Design Concepts and Principles

Instructor: Mehroze Khan

Design Principles

- **Design principles** are guidelines for decomposing a system's required functionality and behavior into modules
- The principles identify the criteria
  - for decomposing a system
  - deciding what information to provide (and what to conceal) in the resulting modules
- Six dominant principles (general):
  - Modularity
  - Interfaces
  - Information hiding
  - Incremental development
  - Abstraction
  - Generality

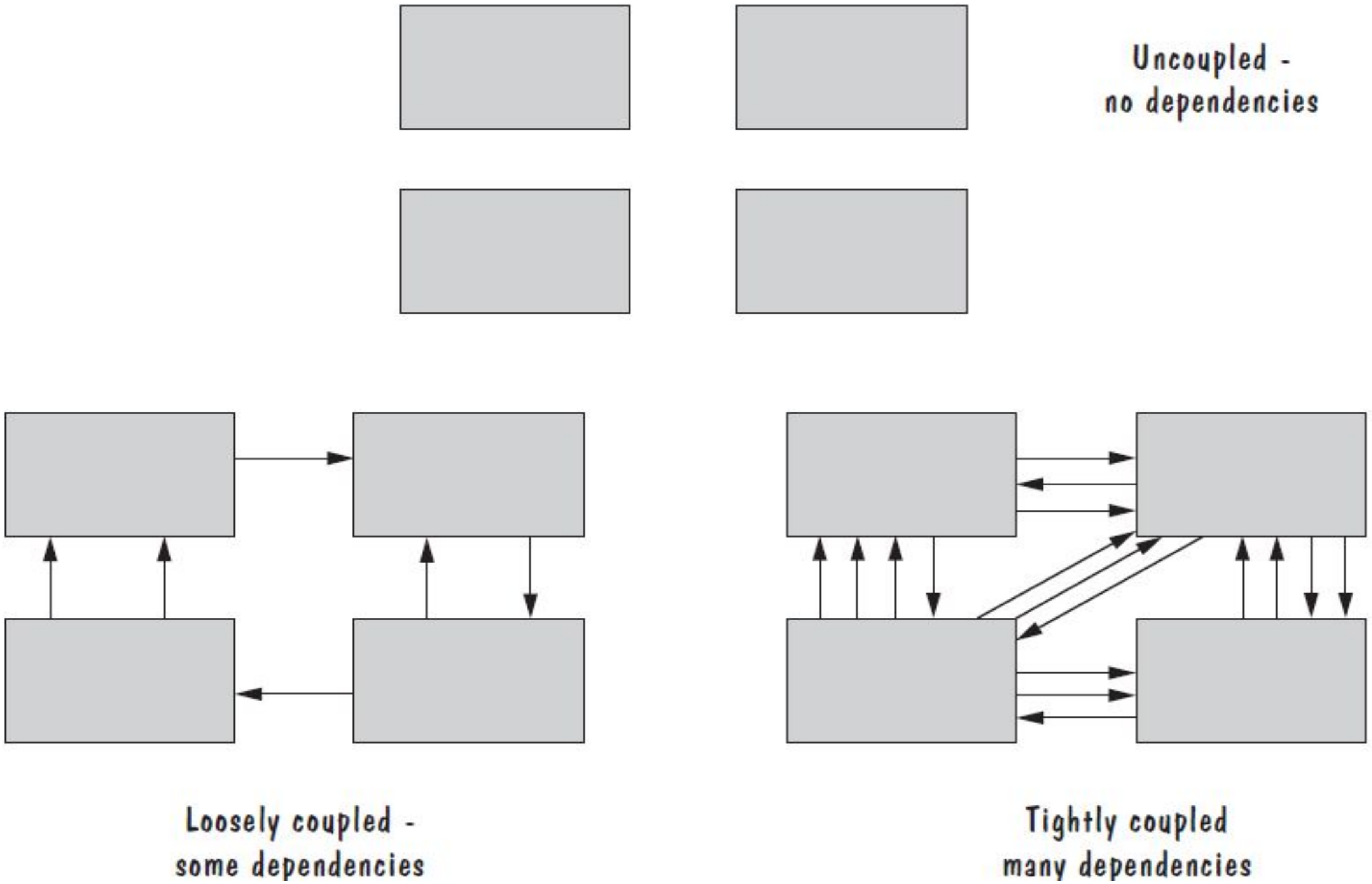
Modularity

- **Modularity** is the principle of keeping the unrelated aspects of a system separate from each other,
  - each aspect can be studied in isolation (also called separation of concerns)
- If the principle is applied well, each resulting module will have a **single purpose** and will be relatively **independent** of the others
  - Each module will be easy to **understand** and **develop**
  - Easier to **locate faults**
    - because there are fewer suspect modules per fault
  - Easier to **change** the system
    - because a change to one module affects relatively few other modules
- To determine how well a design separates concerns, we use two concepts that measure **module independence**: coupling and cohesion

Modularity: Coupling

- Two modules are **tightly coupled** when they depend a great deal on each other
- **Loosely coupled** modules have some dependence, but their interconnections are weak
- **Uncoupled** modules have no interconnections at all; they are completely unrelated

