DESIGNING the MODULES

Mehwish Mumtaz

Design Methodology

- We have an abstract description of a solution to our customer's problem, a software architectural design, a plan for decomposing the design into software units and allocating the system's functional requirements to them
- No distinct boundary between the end of the architecture-design phase and the start of the module-design phase
- No comparable design recipes for progressing from a software unit's specification to its modular design

Design Methodology

- Design decisions are periodically revisited and revised
- Refactoring
  - to simplify complicated solutions or to optimize the design

Design Principles

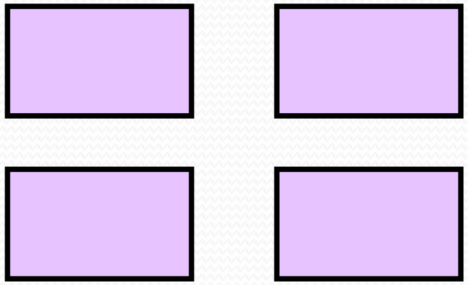
- **Design principles** are guidelines for decomposing a system’s required functionality and behavior into modules
- The principles identify the criteria
  - for decomposing a system
  - deciding what information to provide (and what to conceal) in the resulting modules
- Dominant principles (general):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

Modularity

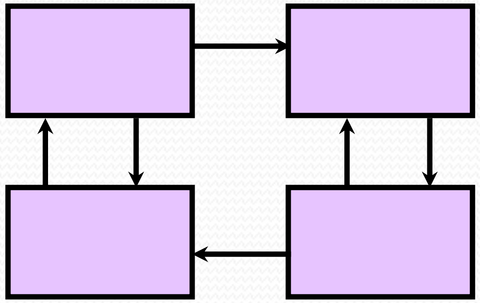
- **Modularity** is the principle of keeping the unrelated aspects of a system separate from each other,
  - each aspect can be studied in isolation (also called separation of concerns)
- If the principle is applied well, each resulting module will have a single purpose and will be relatively independent 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

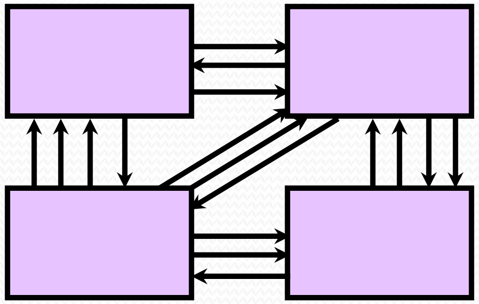
- Two modules are **tightly coupled** when they depend a great deal on each other
- **Loosely coupled** modules have some dependence, but their interconnections are weak
- **Uncoupled** modules have no interconnections at all; they are completely unrelated


