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

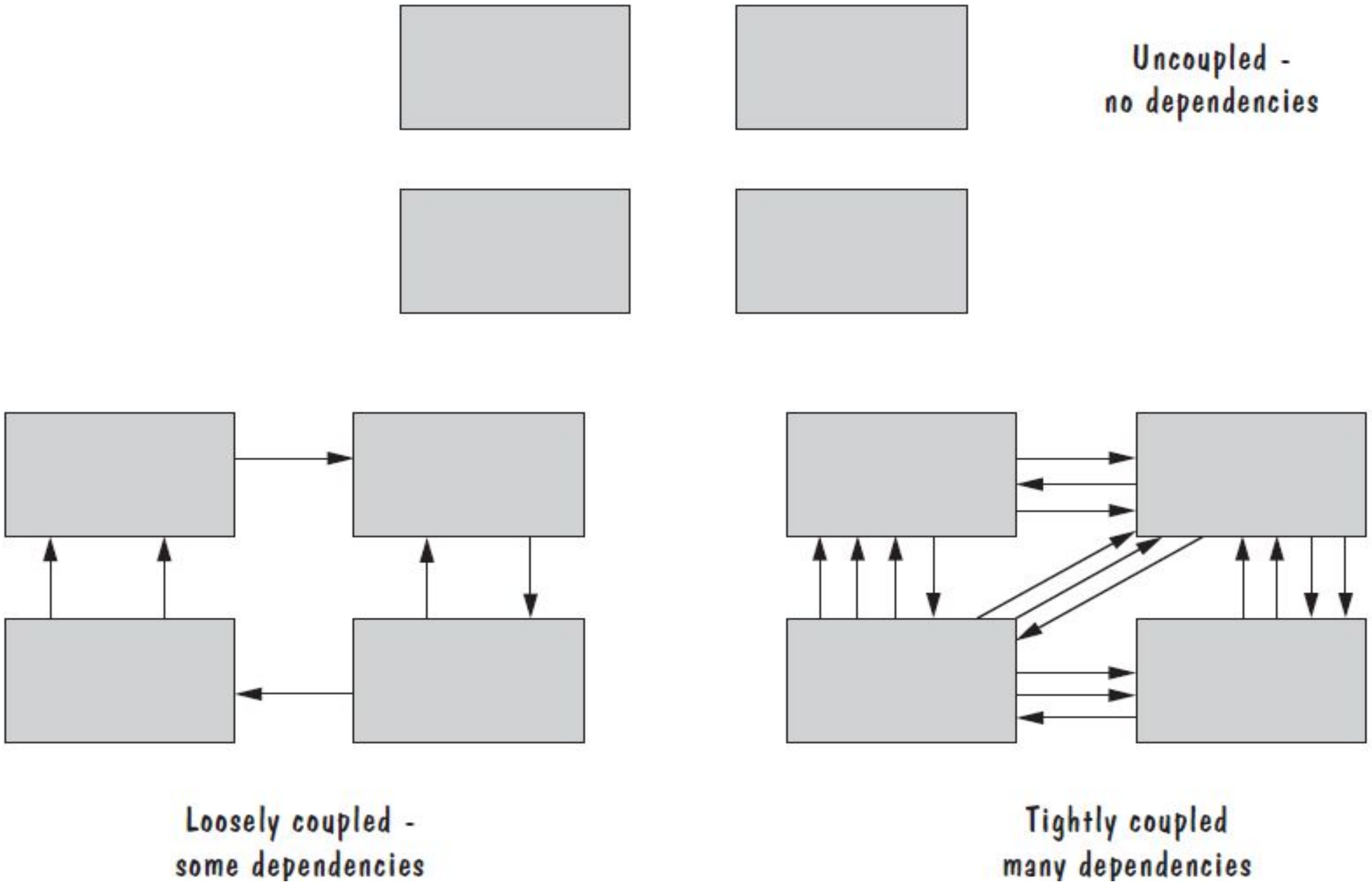
Modularity

- **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



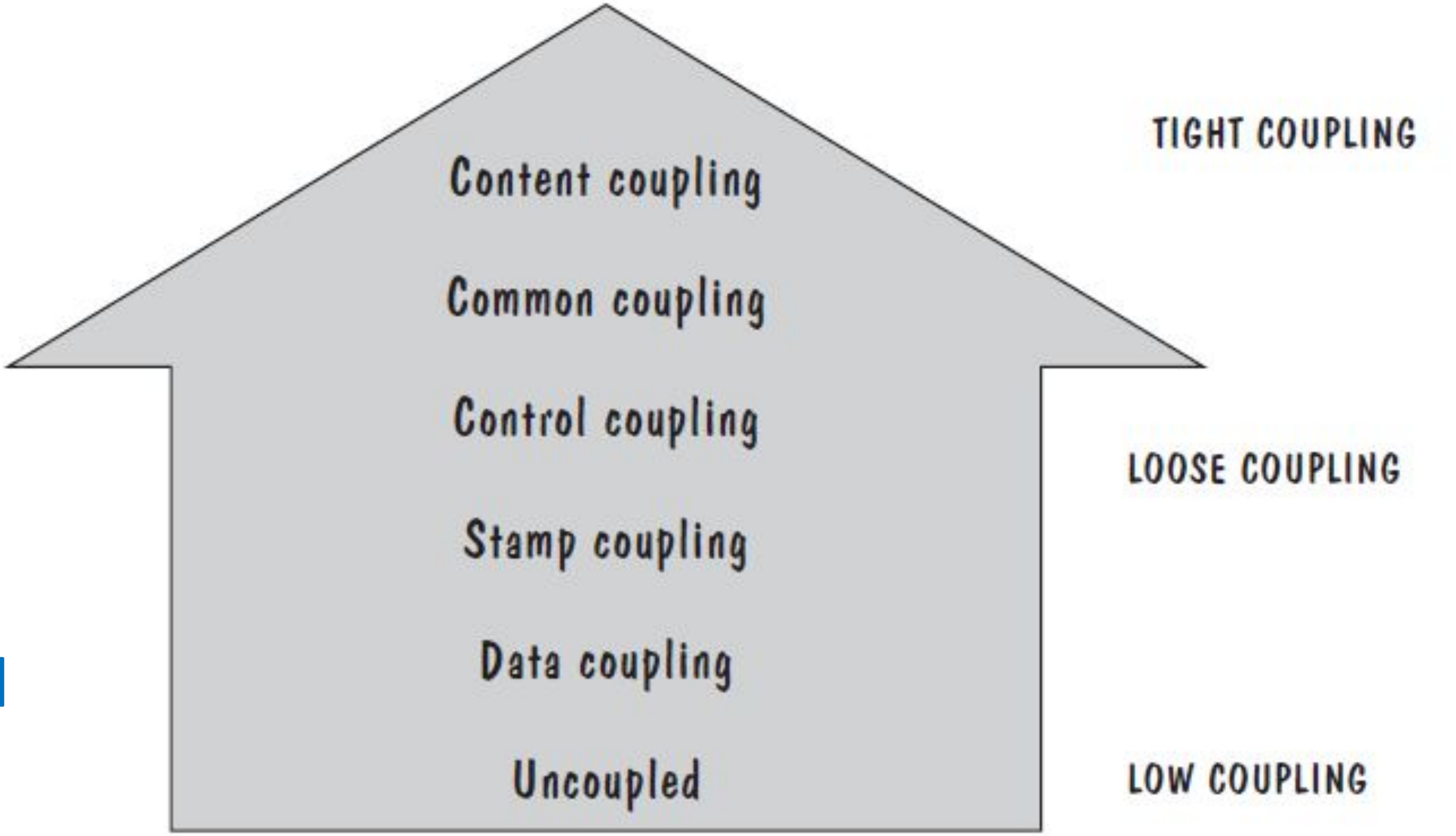
Modularity: Coupling

- 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

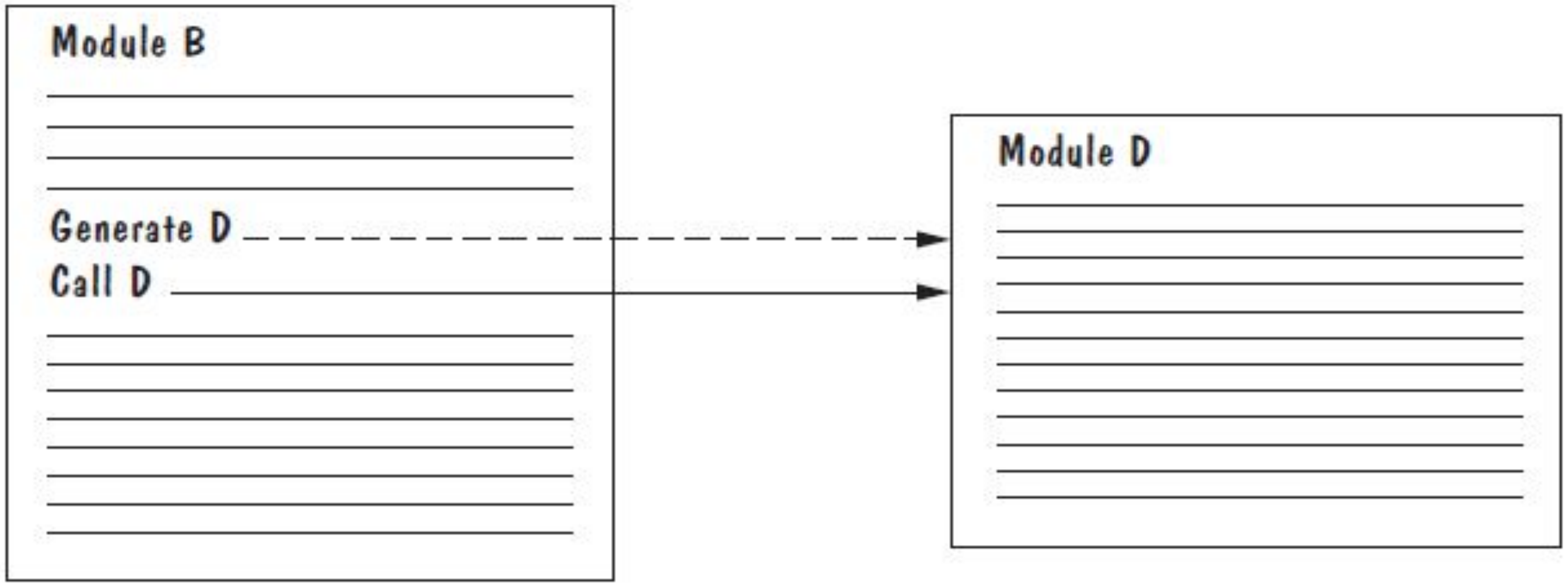
- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling

High coupling is not desired



Modularity: Content Coupling

- One **module modifies another**. The modified module is completely dependent on the modifying one
- One class modifies the content of another class. For example, in C++, [friend classes] can access each other's private members.