Modularity: Coupling



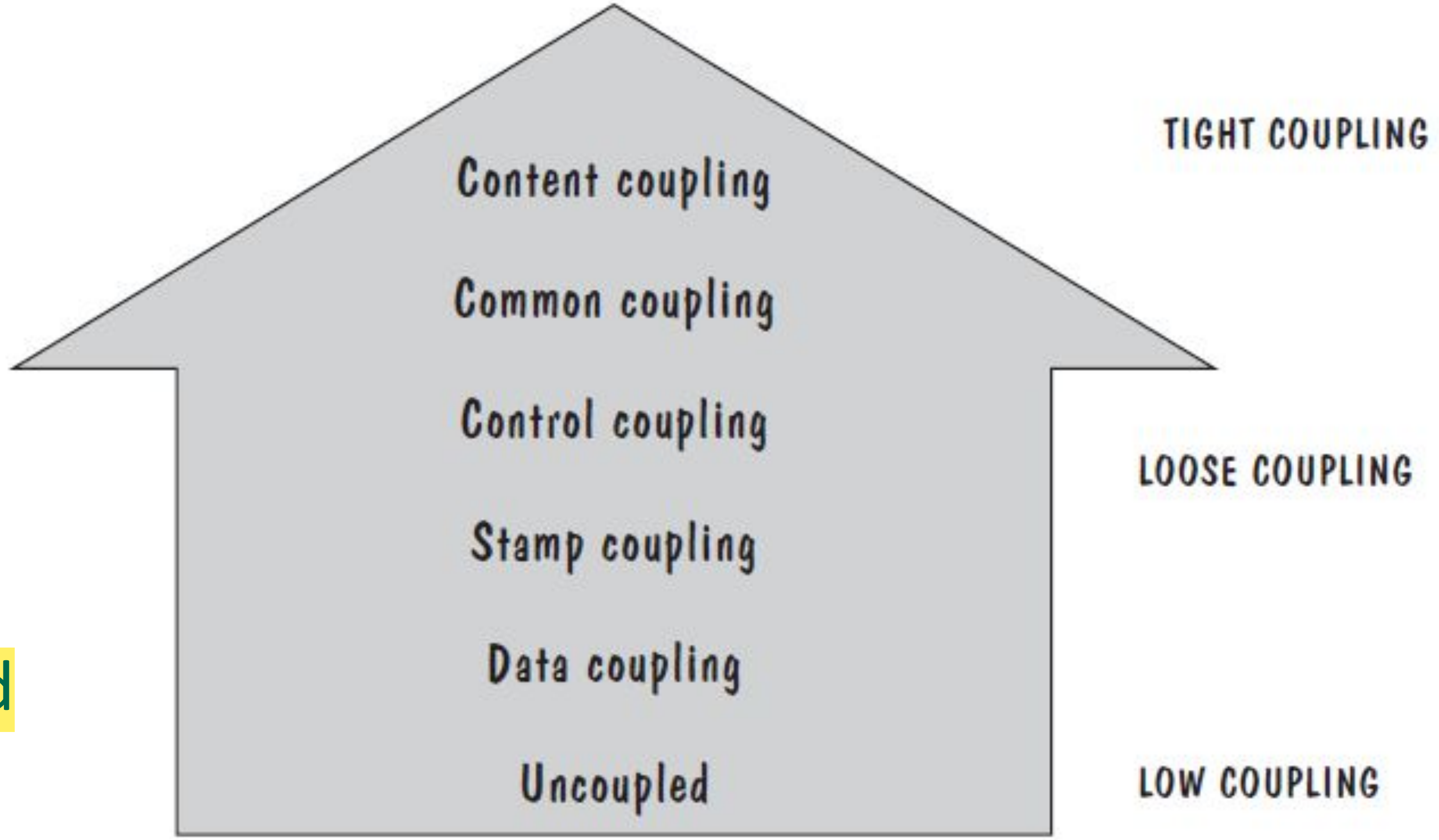
Modularity: Coupling

- There are many ways that modules can depend on each other:
  - The references made from one module to another
  - The amount of data passed from one module to another
  - The amount of control that one module has over the other
- Coupling can be measured along a spectrum of dependence, ranging from complete dependance to complete independence

Modularity: Types of Coupling

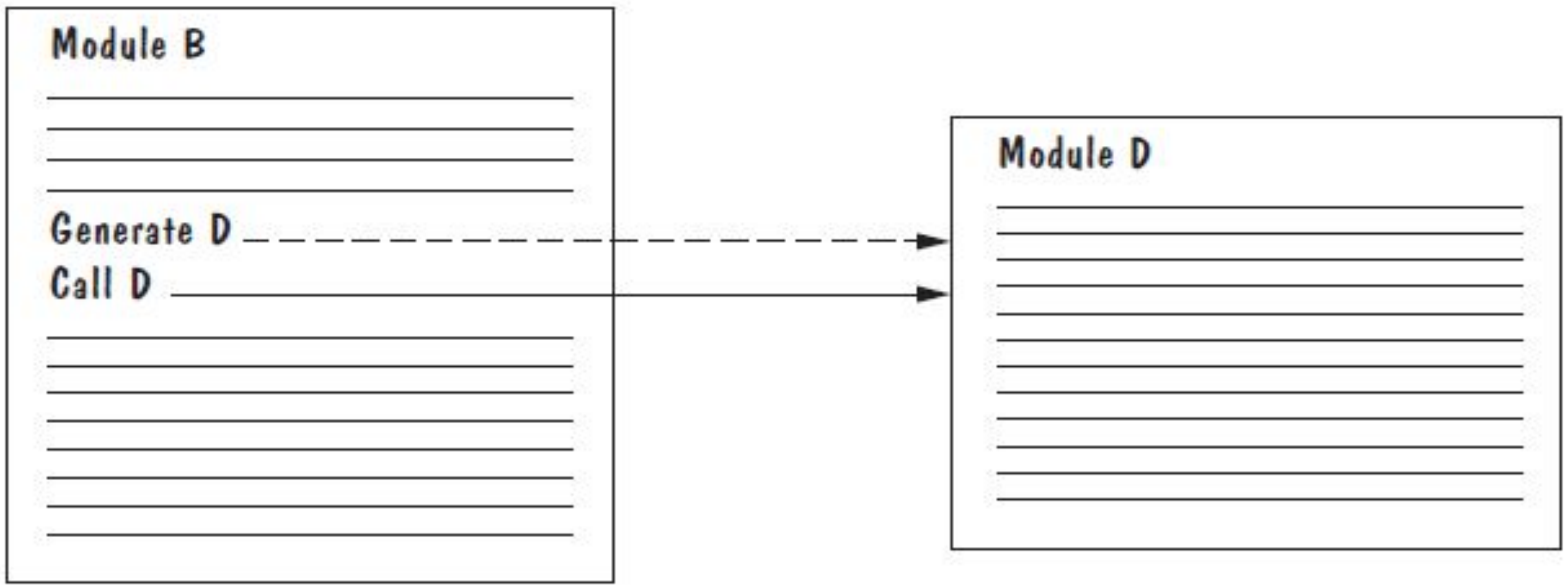
- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling

