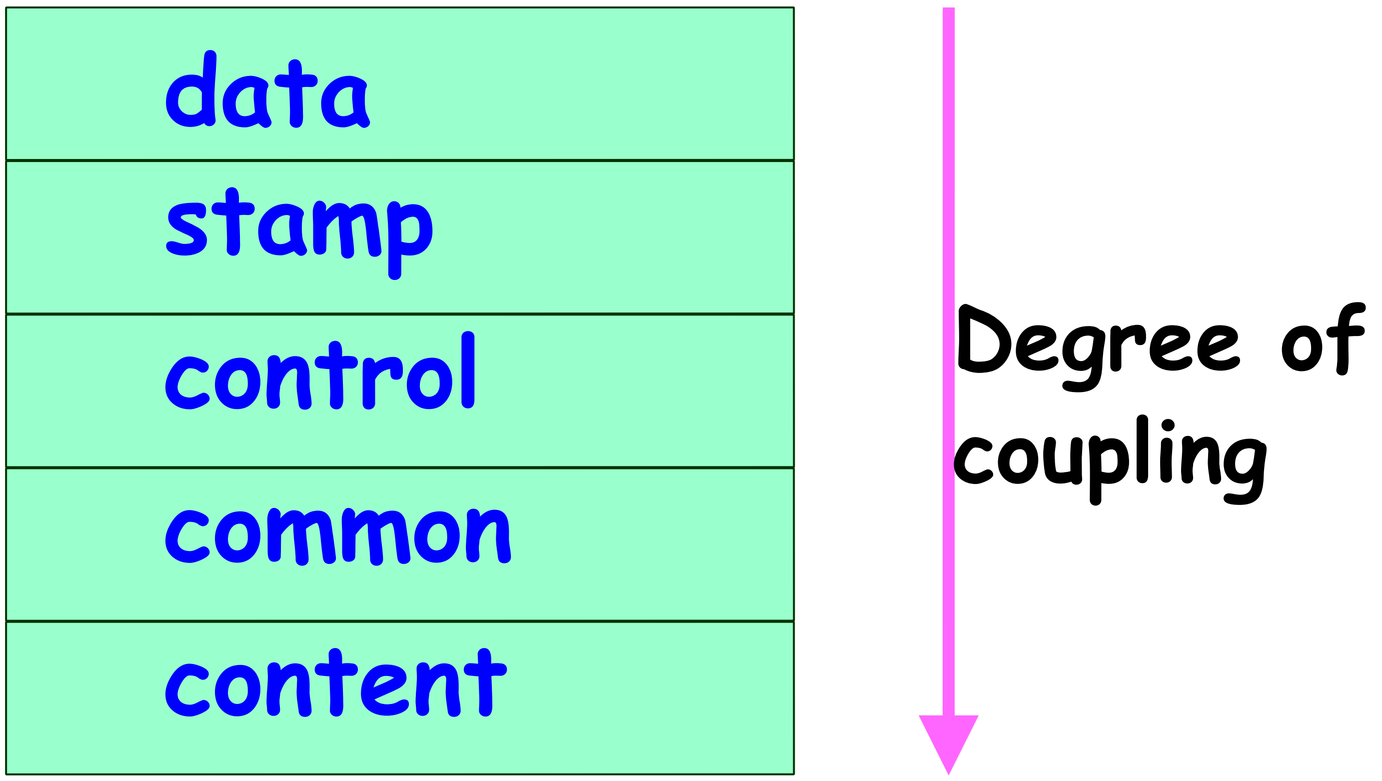
Classes of coupling



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

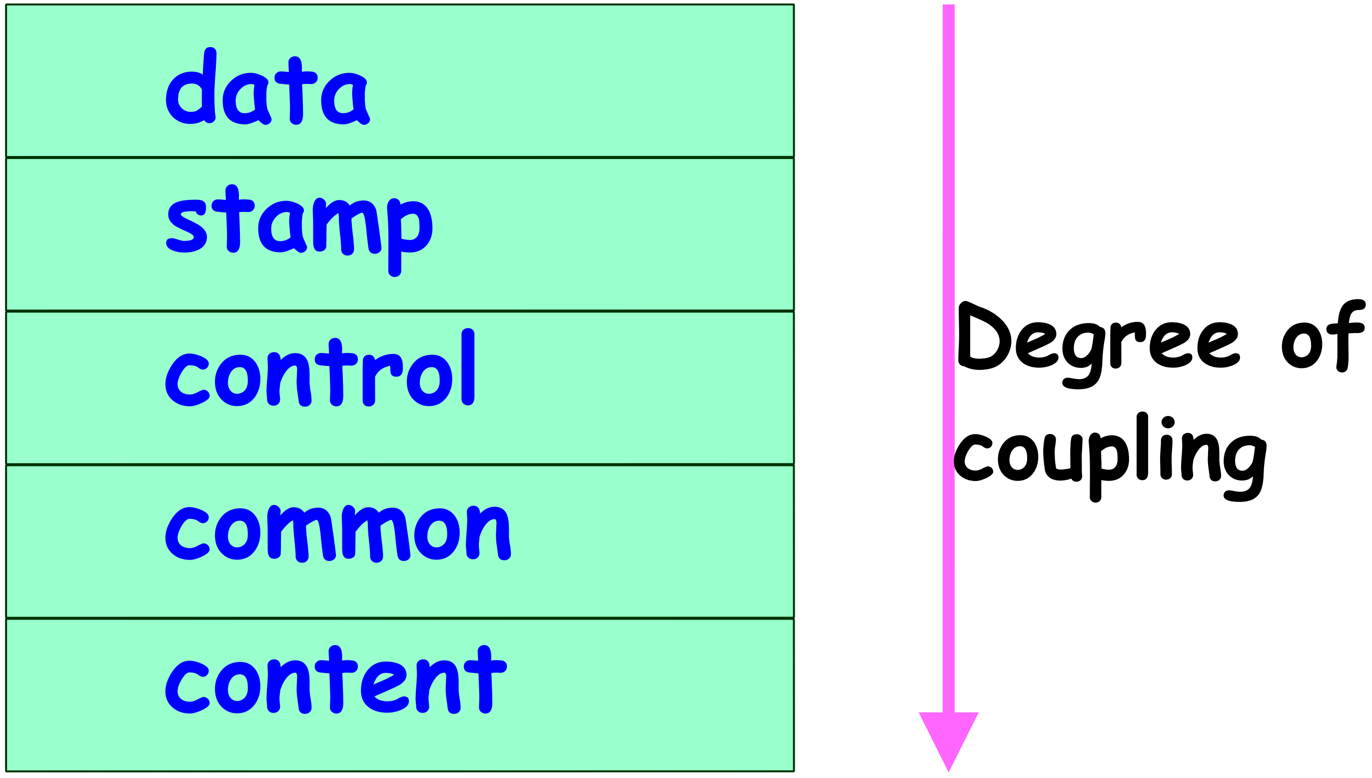


Common Coupling

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

