Software Design

Organization of this Lecture

- Ñ Brief review of previous lectures
- Ñ Introduction to software design
- Ñ Goodness of a design
- Ñ Functional Independence
- Ñ Cohesion and Coupling
- Ñ Function-oriented design vs. Object-oriented design
- Ñ Summary

Software design

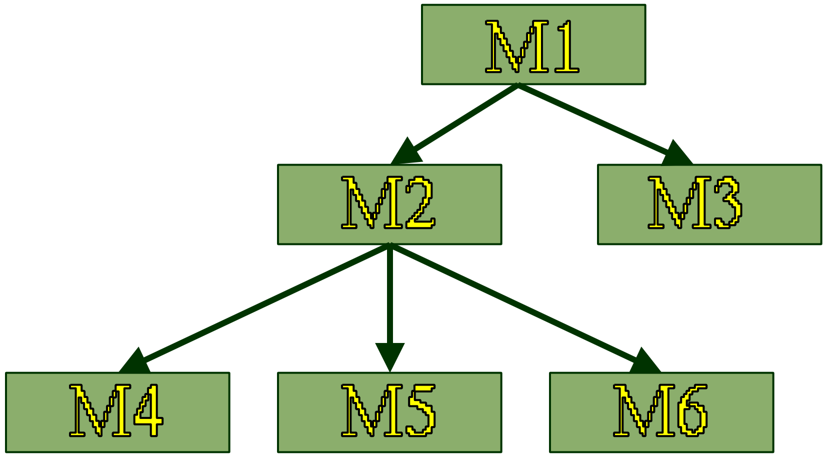
- Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language.

- Design activities can be broadly classified into two important parts:
  - Preliminary (or high-level) design and
  - Detailed design

Items Designed During Design Phase

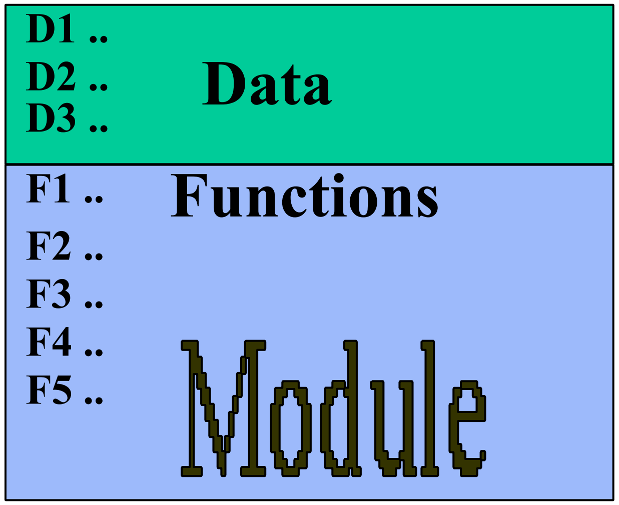
- module structure,
- control relationship among the modules
  - call relationship or invocation relationship
- interface among different modules,
  - data items exchanged among different modules,
- data structures of individual modules,
- algorithms for individual modules.

Module Structure



Introduction

- A module consists of:
  - several functions
  - associated data structures.



High Level and Detailed Design

- High-level design means identification of different **modules and the control relationships** among them and the definition of the interfaces among these modules.

- The outcome of high-level design is called the **program structure or software architecture.** Ex: Tree-like structure.

- Detailed design, the **data structure and the algorithms** of the different modules are designed.

- The outcome of the detailed design stage is usually known as the **module-specification document**.

What Is Good Software Design?

- Should implement all functionalities of the system correctly.
- Should be easily understandable.
- Should be efficient.
- Should be easily amenable to change,
  - i.e. easily maintainable.
- Understandability of a design is a major issue:
  - determines goodness of design:
  - a design that is easy to understand:
    - also easy to maintain and change.

Understandability

- Use consistent and meaningful names
  - for various design components,
- Design solution should consist of:
  - a cleanly decomposed set of modules (modularity),
- Different modules should be neatly arranged in a hierarchy:
  - in a neat tree-like diagram.