–Uncoupled -

–no dependencies


–Loosely coupled -

–some dependencies


–Tightly coupled -

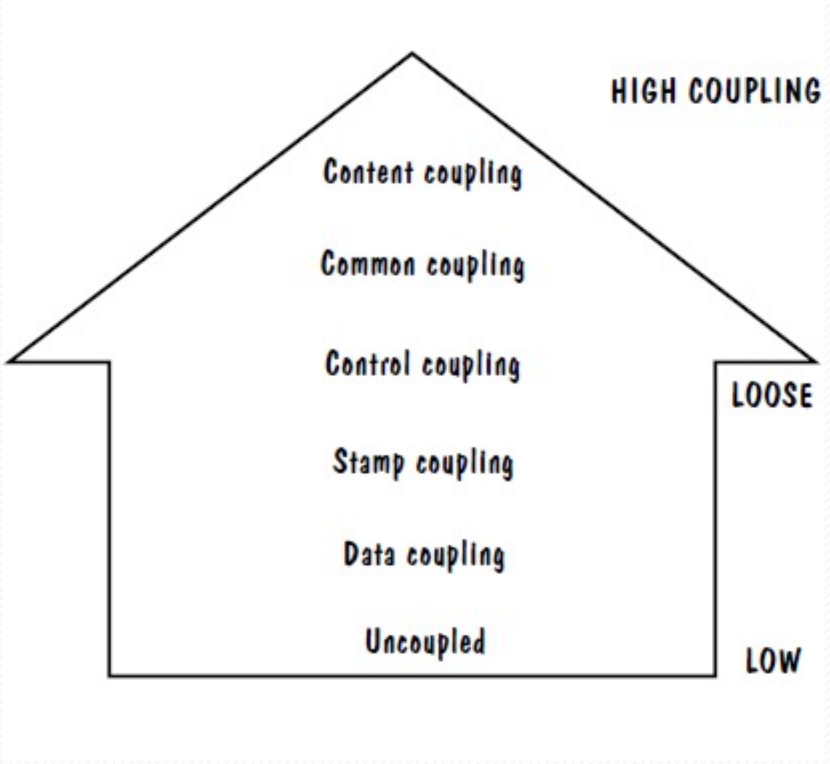
–many dependencies

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

Modularity: Coupling: Types of Coupling

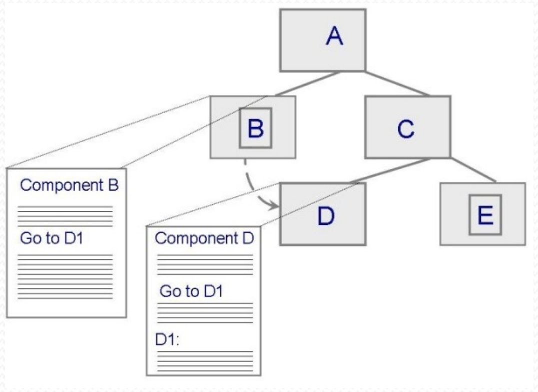
- Content coupling
- Common coupling
- Control -coupling
- Stamp coupling
- Data coupling