High coupling is not desired



Modularity: Content Coupling

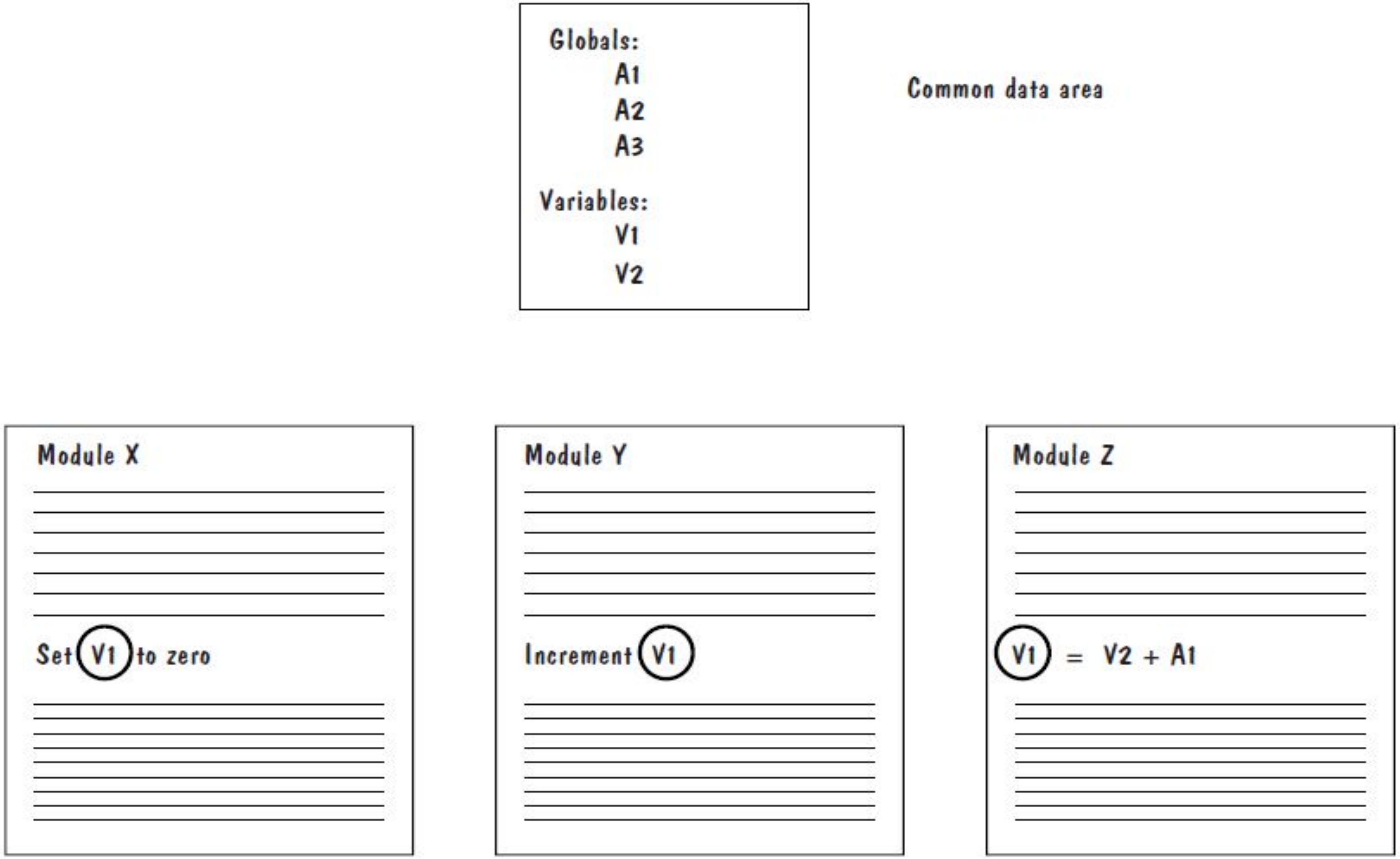
- Content coupling occurs when one module **directly accesses or manipulates the internal workings of another module**, such as its variables or control structures, rather than relying on well-defined interfaces (like functions or methods)
- Content coupling might occur when one module is imported into another module, modifies the code of another module, or branches into the middle of another module



Modularity: Common Coupling

- We can reduce the amount of coupling somewhat by organizing our design so that **data are accessible from a common data store**.
- Dependence still exists; making a change to the common data means that, to evaluate the effect of the change, we must look at all modules that access those data.
- With common coupling, it can be difficult to determine which module is responsible for having set a variable to a particular value.

Modularity: Common Coupling



Modularity: Control Coupling

- When one module passes **parameters** or a **return code** to control the behavior of another module
- It is impossible for the controlled module to function without some direction from the controlling module
- Limit each module to be responsible for only one function or one activity.
- Restriction minimizes the amount of information that is passed to a controlled module
- It simplifies the module's interface to a fixed and recognizable set of parameters and return values.