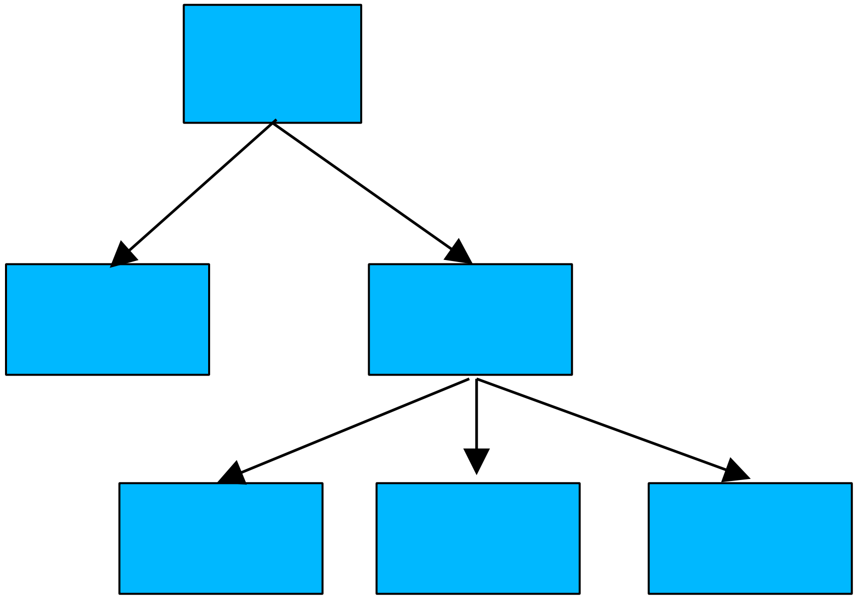
Modularity

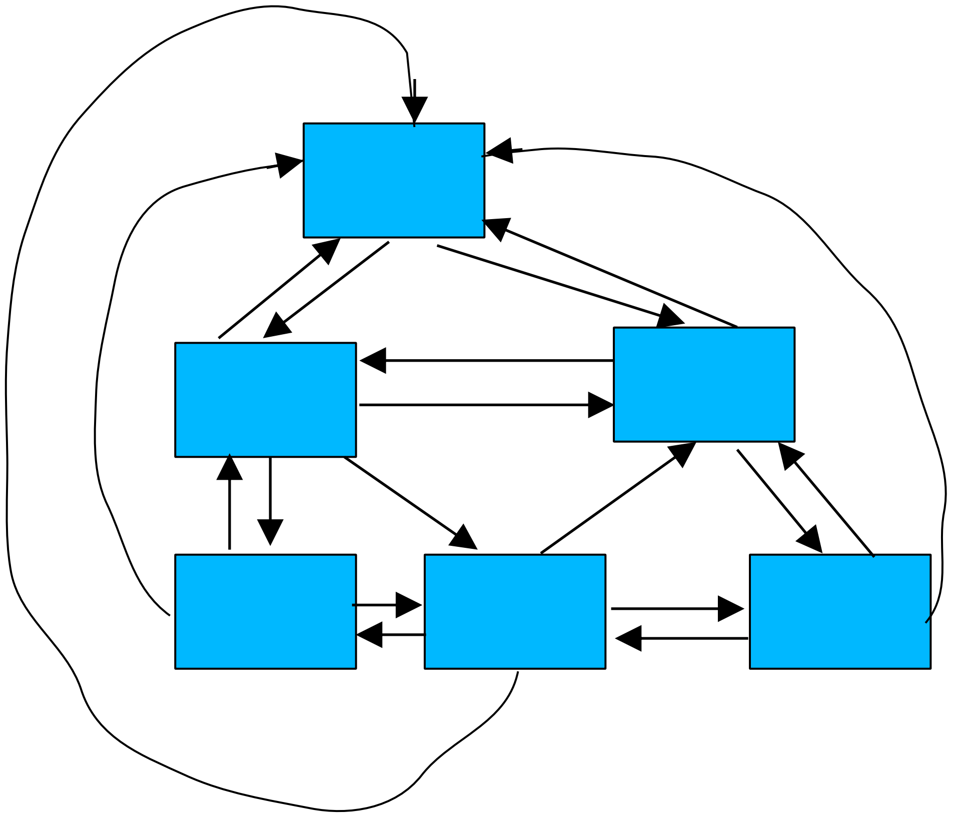
- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem cleanly into modules:
  - Modules are almost independent of each other
  - divide and conquer principle.

Modularity

- If modules are independent:
  - modules can be understood separately,
    - reduces the complexity greatly.
  - To understand why this is so,
    - remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.

Example of Cleanly and Non-cleanly Decomposed Modules





Modularity

- In technical terms, modules should display:
  - high cohesion
  - low coupling.