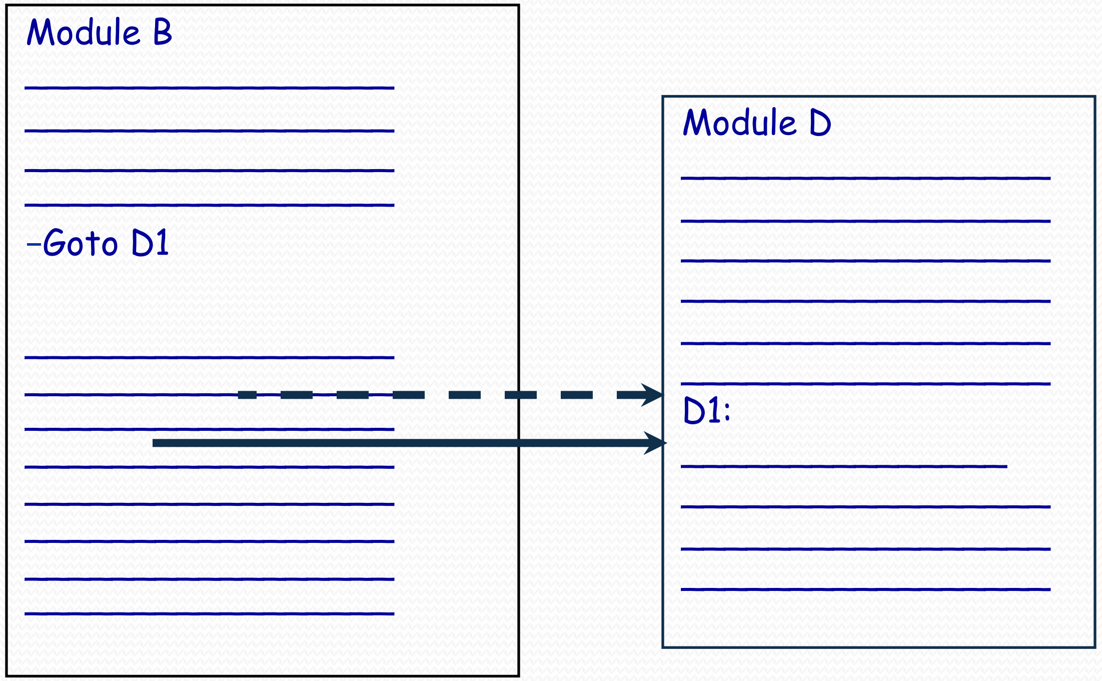


High coupling is not desired

Modularity: Coupling: Content Coupling

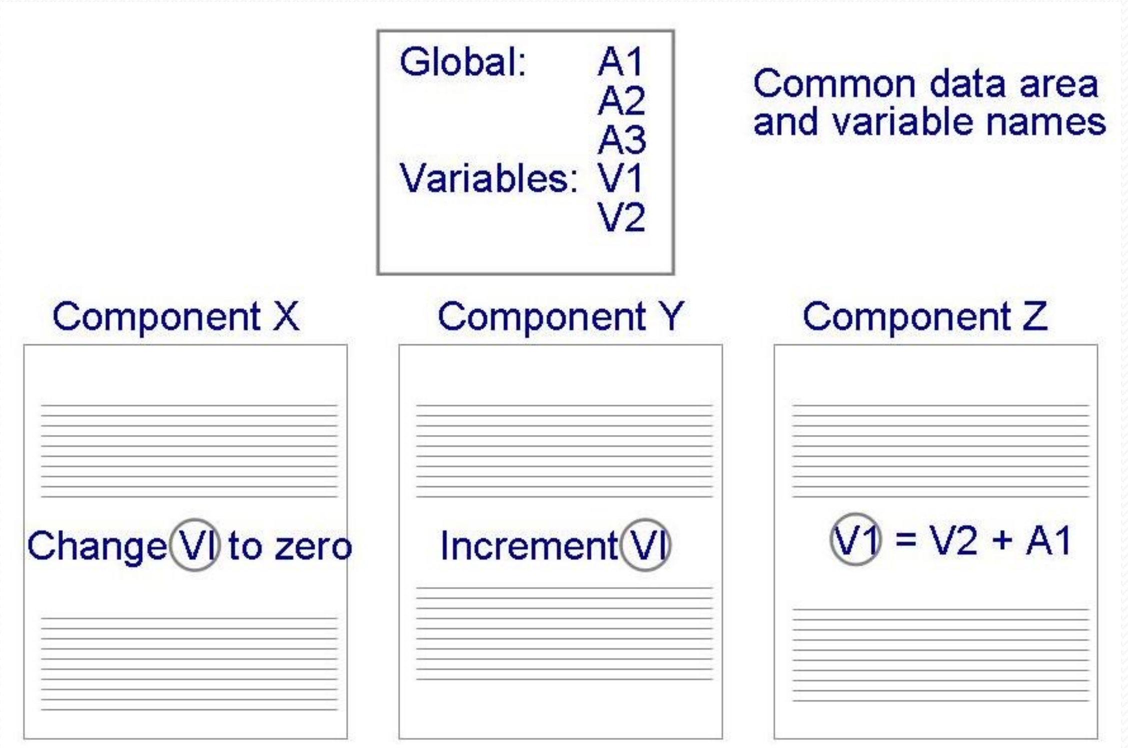
- One component modifies an internal data item of another component, a component (e.g. B) branches into another component (e.g. D)





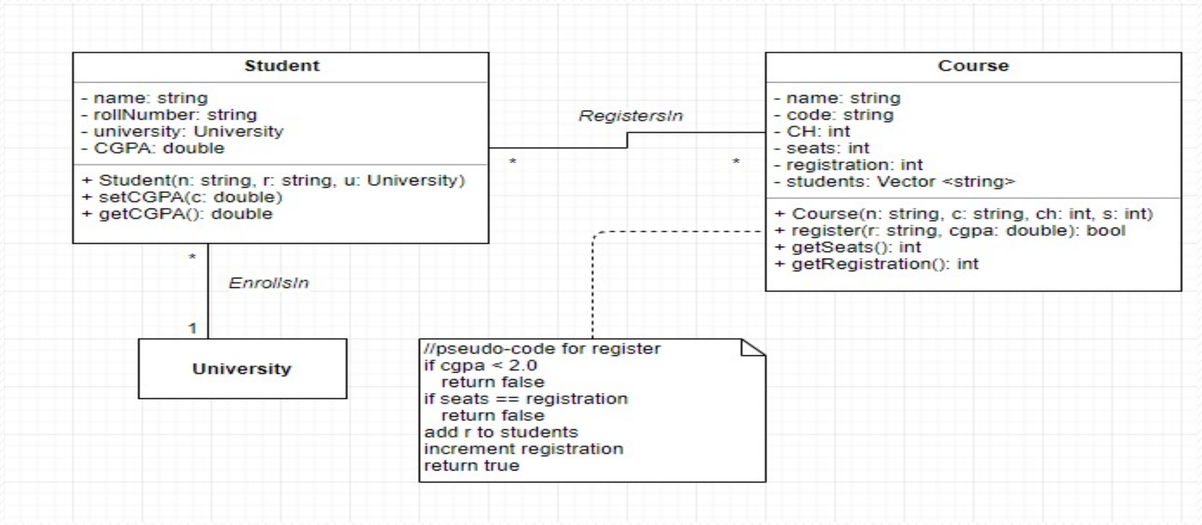
Modularity: Coupling: Common Coupling

- Dependency due to common data. Which component is responsible to set a particular value for the common variable?
  - Tracing back to all components that access those data to evaluate the effect of the change