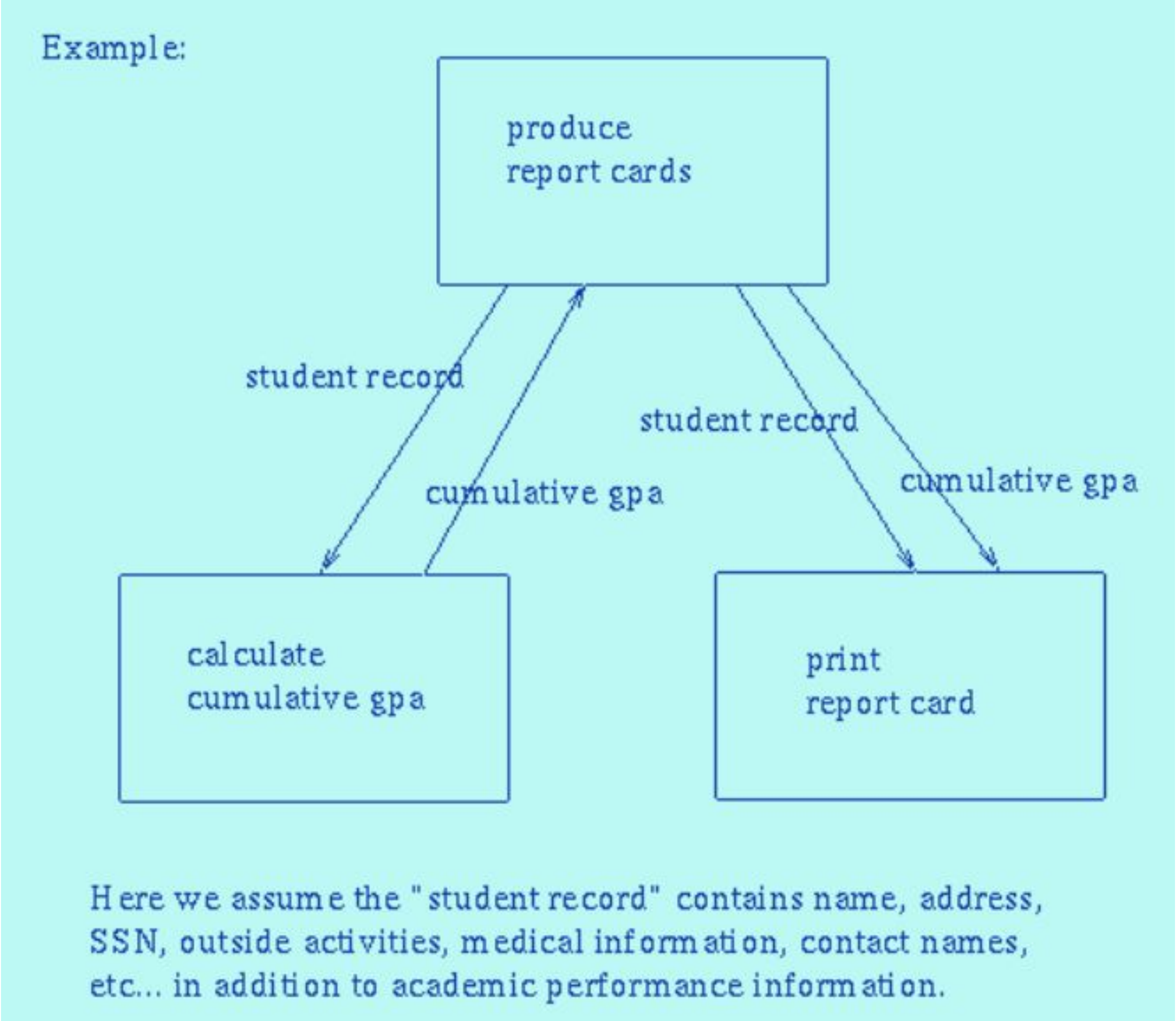
Modularity: Control Coupling

```
bool foo(int x){
    if (x == 0)
        return false;
    else
        return true;
}

void bar(){
    // Calling foo() by passing a value which controls its flow:
    foo(1);
}
```

Modularity: Stamp Coupling

- When complex data structures are passed between modules, we say there is **stamp coupling** between the modules
  - Stamp coupling represents a more complex interface between modules, because the modules have to agree on the data's format and organization



Modularity: Stamp Coupling

- When the signature of one of Class B's functions has class A as its argument or return type.

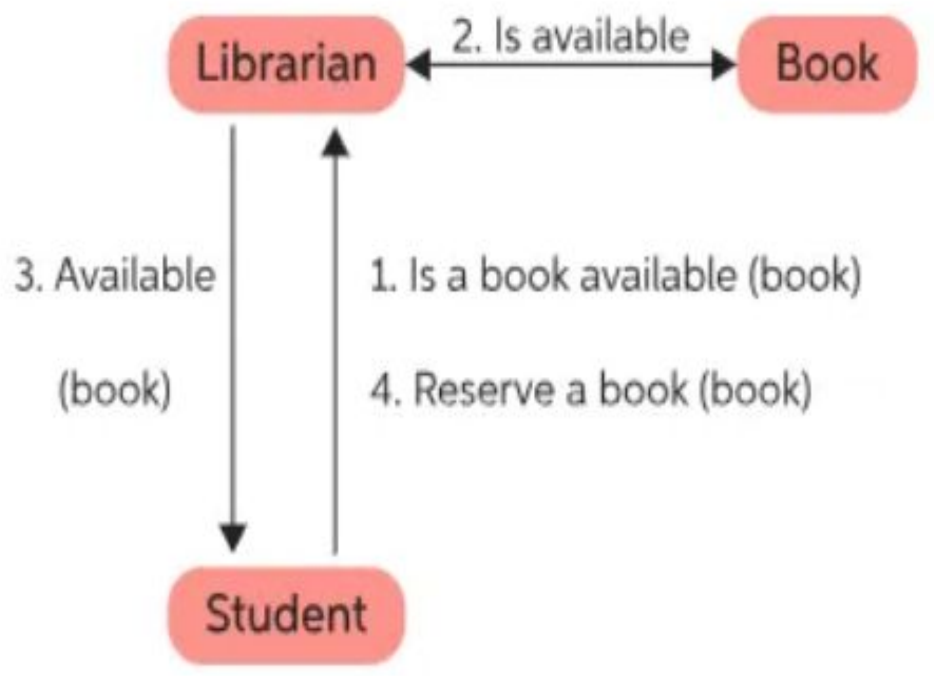
```
class A{
  // Code for class A.
};

class B{
    // Data member of class A type: Type-use coupling
    A var;

    // Argument of type A: Stamp coupling
    void calculate(A data){
        // Do something.
    }
};
```

Modularity: Data Coupling

- If only data values, and not structured data, are passed, then the modules are connected by **data coupling**
  - Data coupling is simpler and less likely to be affected by changes in data representation.
  - Easiest to trace data through and to make changes to data coupled modules.