- 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: Content Coupling

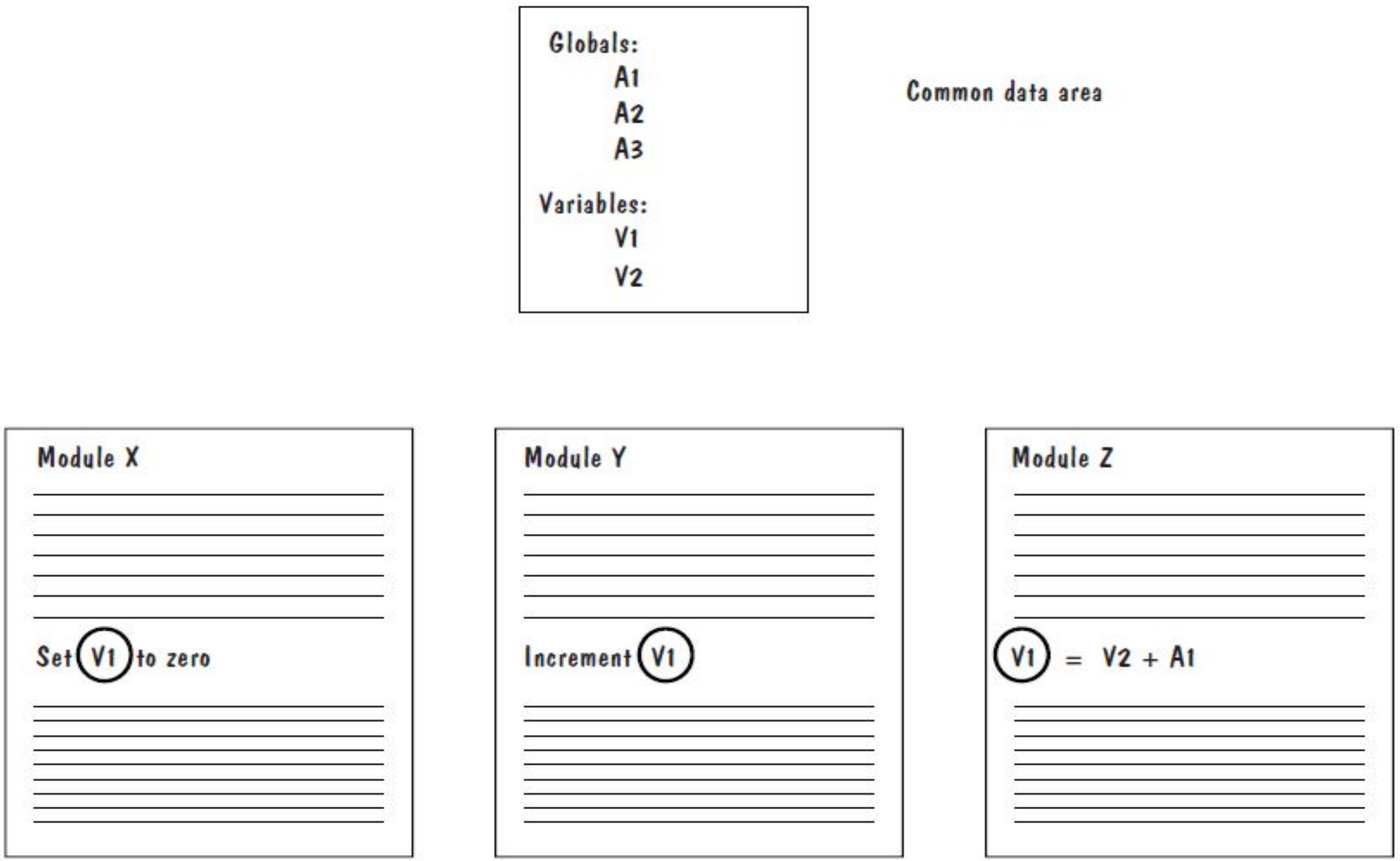
- Consider a scenario where Module A directly accesses the variables of Module B and manipulates them. If Module B's internal structure or variable names change, Module A would need to be updated accordingly.

```cpp
#include <iostream>

class ModuleB {
public:
    int data = 0;
};

class ModuleA {
public:
    void manipulateData(ModuleB& moduleB) {
        moduleB.data++;
    }
};

int main() {
    ModuleB b;
    ModuleA a;
    a.manipulateData(b);
    std::cout << b.data << std::endl;  // Output: 1
    return 0;
}
```