Modularity: Coupling: Control Coupling

- One component passes parameters to control the activity of another component
- The controlled component cannot function without direction from the controlling component
- Make each component perform only one function



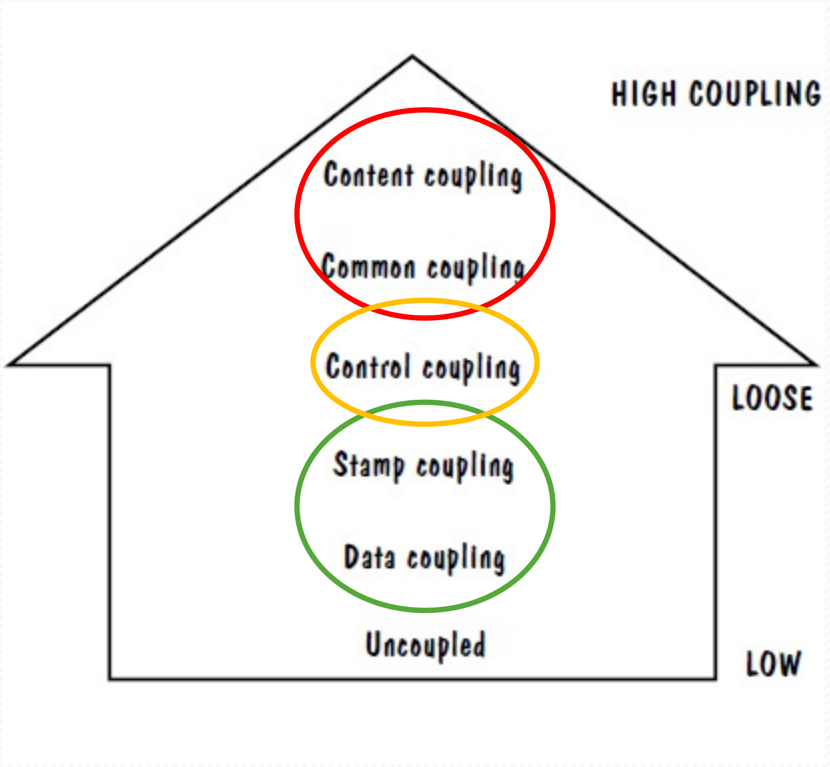
```
//pseudo-code for register
if cgpa < 2.0
   return false
if seats == registration
   return false
add r to students
increment registration
return true
```

Modularity: Coupling: Stamp and Data Coupling

- **Stamp Coupling**: Stamp coupling, or data structure coupling, occurs when modules share a composite data structure and use only a part of it, possibly different parts.
- **Stamp coupling** occurs when complex data structures are passed between modules
  - Stamp coupling represents a more complex interface between modules, because the modules have to agree on the data's format and organization
- **Data coupling**
- **Data Coupling**: Data coupling occurs when methods share data, regularly through parameters. Data coupling is better than stamp coupling, because the module takes exactly what it needs, without the need of it knowing the structure of a particular data structure.
  - Data coupling is simpler and less likely to be affected by changes in data representation
  - Only data value is passed from one component to another
  - Easiest to trace data and make changes

Modularity: Coupling: Types of Coupling

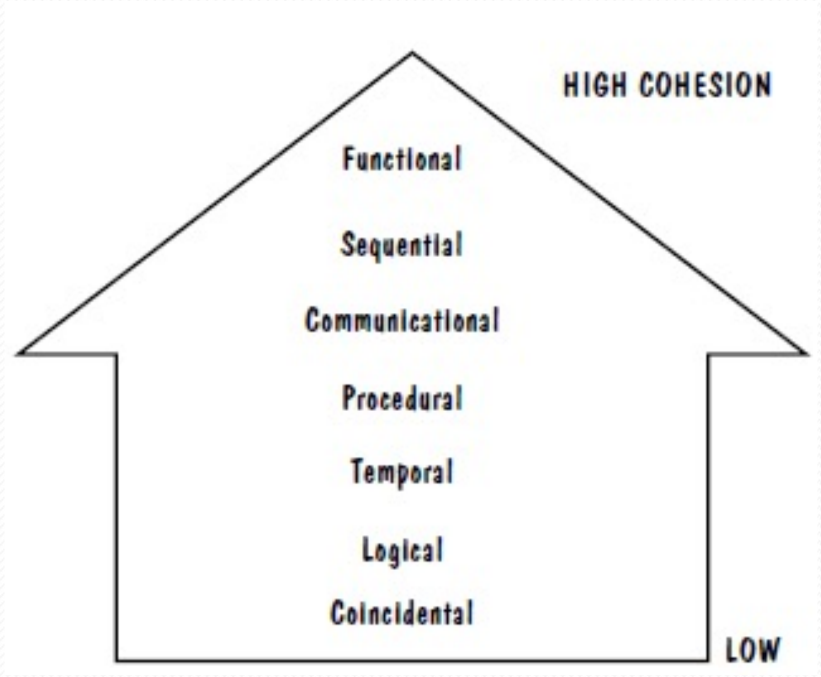
- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling



High coupling is NOT desired

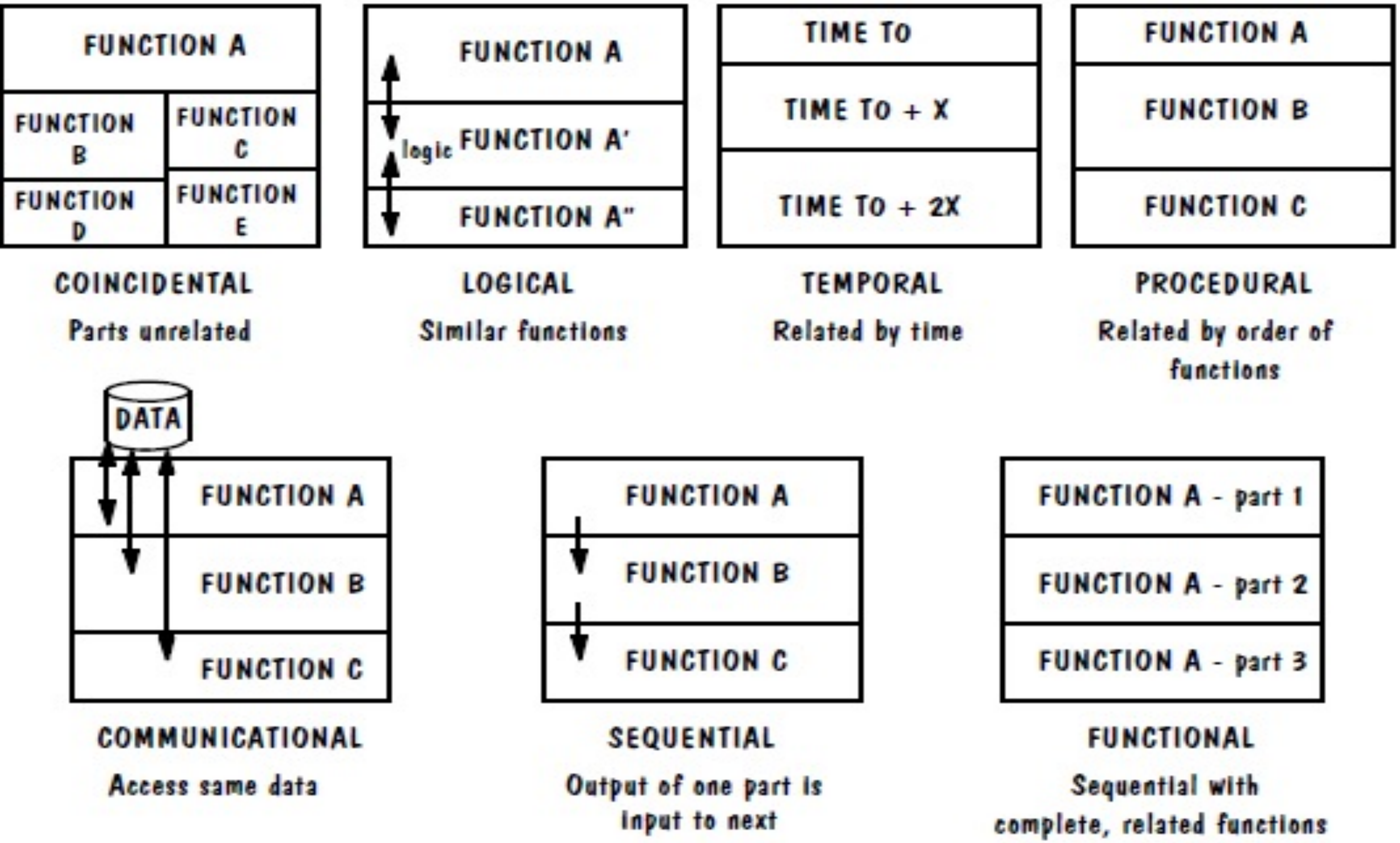
Modularity: Cohesion: Types of Cohesion

- **Cohesion** refers to the dependence within and among a module's internal elements (e.g., data, functions, internal modules)