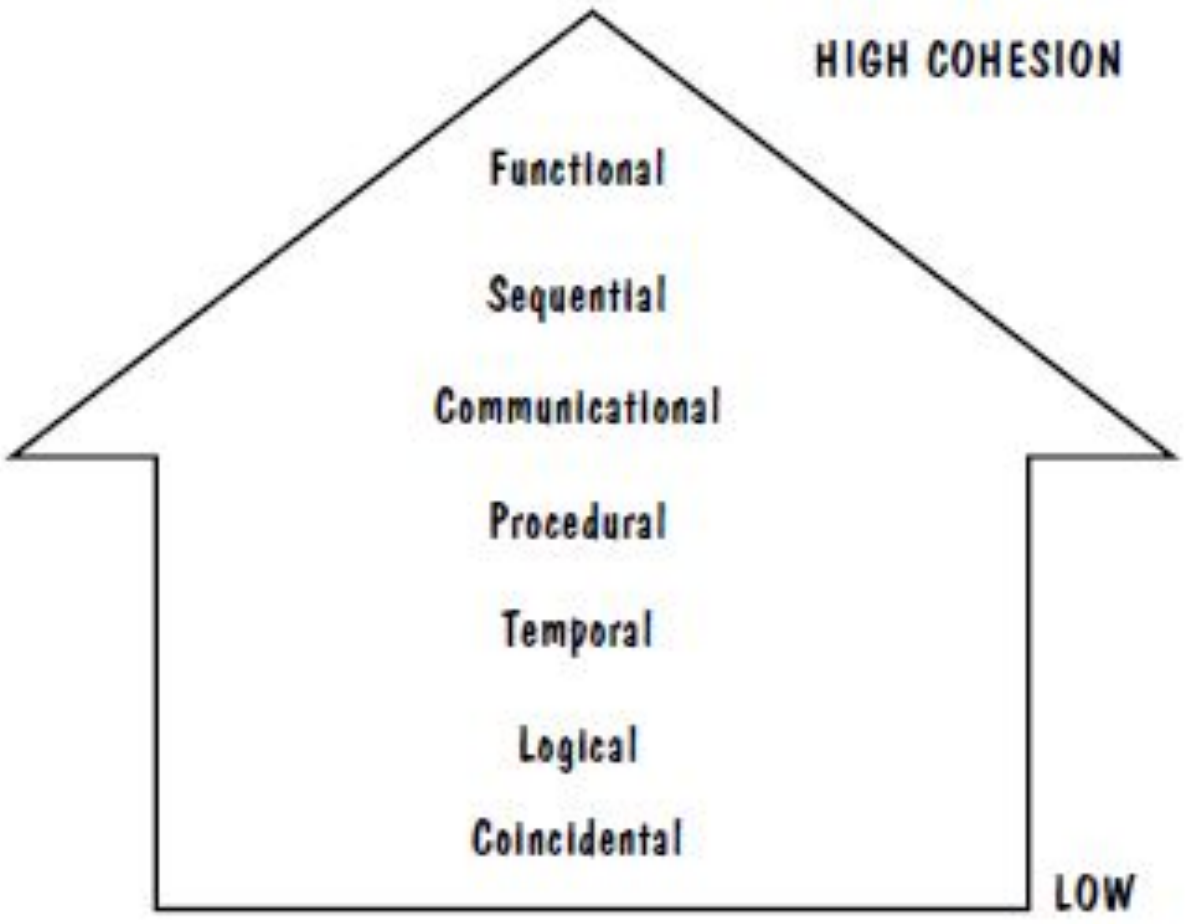
Modularity: Cohesion

- Cohesion refers to the dependence within and among a module’s internal elements (e.g., data, functions, internal modules)
- The more cohesive a module, the more closely related its pieces are

Modularity: Types of Cohesion

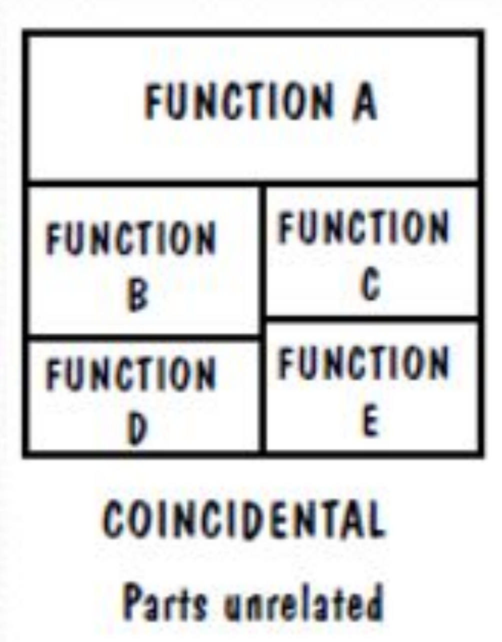
- Coincidental cohesion
- Logical cohesion
- Temporal cohesion
- Procedural cohesion
- Communicational cohesion
- Functional cohesion
- Sequential cohesion