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

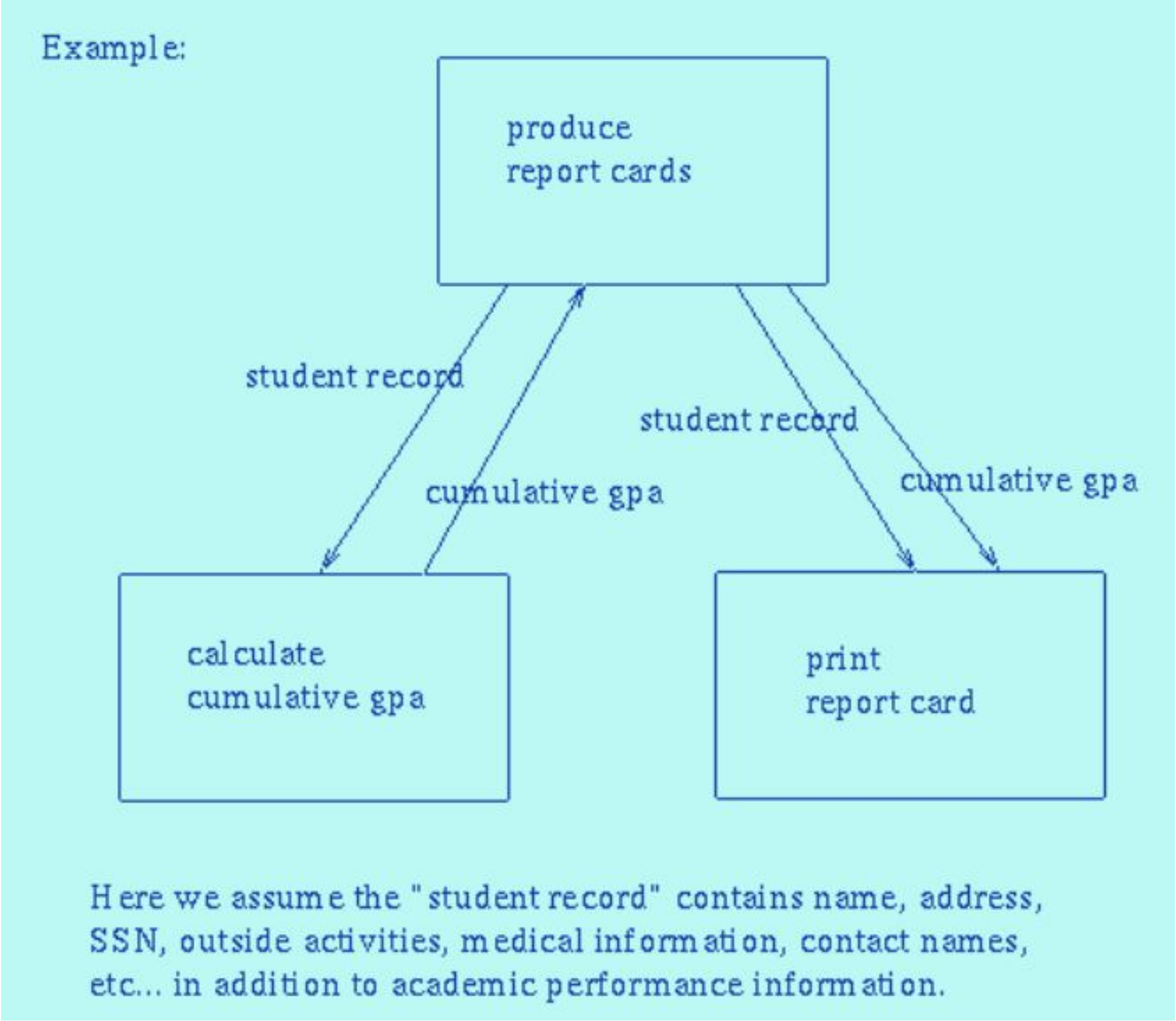
- When one module passes **parameters** or a **return code** to control the behavior of another module
- It is impossible for the controlled module to function without some direction from the controlling module
- Limit each module to be responsible for only one function or one activity.
- Restriction 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

```cpp
#include <iostream>

void moduleA(int flag) {
    if (flag == 1) {
        std::cout << "Executing operation 1" << std::endl;
    } else if (flag == 2) {
        std::cout << "Executing operation 2" << std::endl;
    }
}

int main() {
    moduleA(1);  // Output: Executing operation 1
    moduleA(2);  // Output: Executing operation 2
    return 0;
}
```

Modularity: Stamp Coupling

- Stamp coupling occurs when modules **exchange complex data structures**, typically large parameters or data objects, instead of passing only the necessary information.
- Stamp coupling represents a more complex interface between modules, because the modules have to agree on the data's format and organization



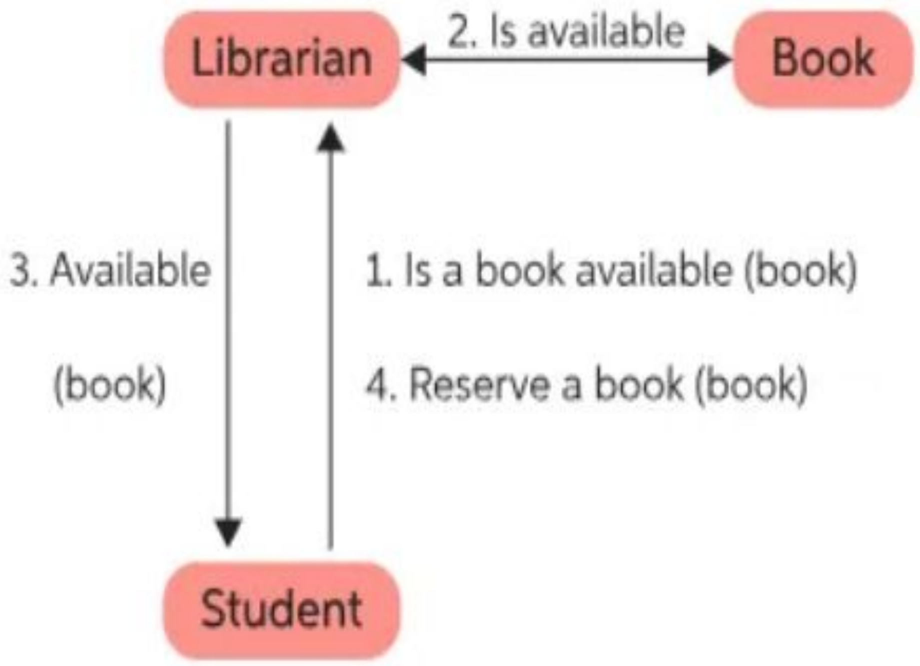
Modularity: Stamp Coupling

- 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: Data Coupling

```cpp
#include <iostream>

int moduleA(int x, int y) {
    return x + y;
}

int moduleB(int x, int y) {
    return x * y;
}

int main() {
    int resultA = moduleA(3, 4);  // Output: 7
    int resultB = moduleB(3, 4);  // Output: 12
    return 0;
}
```