Cohesion and Coupling

- Cohesion is a measure of:
  - functional strength of a module.
  - A cohesive module performs a single task or function.
- Coupling between two modules:
  - a measure of the degree of interdependence or interaction between the two modules.

Cohesion and Coupling

- A module having high cohesion and low coupling:
  - functionally independent of other modules:
    - A functionally independent module has minimal interaction with other modules.

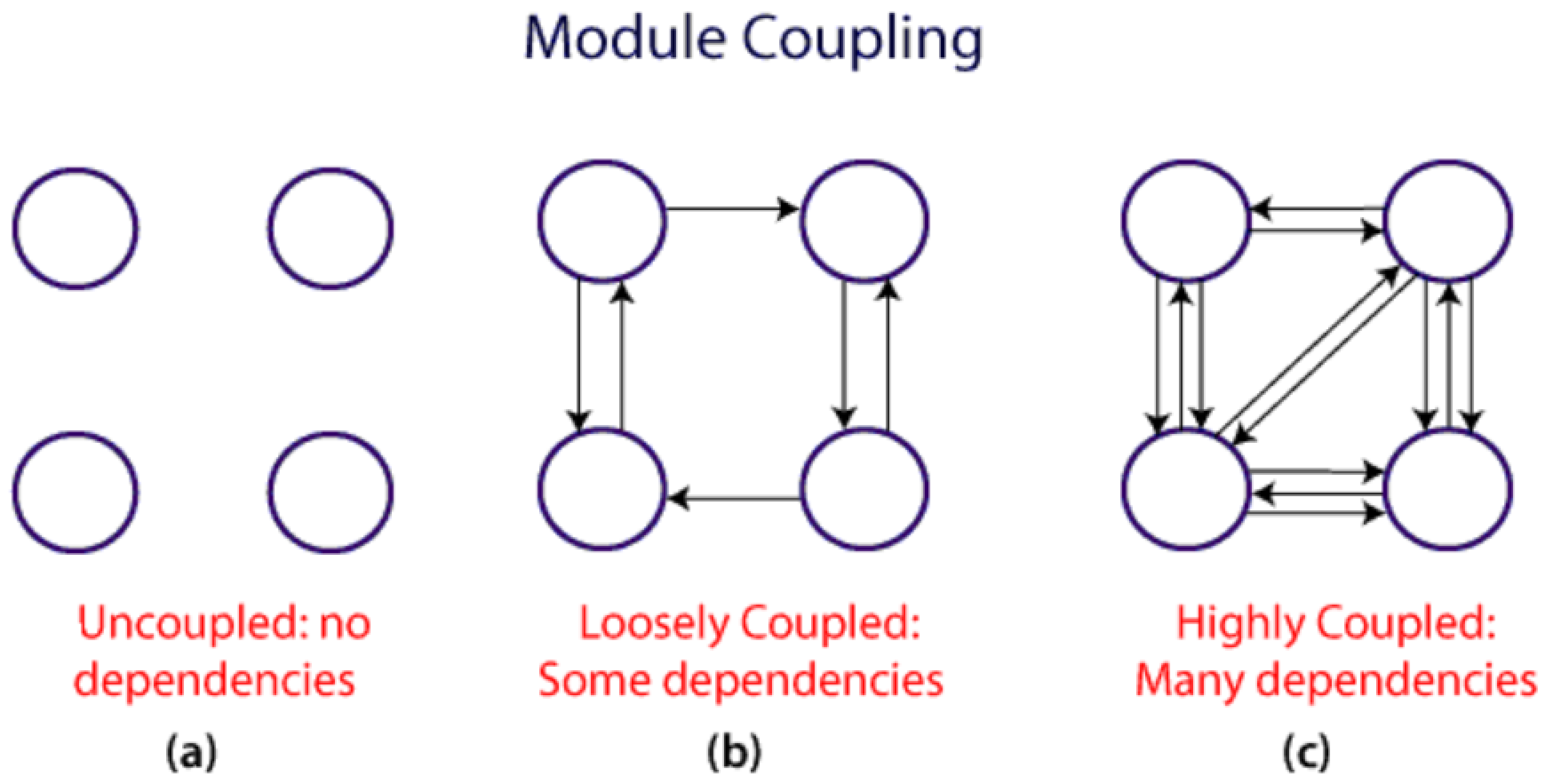
Advantages of Functional Independence

- Better understandability and good design:
- Complexity of design is reduced,
- Different modules easily understood in isolation:
  - modules are independent