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

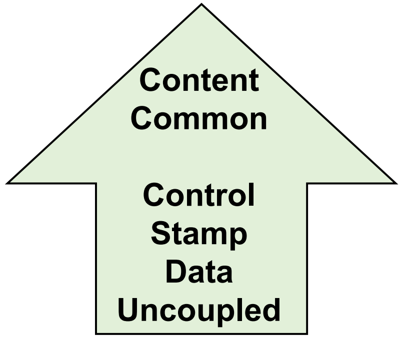
Classes of coupling

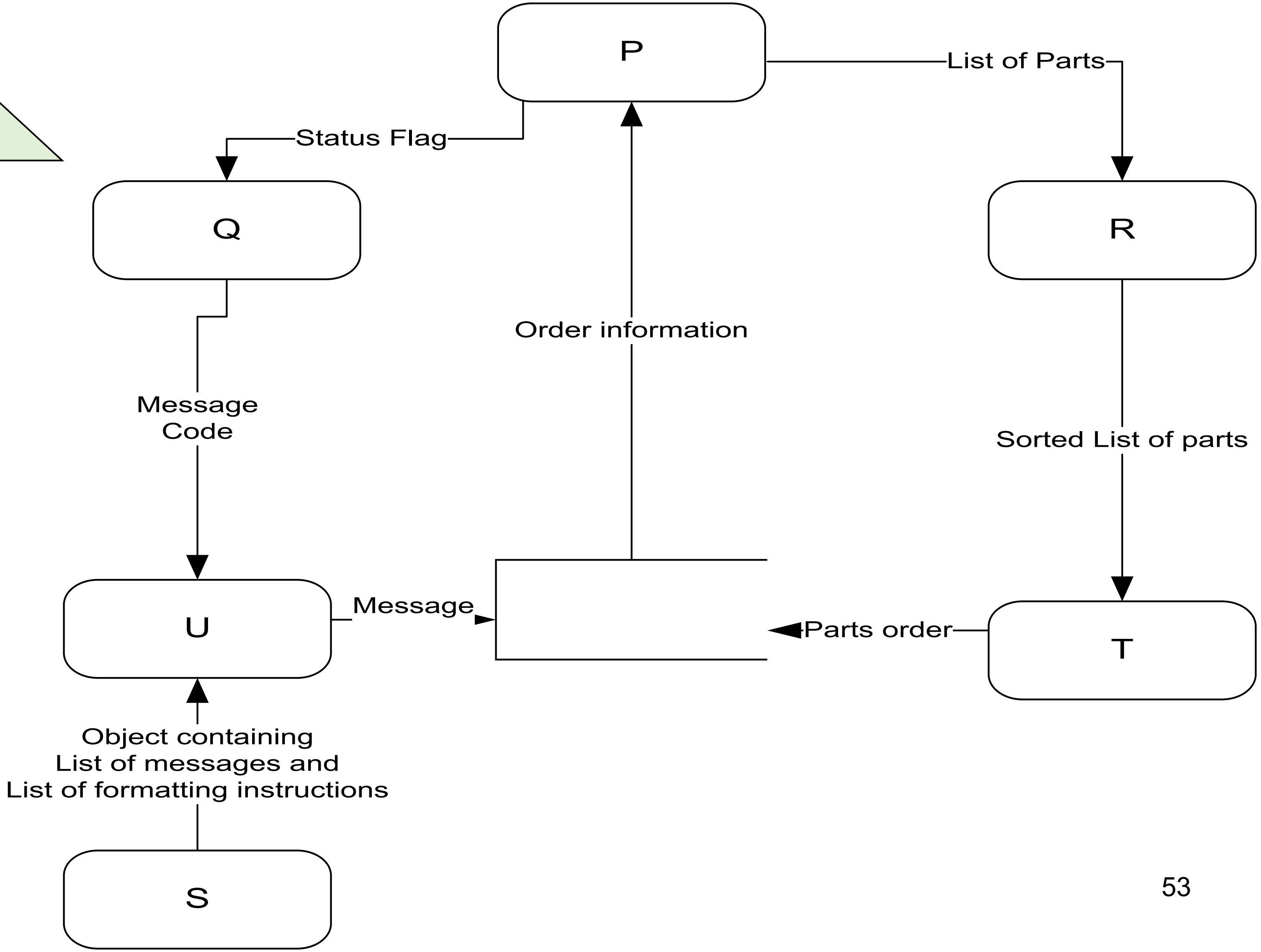


Content coupling