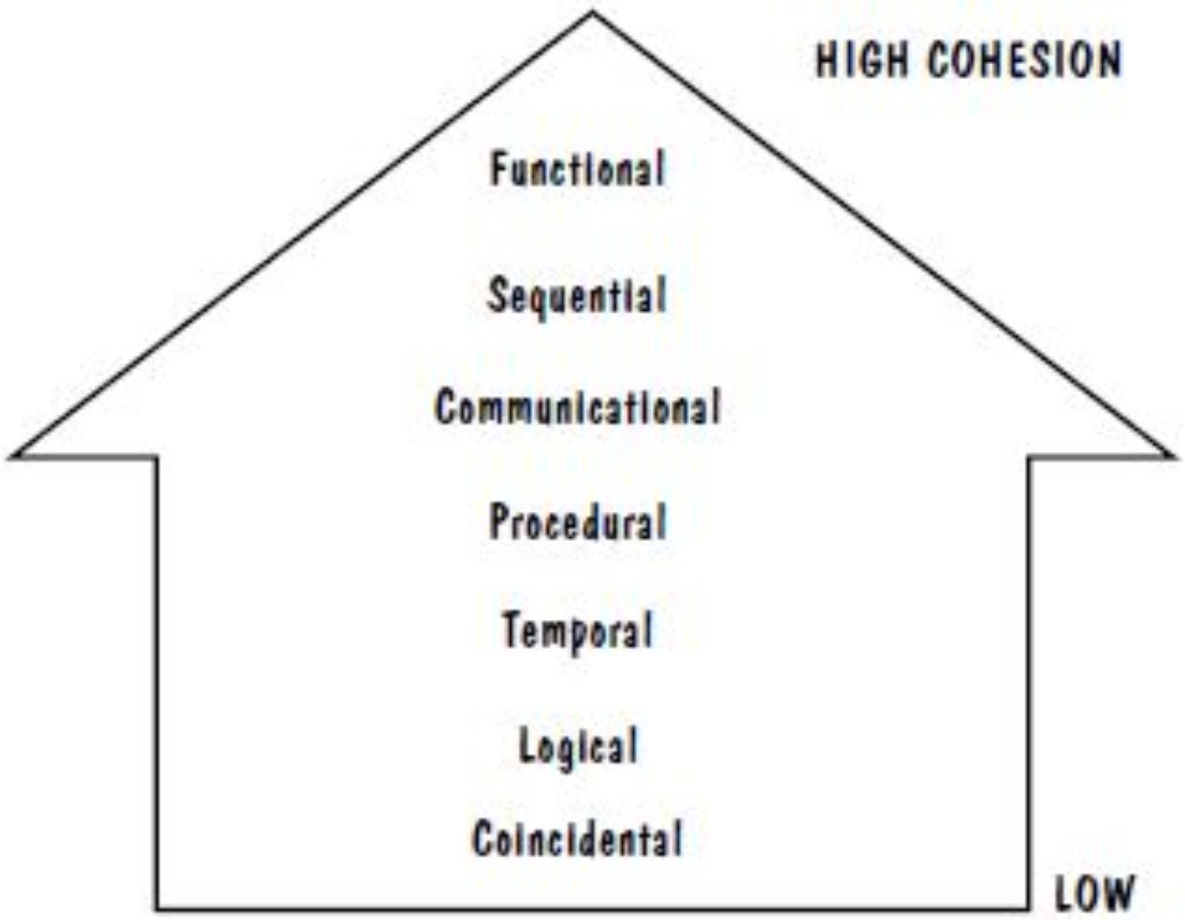
Modularity: Cohesion

- 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

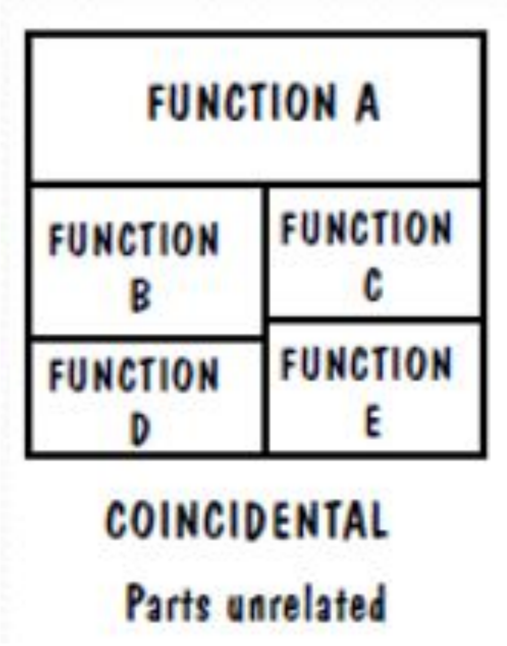
- Coincidental cohesion
- Logical cohesion
- Temporal cohesion
- Procedural cohesion
- Communicational cohesion
- Functional cohesion
- Sequential cohesion

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
- This is often considered the weakest form of cohesion because it doesn't reflect a logical design principle.
- Unrelated functions, processes, or data are combined in the same module for reasons of convenience