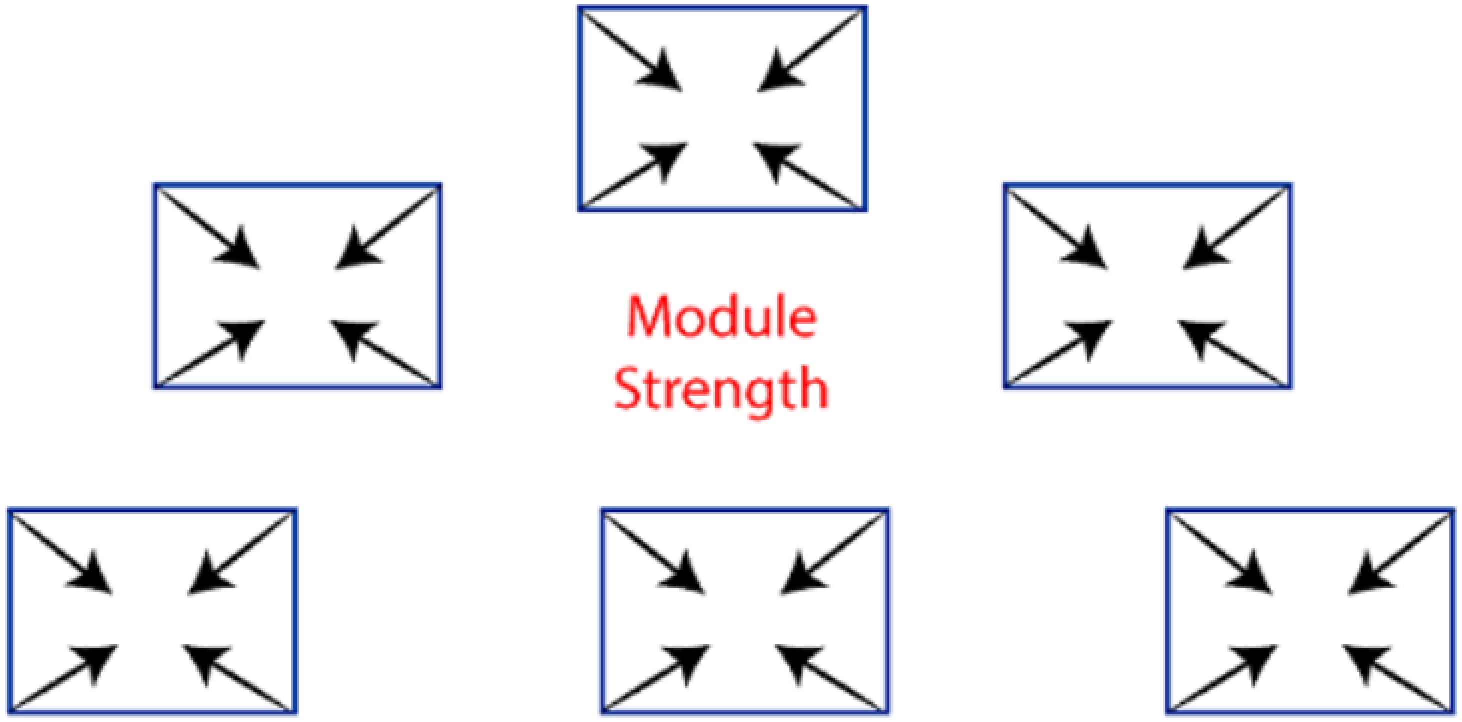
Advantages of Functional Independence

- Functional independence reduces error propagation.
  - degree of interaction between modules is low.
  - an error existing in one module does not directly affect other modules.
- Reuse of modules is possible.