- 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.

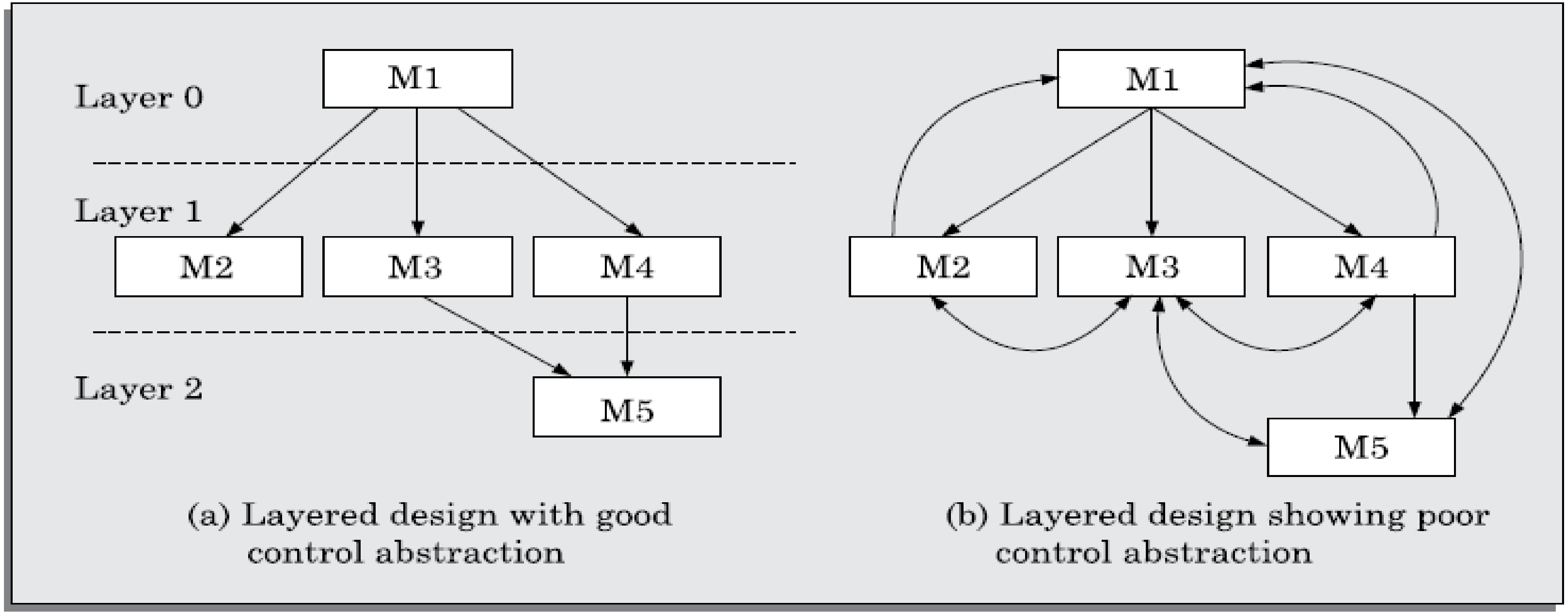


FIGURE 4.6 Examples of good and poor control abstraction.

✅ 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.