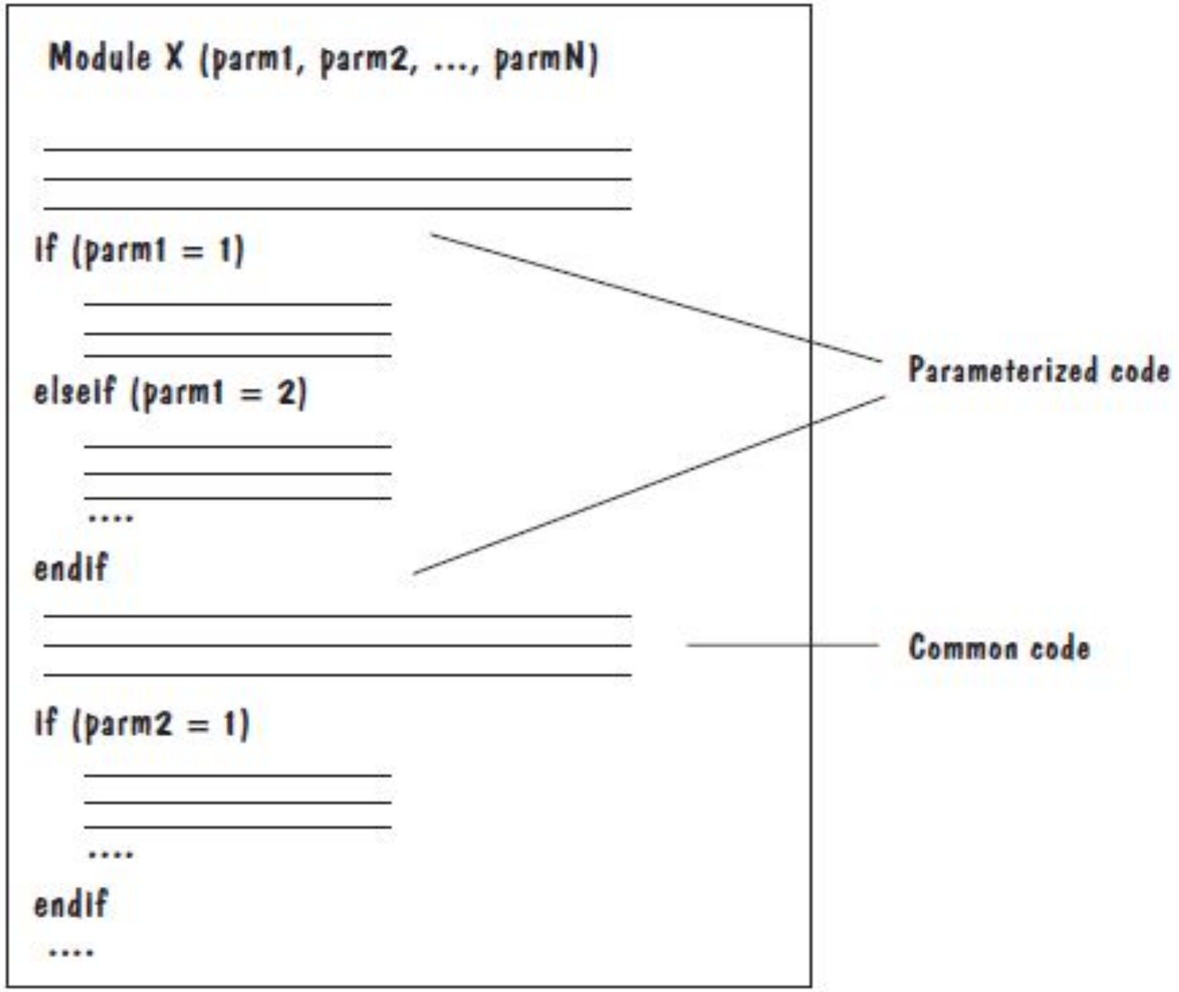


Modularity: Coincidental Cohesion

```cpp
class CoincidentalExample {
public:
    void greetUser() {
        std::cout << "Hello, User! ";
    }

    void calculateSquare(int num) {
        std::cout << "Square of " << num << " is " << (num * num) << " ";
    }

    void displayMenu() {
        std::cout << "1. Option 1\n2. Option 2\n3. Option 3";
    }
};

int main() {
    CoincidentalExample example;
    example.greetUser();
    example.calculateSquare(5);
    example.displayMenu();
    return 0;
}
```

Modularity: Logical Cohesion

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



Modularity: Logical Cohesion

```
class LogicalCohesionExample {
public:
    void checkNumber(int num) {
        if (num % 2 == 0) {
            std::cout << num << " is even." << std::endl;
        } else {
            std::cout << num << " is odd." << std::endl;
        }
        printMessage();
    }

private:
    void printMessage() {
        std::cout << "Process ended." << std::endl;
    }
};

int main() {
    LogicalCohesionExample example;
    example.checkNumber(5);
    example.checkNumber(8);
    return 0;
}
```

Modularity: Logical Cohesion

```cpp
class LogicalCohesion {
public:
    void calculateSum(int a, int b) {
        int sum = a + b;
        std::cout << "Sum: " << sum << std::endl;
    }

    void calculateDifference(int a, int b) {
        int diff = a - b;
        std::cout << "Difference: " << diff << std::endl;
    }
};

int main() {
    LogicalCohesion module;
    module.calculateSum(10, 5);
    module.calculateDifference(10, 5);
    return 0;
}
```

Modularity: Temporal Cohesion

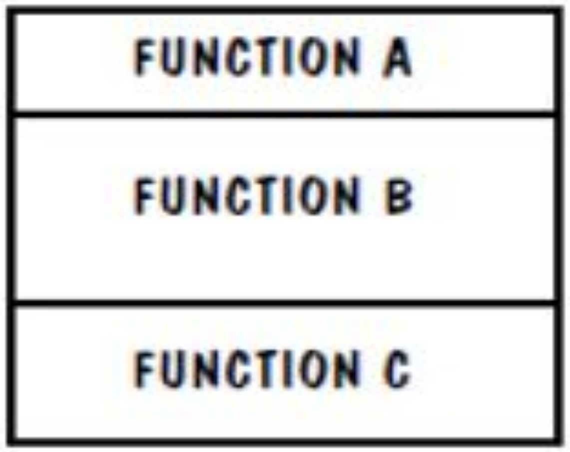
- Elements of component are related by **timing.**
- Temporal cohesion occurs when elements within a module are executed sequentially due to their temporal relationship.
- These elements may not be logically related but are performed together because they happen to be executed in sequence.

Modularity: Temporal Cohesion

```cpp
class OrderProcessing {
public:
    void processOrder(int orderId) {
        retrieveOrderDetails(orderId);
        calculateTotal(orderId);
        sendConfirmationEmail(orderId);
    }

private:
    void retrieveOrderDetails(int orderId) {
        std::cout << "Retrieving order details for Order ID: " << orderId << std::endl;
    }

    void calculateTotal(int orderId) {
        std::cout << "Calculating total for Order ID: " << orderId << std::endl;
    }

    void sendConfirmationEmail(int orderId) {
        std::cout << "Sending confirmation email for Order ID: " << orderId << std::endl;
    }
};

int main() {
    OrderProcessing orderProcessor;
    orderProcessor.processOrder(123456);
}
```

Modularity: Procedural Cohesion

- Procedural cohesion occurs when elements within a module are grouped together because they are required to perform a specific **task** or **procedure**.
- The elements work together to accomplish a common procedural goal.



PROCEDURAL

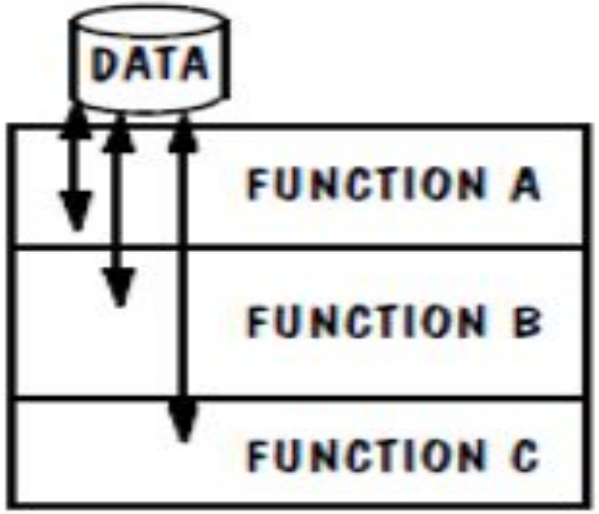
Related by order of functions

Modularity: Procedural Cohesion

```
class Recipe {
public:
    void prepareDough() {
        mixIngredients();
        kneadDough();
        letDoughRise();
    }

private:
    void mixIngredients() {
        std::cout << "Mixing ingredients for dough..." << std::endl;
    }

    void kneadDough() {
        std::cout << "Kneading the dough..." << std::endl;
    }

    void letDoughRise() {
        std::cout << "Letting the dough rise..." << std::endl;
    }
};

int main() {
    Recipe breadRecipe;
    breadRecipe.prepareDough();
    return 0;
}
```

Modularity: Communicational Cohesion

- Communicational cohesion occurs when elements within a module are grouped together because they **operate on the same data or input/output**.
- The elements communicate with each other through shared data or parameters.



COMMUNICATIONAL

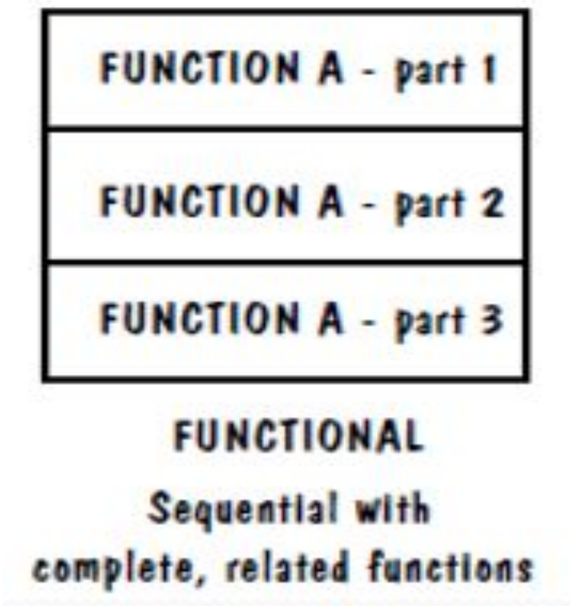
Access same data

Modularity: Communicational Cohesion

```cpp
class CommunicationalCohesion {
public:
    void processData(int data) {
        processDataA(data);
        processDataB(data);
    }

private:
    void processDataA(int data) {
        std::cout << "Processing data A: " << data << std::endl;
    }

    void processDataB(int data) {
        std::cout << "Processing data B: " << data << std::endl;
    }
};

int main() {
    CommunicationalCohesion module;
    module.processData(123);
    return 0;
}
```