The various types of coupling techniques are shown in fig:

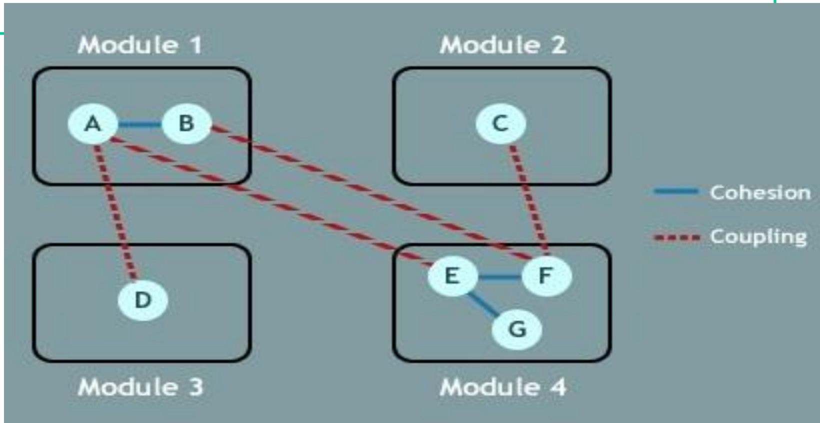

Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

Cohesion

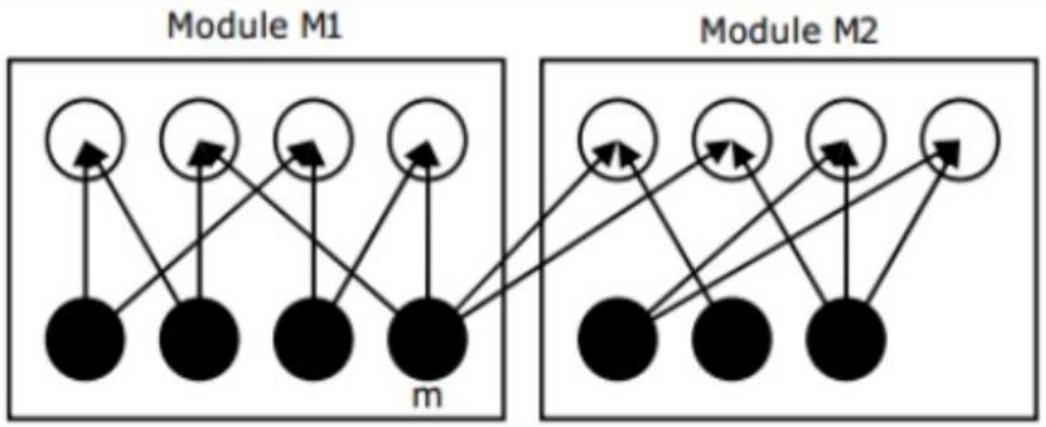


Cohesion= Strength of relations within Modules

Cohesion	Coupling
Cohesion is the indication of the relationship **within** module.	Coupling is the indication of the relationships **between** modules.
Cohesion shows the module's **relative functional strength**.	Coupling shows the **relative independence among the modules**.
Cohesion is a degree (quality) to which a component / module **focuses on the single thing.**	Coupling is a degree to which a component / module is **connected to the other modules.**
While designing you should strive for high cohesion i.e. a cohesive component/ module focus on a single task.	While designing you should strive for low coupling i.e. Dependency between modules should be less.
Cohesion is **Intra – Module** Concept.	Coupling is **Inter -Module** Concept.



Filled circles represents Methods
Unfilled circle represent Attributes



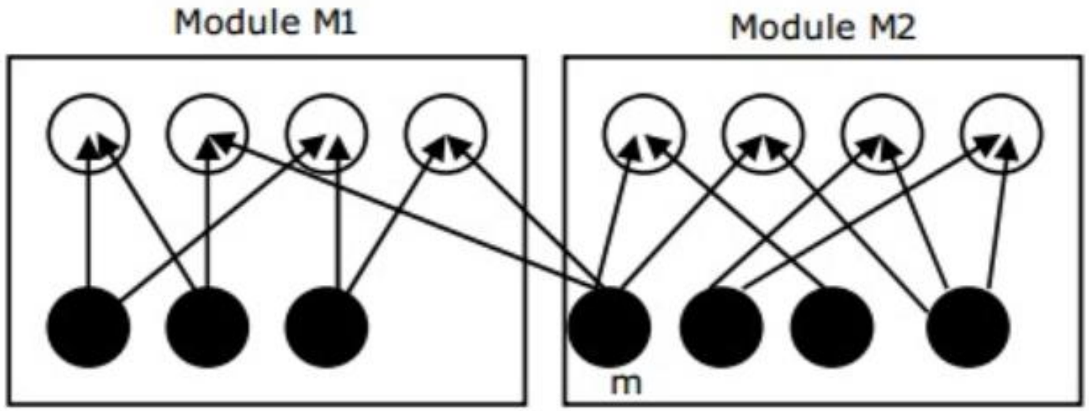
$$\text{Coupling} = \frac{\text{number of external links}}{\text{number of modules}} = \frac{2}{2}$$

$$\text{Cohesion of a module} = \frac{\text{number of internal links}}{\text{number of methods}}$$

$$\text{Cohesion of } M_1 = \frac{8}{4}; \text{ Cohesion of } M_2 = \frac{6}{3}; \quad \text{Average cohesion=2}$$