Low cohesion is not desired

Modularity: Cohesion: Types of Cohesion



Types of cohesion

Coincidental Cohesion

- **E.g: Transaction Processing System: In a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module**

Logical Cohesion:

- **Logical cohesion is when parts of a module are grouped because they are logically categorized to do the same thing.**
- **Print Functions: An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.**

Temporal Cohesion:

The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Communicational Cohesion

Example: update record in the database and send it to the printer.

Sequential Cohesion:

e.g. a function which reads data from a file and processes the data).

Functional Cohesion.

Assign Seat to passenger

Modularity: Cohesion: Types of Cohesion

- **In Sequential cohesion**, activities are related and output for current activities is input for next activity but in procedural cohesion activities are unrelated.
- **Example for Sequential Cohesion** : Let us take an example of getting data from database. Below will be steps for this task.
1. Get result set from sql command
2. prepare result set
3. return result set

In this example sequence is followed and each activity's result is input for next activity. If any of the activity is not executed successfully then next activity will not executed.

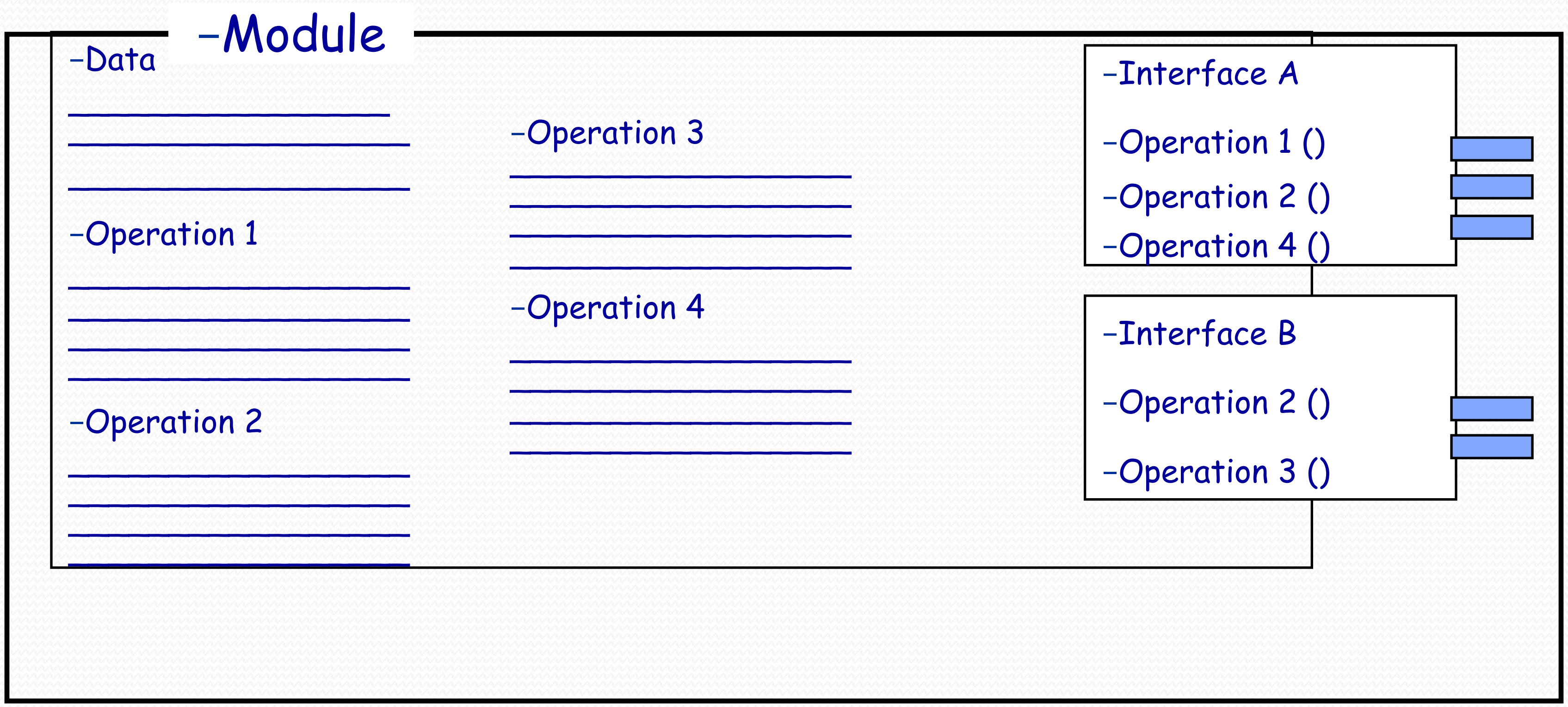
- Example for **Procedure Cohesion** : Let us take example of above module.
1. create connection string
2. Open connection using SqlConnection class
3. Execute sql command suing SqlCommand
4. Get resultset using SqlDataReader