Modularity: Functional Cohesion

- All elements essential to a single function are contained in one module, and all of that module's elements are essential to the performance of that function
- A functionally cohesive module performs only the function for which it is designed, and nothing else

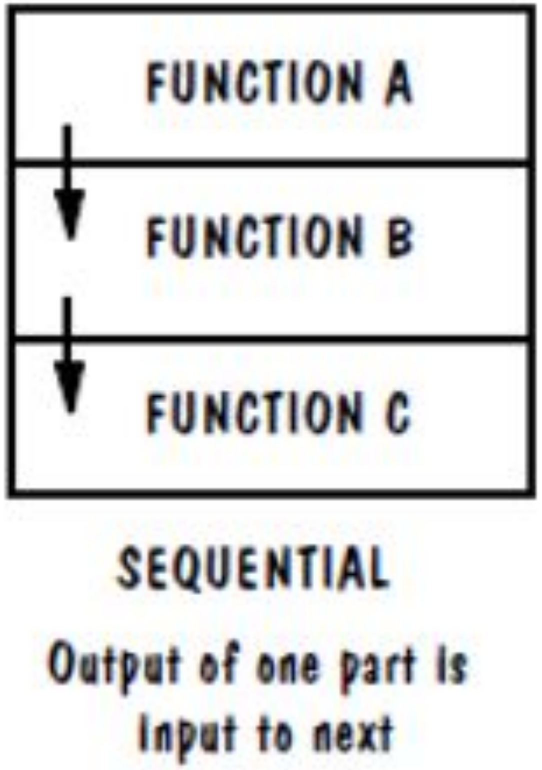


Modularity: Functional Cohesion

```cpp
class FileManager {
public:
    void openFile(const std::string& filename) {
        std::cout << "Opening file: " << filename << std::endl;
        // Code to open the file
    }

    void readFile(const std::string& filename) {
        std::cout << "Reading file: " << filename << std::endl;
        // Code to read data from the file
    }

    void writeFile(const std::string& filename, const std::string& data) {
        std::cout << "Writing data to file: " << filename << std::endl;
        // Code to write data to the file
    }

    void closeFile(const std::string& filename) {
        std::cout << "Closing file: " << filename << std::endl;
        // Code to close the file
    }
};

int main() {
    FileManager fileManager;
    fileManager.openFile("example.txt");
    fileManager.readFile("example.txt");
    fileManager.writeFile("example.txt", "Hello, World!");
    fileManager.closeFile("example.txt");
    return 0;
}
```

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: Sequential Cohesion

```cpp
class DataProcessor {
public:
    void processData(const std::string& input) {
        std::string cleanedData = cleanData(input);
        std::string processedData = process(cleanedData);
        displayResult(processedData);
    }

private:
    std::string cleanData(const std::string& input) {
        std::cout << "Cleaning data..." << std::endl;
        // Code to clean the input data
        return "Cleaned " + input;
    }

    std::string process(const std::string& input) {
        std::cout << "Processing data..." << std::endl;
        // Code to process the cleaned data
        return "Processed " + input;
    }

    void displayResult(const std::string& input) {
        std::cout << "Displaying result: " << input << std::endl;
    }
};
```

Modularity: Informational Cohesion

- Elements within a module are grouped together because they are **related by providing information** about a specific **topic**, **entity**, or **concept**.
- In other words, the elements within the module are responsible for **gathering**, **processing**, and **presenting** information about a common subject.
- The design goal is the same: to **put data, actions, or objects together** only when they have one **common, sensible purpose**.
- For example, we say that an OO design component is cohesive if all of the attributes, methods, and action are strongly interdependent and essential to the object

Modularity: Informational Cohesion

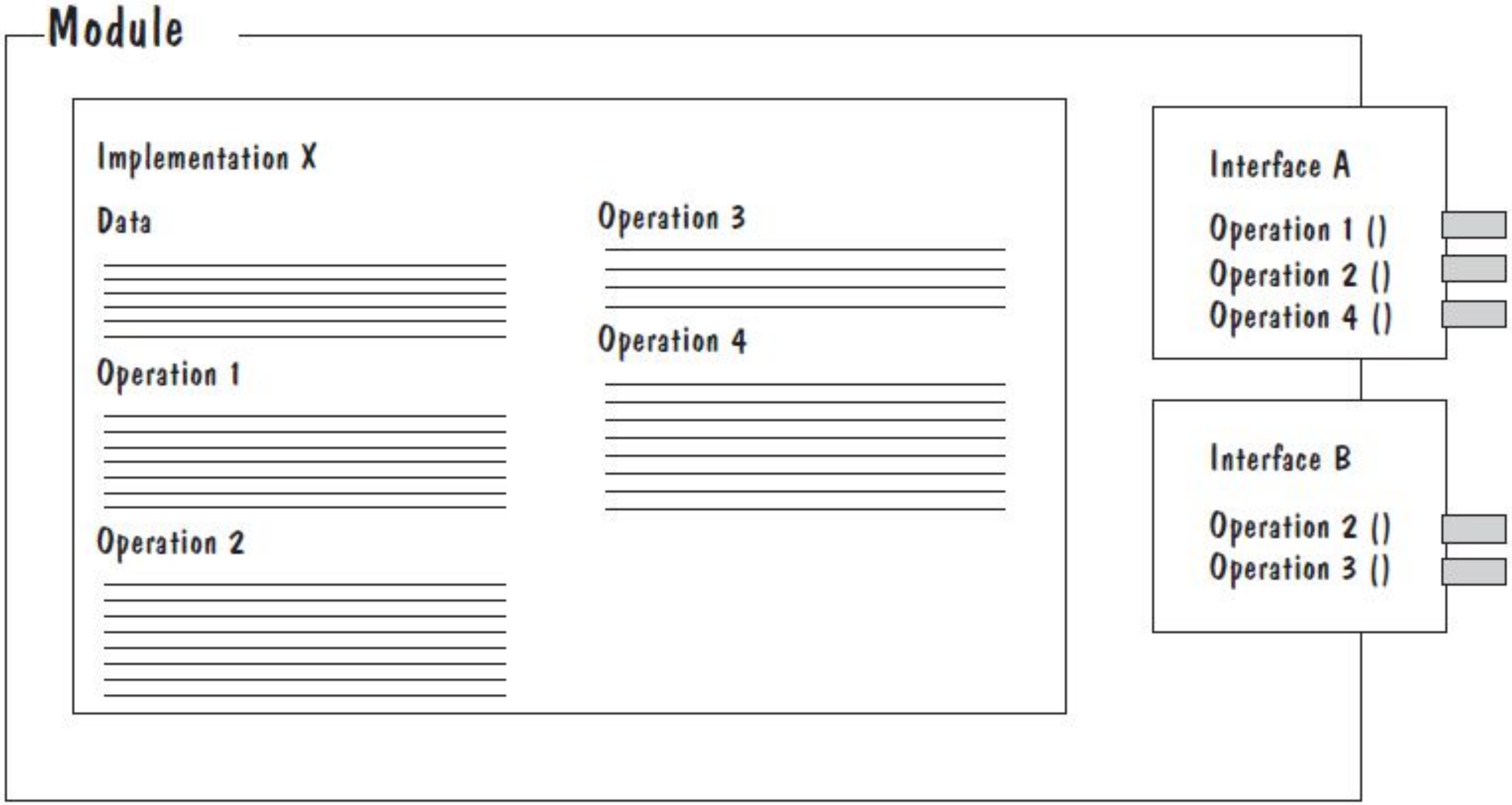
```
class WeatherApp {
public:
    void fetchData() {
        std::cout << "Fetching weather data from API... ";
        // Code to fetch weather data from an external API
    }

    void processWeatherData() {
        std::cout << "Processing weather data to extract relevant information... ";
        // Code to analyze and process the fetched weather data
    }

    void displayWeatherForecast() {
        std::cout << "Displaying weather forecast to the user... ";
        // Code to present the processed weather forecast to the user
    }
};

int main() {
    WeatherApp weatherApp;
    weatherApp.fetchData();
    weatherApp.processWeatherData();
    weatherApp.displayWeatherForecast();
    return 0;
}
```