Move method m to M2

After moving method m to M2, graph will become

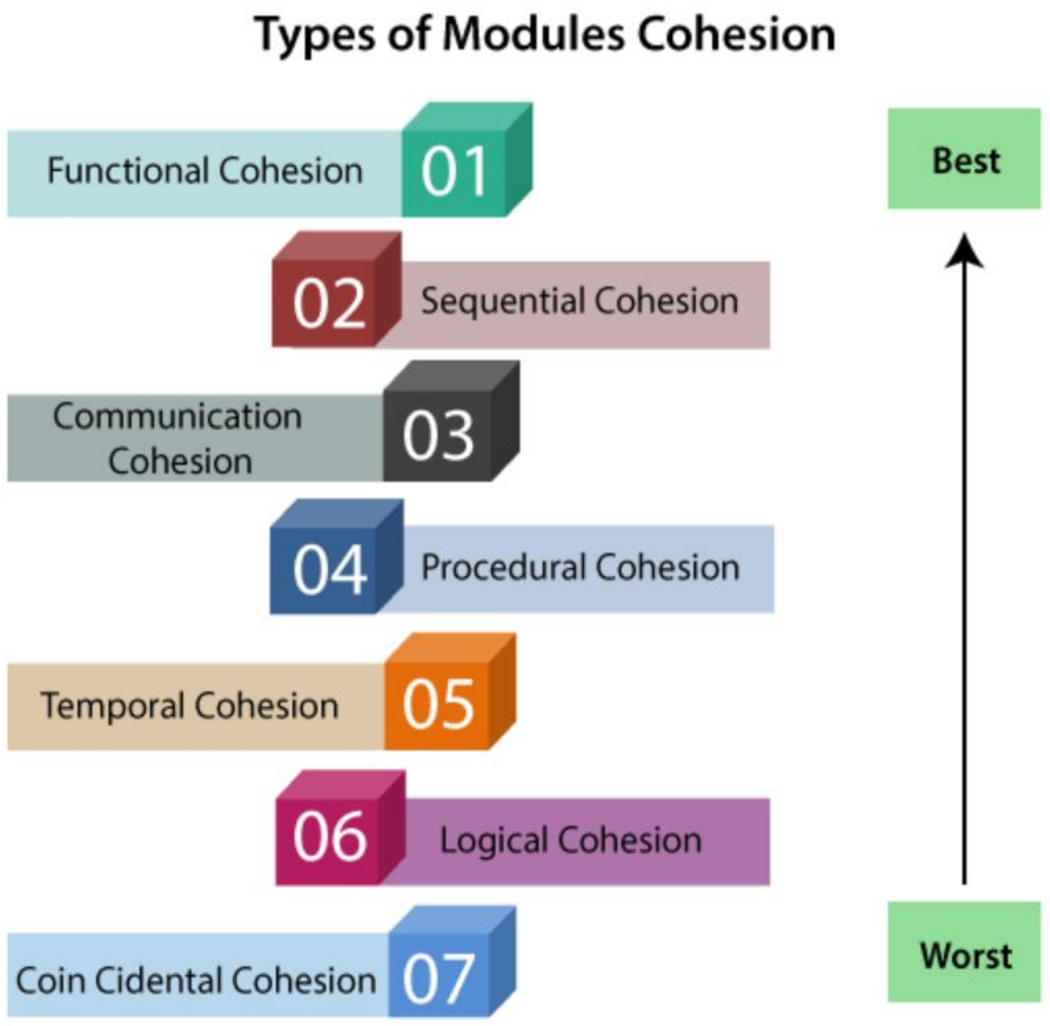


$\text{Coupling} = \frac{2}{2}$

Cohesion of $M_1 = \frac{6}{3}$; Cohesion of $M_2 = \frac{8}{4}$; Average cohesion=2

$\therefore$ answer is no change

Types of Modules Cohesion



Classification of Cohesiveness

- Classification is often subjective:
  - yet gives us some idea about cohesiveness of a module.
- By examining the type of cohesion exhibited by a module:
  - we can roughly tell whether it displays high cohesion or low cohesion.