Low cohesion is not desired



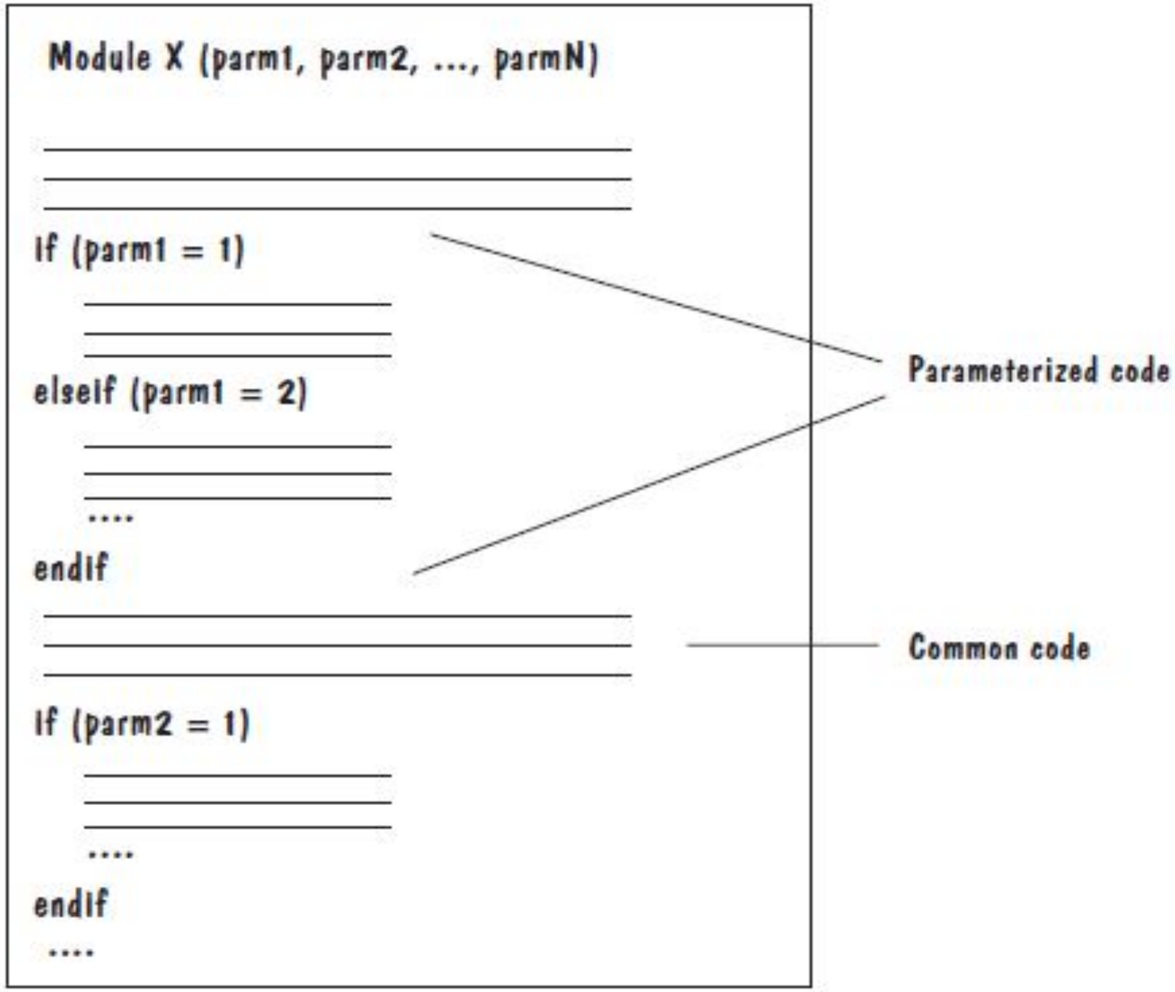
Modularity: Coincidental Cohesion

- The worst degree of cohesion, **coincidental**, is found in a module whose parts are unrelated to one another
- Unrelated functions, processes, or data are combined in the same module for reasons of convenience



Modularity: Logical Cohesion

- A module has **logical cohesion** if its parts are related only by the logic structure of its code

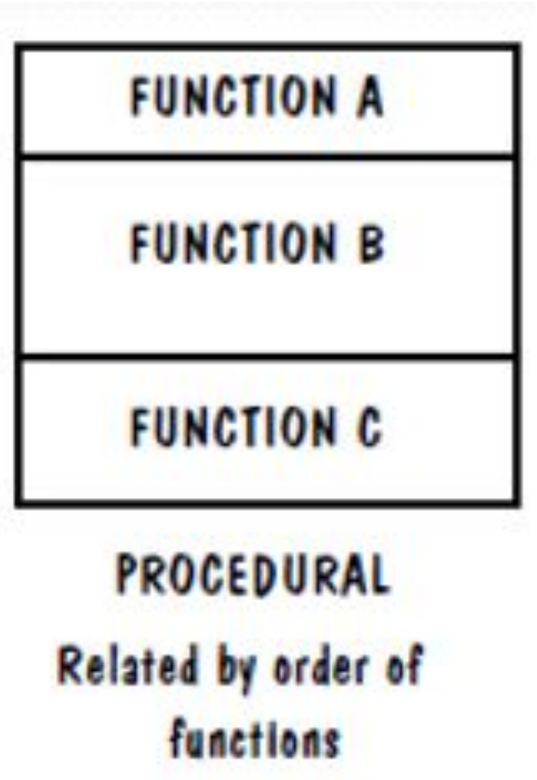


Modularity: Temporal Cohesion

- Elements of component are related by timing
- A module has temporal cohesion when it performs a series of operations related in time

Modularity: Procedural Cohesion

- When functions are grouped together in a module to encapsulate the order of their execution, we say that the module is **procedurally cohesive**.

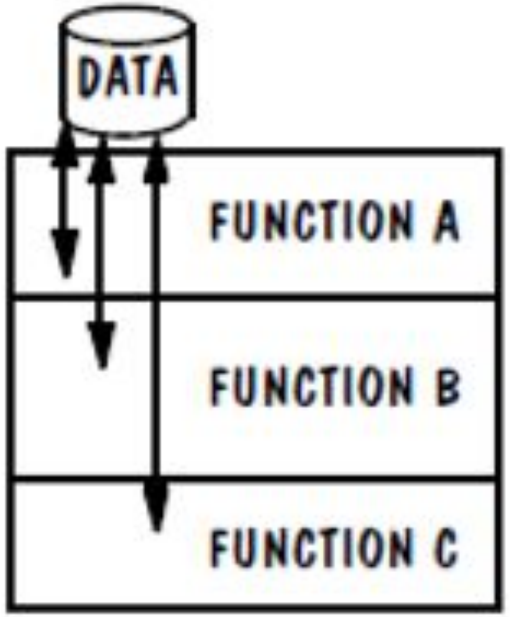


Modularity: Procedural Cohesion

- Think of an **order processing system** for an e-commerce platform. A module might handle the entire procedure of placing an order:
  1. **Validate Payment Information**
  2. **Check Inventory**
  3. **Apply Discount**
  4. **Calculate Shipping**
  5. **Send Order Confirmation**
- In this case, these steps are grouped together in one function because they need to happen in a certain order when processing an order, even though each step could be logically independent from the others.