Interfaces

- An **interface** defines what services the software unit provides to the rest of the system, and how other units can access those services

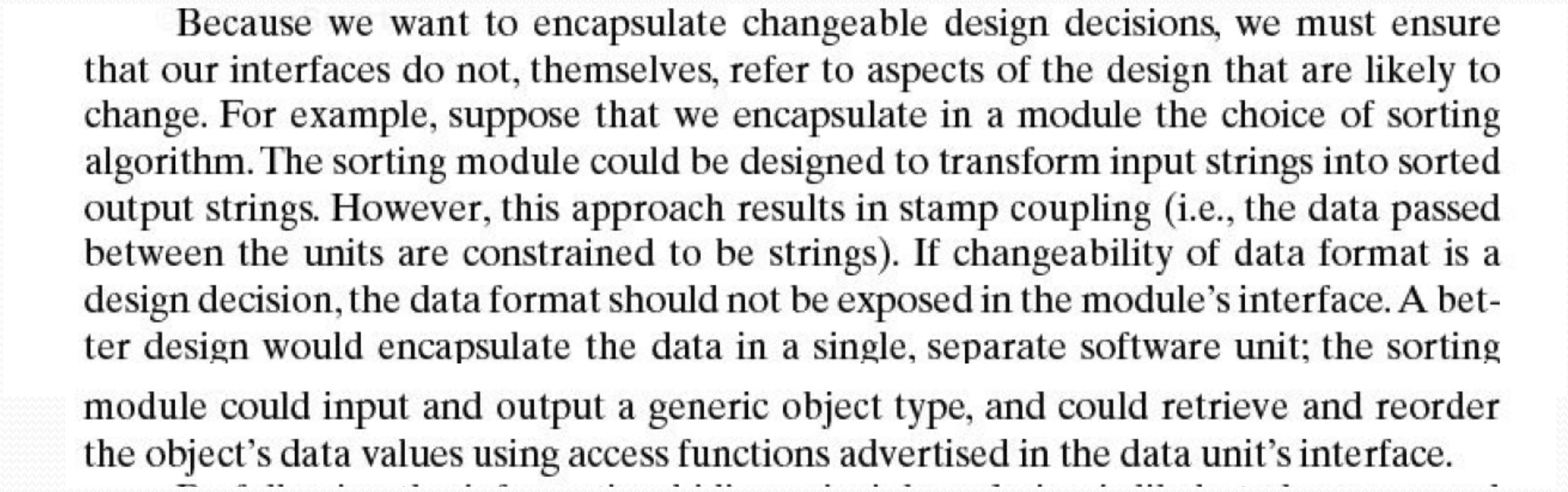
Interfaces (Contd.)

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



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



Because we want to encapsulate changeable design decisions, we must ensure that our interfaces do not, themselves, refer to aspects of the design that are likely to change. For example, suppose that we encapsulate in a module the choice of sorting algorithm. The sorting module could be designed to transform input strings into sorted output strings. However, this approach results in stamp coupling (i.e., the data passed between the units are constrained to be strings). If changeability of data format is a design decision, the data format should not be exposed in the module's interface. A better design would encapsulate the data in a single, separate software unit; the sorting module could input and output a generic object type, and could retrieve and reorder the object's data values using access functions advertised in the data unit's interface.

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

Generality

- **Generality** is the design principle that makes 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.