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

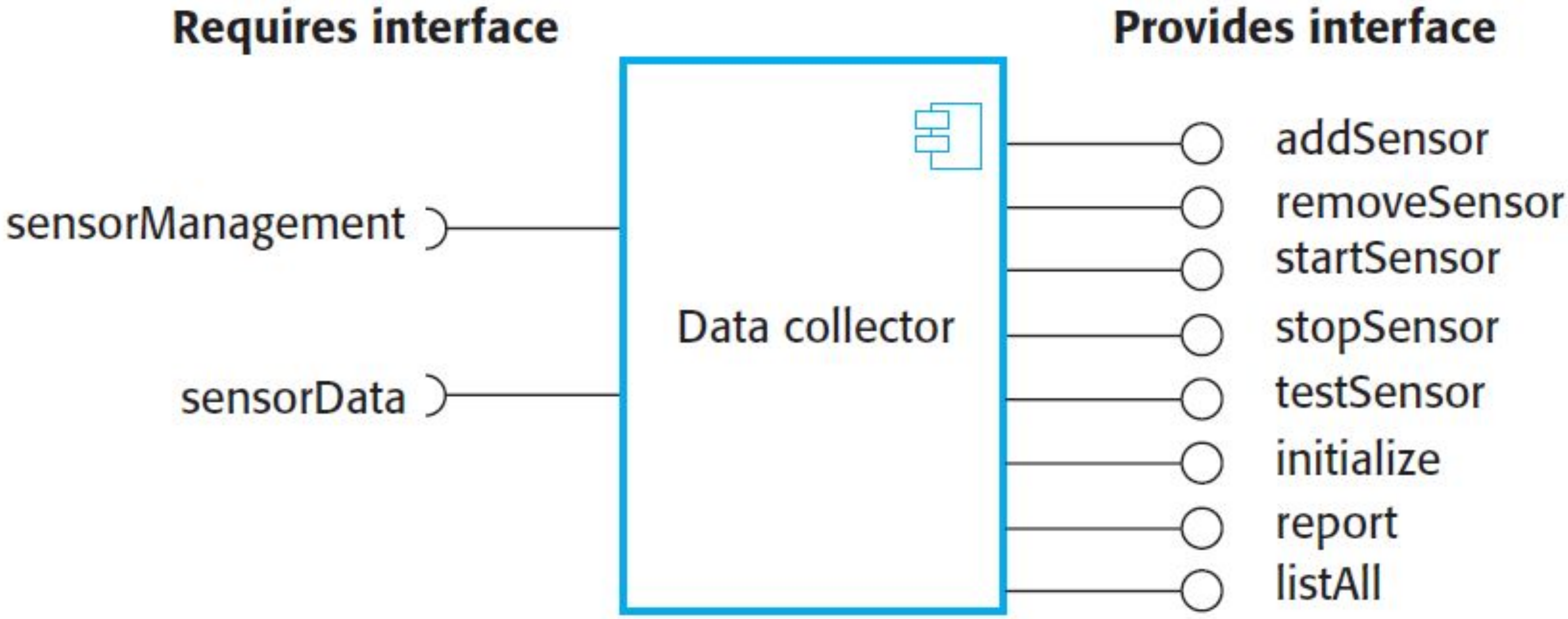
- A software unit may have several interfaces that make different demands on its environment or that offer different levels of service



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

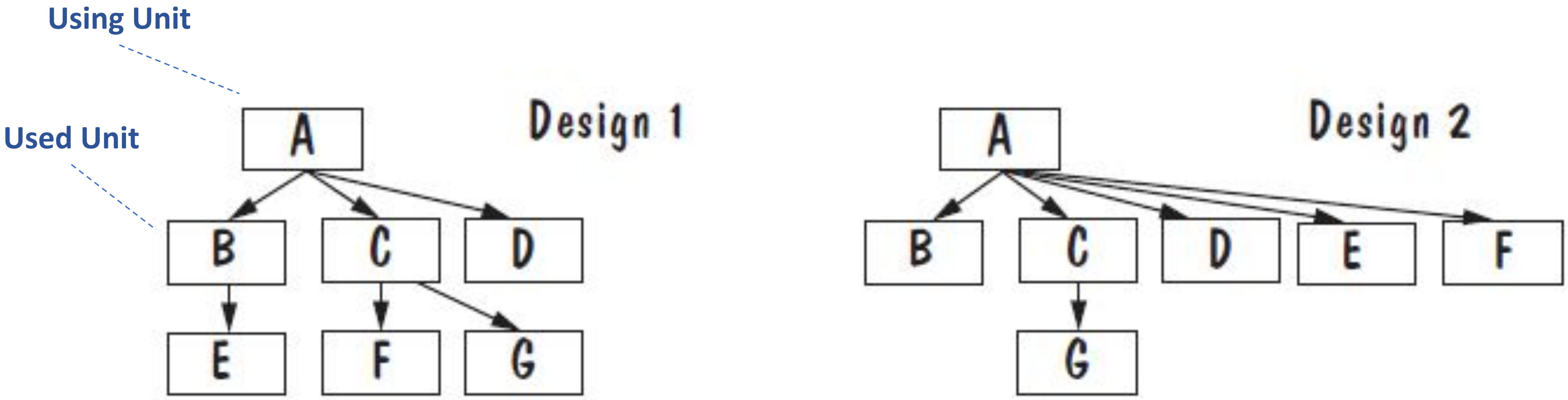
- **Information hiding** is distinguished by its guidance for decomposing a system:
  - Each software unit encapsulates a separate design decision that could be changed in the future
  - Then the interfaces and interface specifications are used to describe each software unit in terms of its externally visible properties
- Using this principle, modules may exhibit different kinds of cohesion
  - A module that hides a data representation may be informationally cohesive
  - 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

Incremental Development

- Given a design consisting of software units and their interfaces, we can use the information about the units' dependencies to devise an incremental schedule of development
- Start by mapping out the units' **uses relation**
    - relates each software unit to the other software units on which it depends
- **Uses graphs** can help to identify progressively larger subsets of our system that we can implement and test incrementally