Modularity: Communicational Cohesion

- Associate certain functions because they operate on the same data set



COMMUNICATIONAL

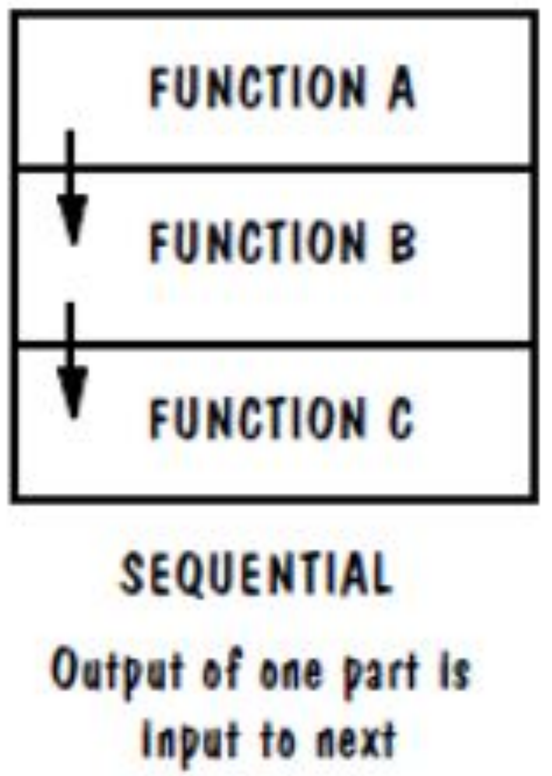
Access same data

Modularity: Communicational Cohesion

- Consider a system that generates a **monthly sales report** for a retail store. In this case, the module might group together the following tasks:
  1. **Fetch Sales Data** from the database for the current month.
  2. **Calculate Total Sales** for the month.
  3. **Determine Top-Selling Products** based on the sales data.
  4. **Generate a Graph** or chart representing sales trends.
  5. **Format and Export the Report** as a PDF.

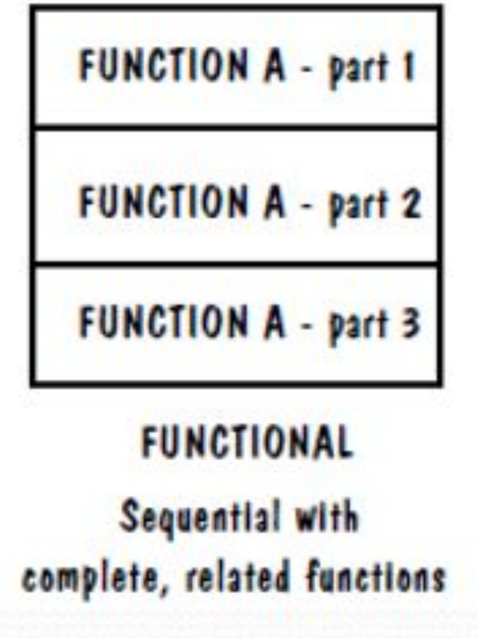
Modularity: Sequential Cohesion

- Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line



Modularity: Functional Cohesion

- Functional cohesion is the strongest and most desirable type of cohesion. It occurs when all elements of a module work together to achieve a **single, well-defined task**.
- Everything within the module is directly related to performing a specific function, and there are no unrelated actions included.



Modularity: Functional Cohesion

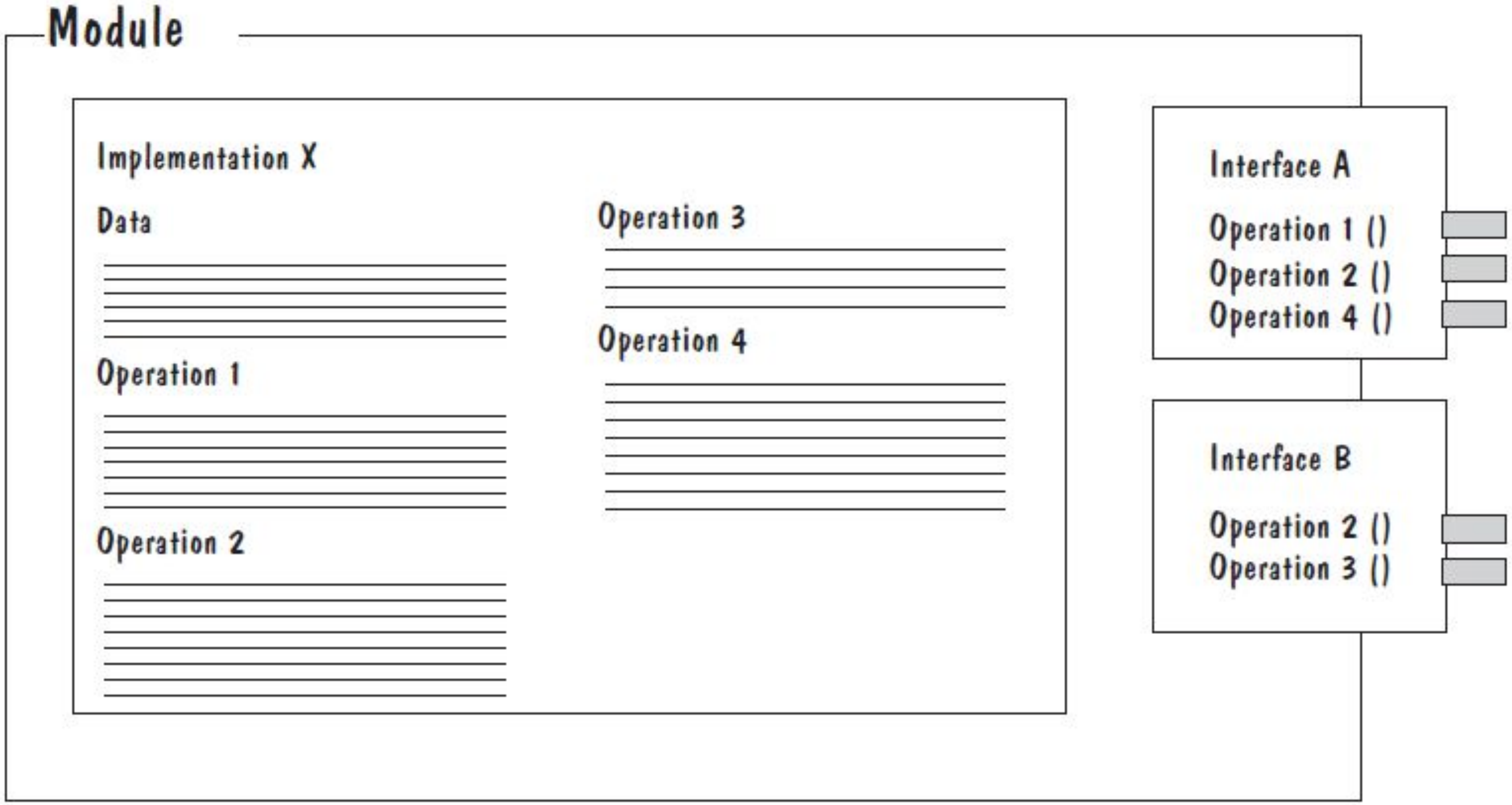
- Consider a **payment validation module** in the e-commerce platform. The sole purpose of this module is to **validate payment details**, and it performs the following tasks:
  1. **Check Card Number Format**
  2. **Verify Expiry Date**
  3. **Authenticate with Payment Gateway**
  4. **Handle Payment Errors**
- All of these tasks are directly related to the specific function of validating payment information.