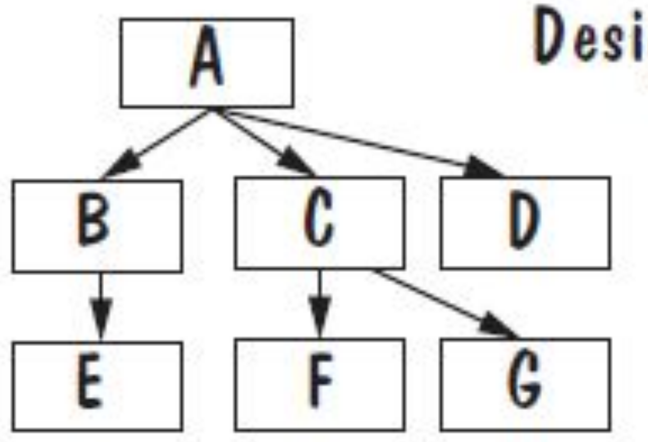
Incremental Development

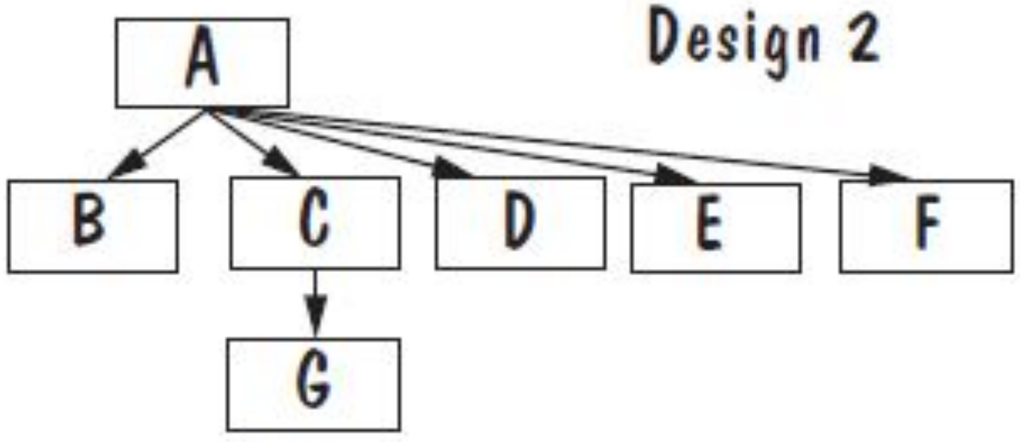
- Nodes represent **software units**, and directed edges run from the **using units**, such as A, to the **used units**, such as B.
- Uses graphs for two designs
  - **Fan-out** refers to the number of units used by particular software unit
  - **Fan-in** refers to the number of units that use a particular software unit



Incremental Development

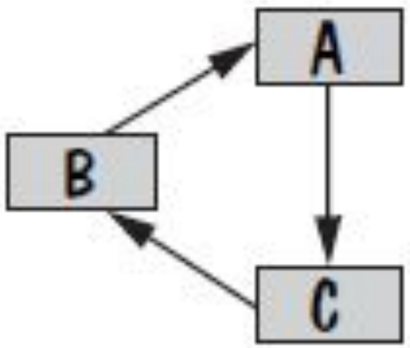
- A uses graph can also help us to identify areas of the design that could be improved
- Unit A has a fan-out of three in Design 1 but a fan-out of five in Design 2.
- Goal in designing a system is to create software units with **high fan-in** and **low fan-out**.
- High fan-out usually indicates that the software unit is doing too much and probably ought to be decomposed into smaller, simpler units





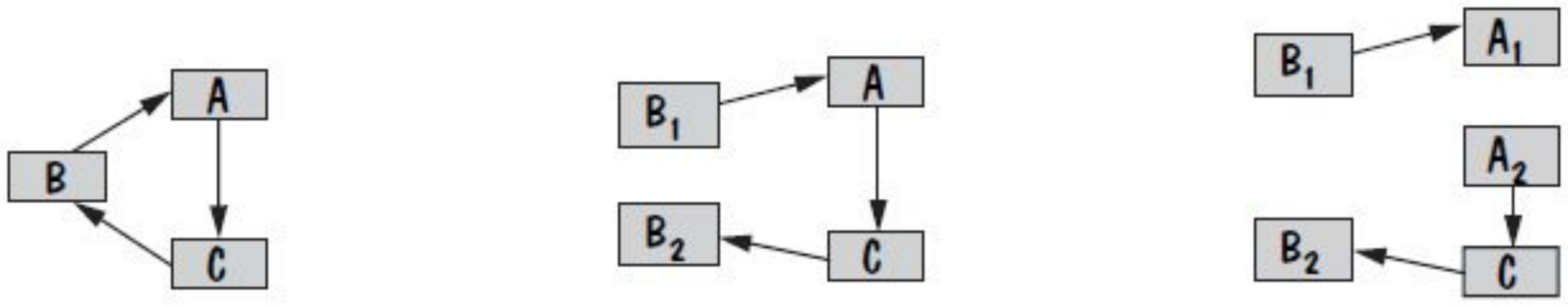
Incremental Development

- The cycle in uses graph identifies a collection of units that are **mutually dependent** on each other.
- Cycles are not necessarily bad. If the problem that the units are solving is naturally **recursive**, then it makes sense for the design to include modules that are mutually recursive.
- Large cycles limit the design's ability to support incremental development: none of the units in the cycle can be developed (i.e., implemented, tested, debugged) until all the cycle's units are developed



Incremental Development

- We can try to break a cycle in the uses graph using a technique called **sandwiching**
  - One of the cycle's units is **decomposed** into two units, such that one of the new units has **no dependencies**
  - Sandwiching can be applied more than once, to break either mutual dependencies in tightly coupled units or long dependency chains



(a) (b) (c)

Abstraction

- An **abstraction** is a model or representation that **omits some details** so that it can **focus on other details**
- The definition is vague about which details are left out of a model, because different abstractions, built for different purposes, omit different kinds of details

Using Abstraction (Sidebar)

- Suppose that one of the system's function is to sort the elements of a list L. The initial description of the design is:
  1. Sort L in ascending order
  2. The next level of abstraction may be a particular algorithm:

```
DO WHILE I is between 1 and (length of L)-1:
   Set LOW to index of smallest value in L(I),..., L(length of L)
Interchange L(I) and L(LOW)
ENDDO
```

- The algorithm provides a great deal of additional information; however, it can be made even more detailed

Using Abstraction (Sidebar)

3. The third and final algorithm describes exactly how the sorting operation will work:

```
DO WHILE I is between 1 and (length of L)-1
  Set LOW to current value of I
  DO WHILE J is between I+1 and (length of L)
      IF L(LOW) is greater than L(J)
          THEN set LOW to current value of J
      ENDIF
  ENDDO
  Set TEMP to L(LOW)
  Set L(LOW) to L(I)
  Set L(I) to TEMP
ENDDO
```

Using Abstraction (Sidebar)

- Each level of abstraction serves a purpose.
- If we care only about what L looks like before and after sorting, then the first abstraction provides all the information we need.
- If we are concerned about the speed of the algorithm, then the second level of abstraction provides sufficient detail to analyze the algorithm’s complexity.
- However, if we are writing code for the sorting operation, the third level of abstraction tells us exactly what is to happen; little additional information is needed.

Generality

- Make a software unit as universally applicable as possible, to increase the chance that it will be useful in some future system
- We make a unit more general by increasing the number of contexts in which it can be used.

Generality

- There are several ways of doing this:

 Parameterizing context-specific information: We create a more general version of our software by making into parameters the data on which it operates.

 Removing preconditions: We remove preconditions by making our software work under conditions that we previously assumed would never happen.

 Simplifying postconditions: We reduce postconditions by splitting a complex software unit into multiple units that divide responsibility for providing the postconditions. The units can be used together to solve the original problem, or used separately when only a subset of the postconditions is needed.

Generality

- The following four procedure interfaces are listed in order of increasing generality:

```
PROCEDURE SUM: INTEGER;
POSTCONDITION: returns sum of 3 global variables

PROCEDURE SUM (a, b, c: INTEGER): INTEGER;
POSTCONDITION: returns sum of parameters

PROCEDURE SUM (a[]: INTEGER; len: INTEGER): INTEGER
PRECONDITION: 0 <= len <= size of array a
POSTCONDITION: returns sum of elements 1..len in array a

PROCEDURE SUM (a[]: INTEGER): INTEGER
POSTCONDITION: returns sum of elements in array a
```

References

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

2. Roger S. Pressman, Software Engineering A Practitioner's Approach, 9th Edition. McGrawHill