Interfaces

- An **interface** defines what services the software unit provides to the rest of the system, and how other units can access those services
  - For example, the interface to an object is the collection of the object's public operations and the operations' **signatures**, which specify each operation's name, parameters, and possible return values
- An interface must also define what the unit requires, in terms of services or assumptions, for it to work correctly
- A software unit's interface describes what the unit requires of its environment, as well as what it provides to its environment

Interfaces

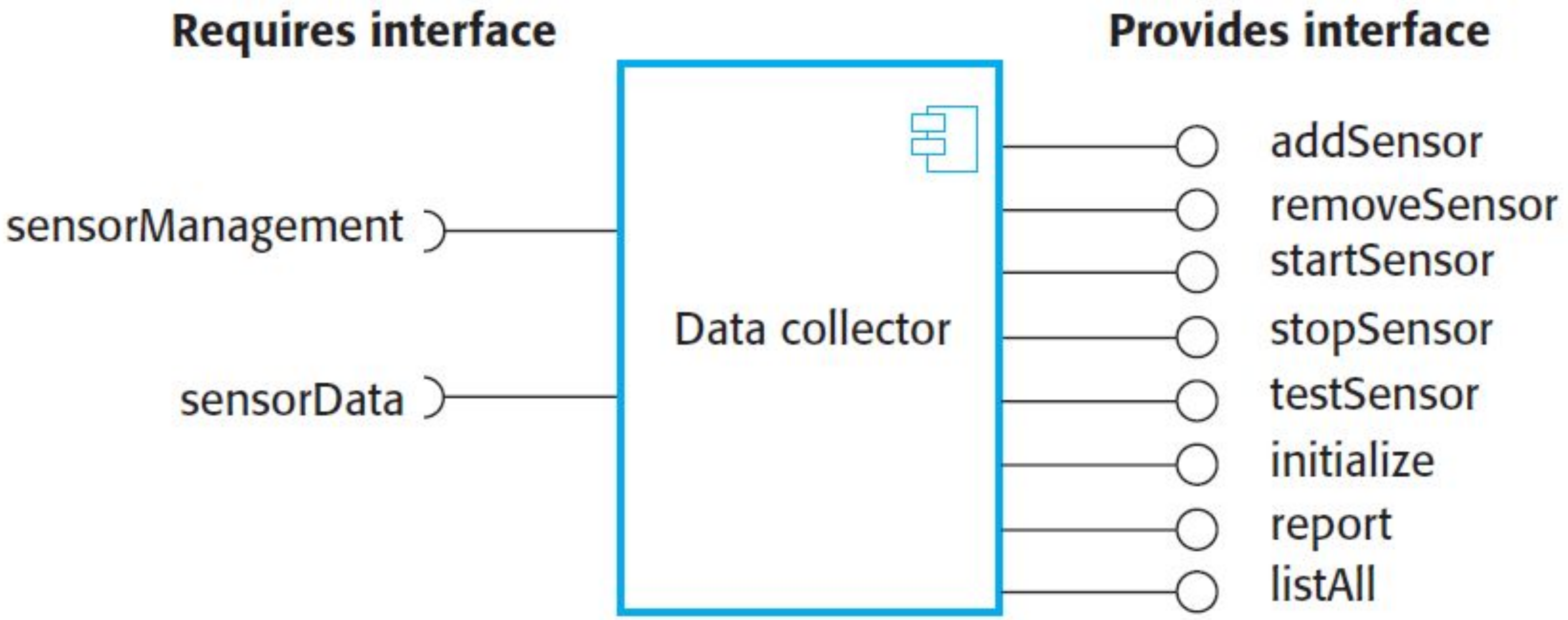
- A software unit may have several interfaces that make different demands on its environment or that offer different levels of service



Interfaces

- The **specification** of a software unit's interface describes the externally visible properties of the software unit
- An interface specification should communicate to other system developers everything that they need to know to use our software unit correctly
  - Purpose
  - Preconditions (assumptions)
    - values of input parameters, states of global resources, or presence of program libraries or other software units
  - Protocols
    - order in which access functions should be invoked, or the pattern in which two components should exchange messages
  - Postconditions (visible effects)
    - return values, raised exceptions, and changes to shared variables
  - Quality attributes
    - performance, reliability

A Component with Interfaces



Information Hiding

- **Information hiding** is distinguished by its guidance for decomposing a system:
  - Each software unit encapsulates a separate design decision that could be changed in the future
  - Then the interfaces and interface specifications are used to describe each software unit in terms of its externally visible properties
- Using this principle, modules may exhibit different kinds of cohesion
  - A module that hides an algorithm may be functionally cohesive
  - A module that hides the sequence in which tasks are performed may be procedurally cohesive.
- A big advantage of information hiding is that the resulting software units are loosely coupled

References

1. Shari PFleeger, Joanne Atlee, Software Engineering: Theory and Practice, 4th Edition

2. Roger S. Pressman, Software Engineering A Practitioner’s Approach, 6th Edition. McGrawHill