- **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

- **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

- **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.

- **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

- **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

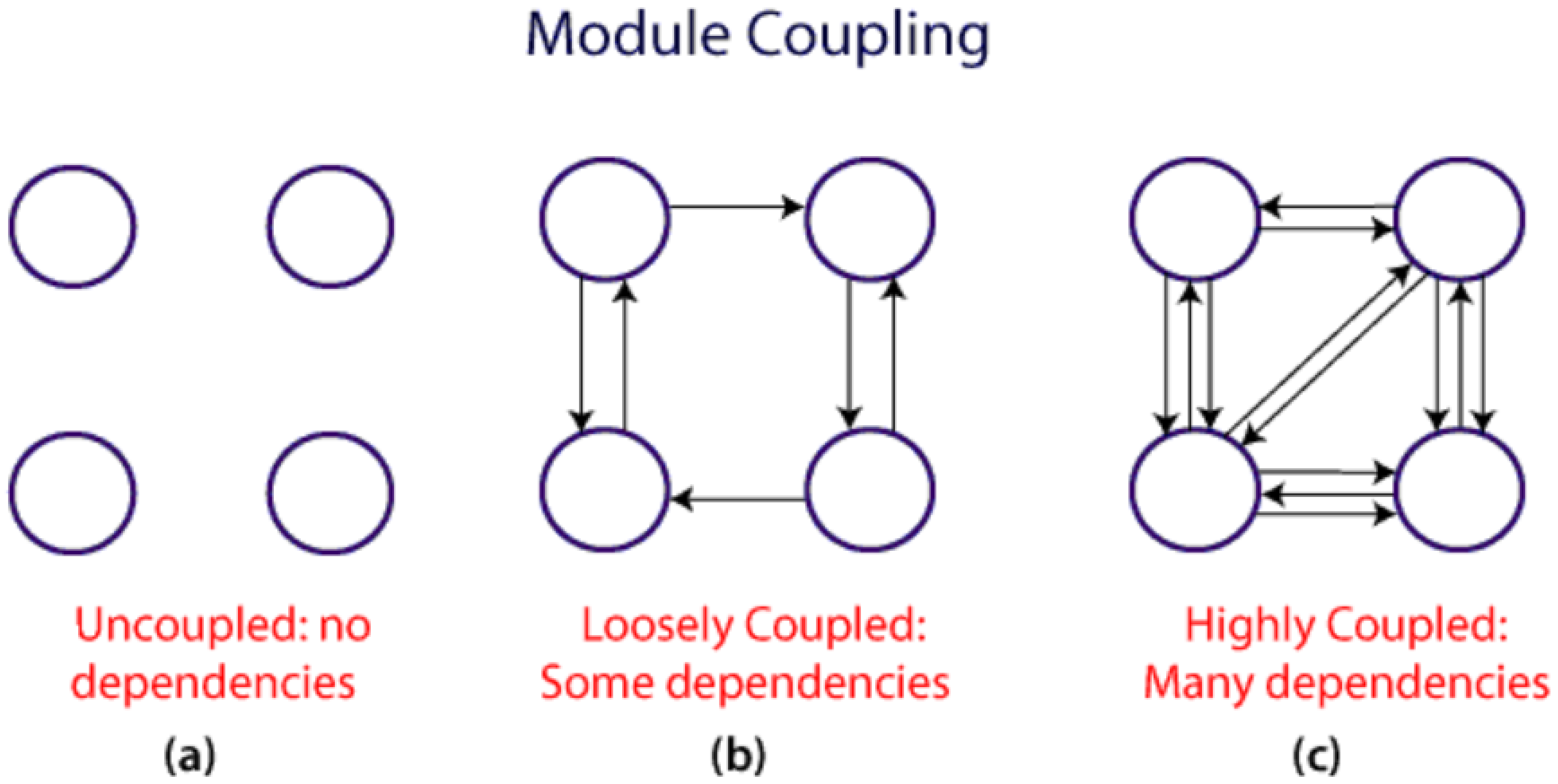
- **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

- **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

Coupling

- Coupling indicates:
  - how closely two modules interact or how interdependent they are.
  - The degree of coupling between two modules depends on their interface complexity.

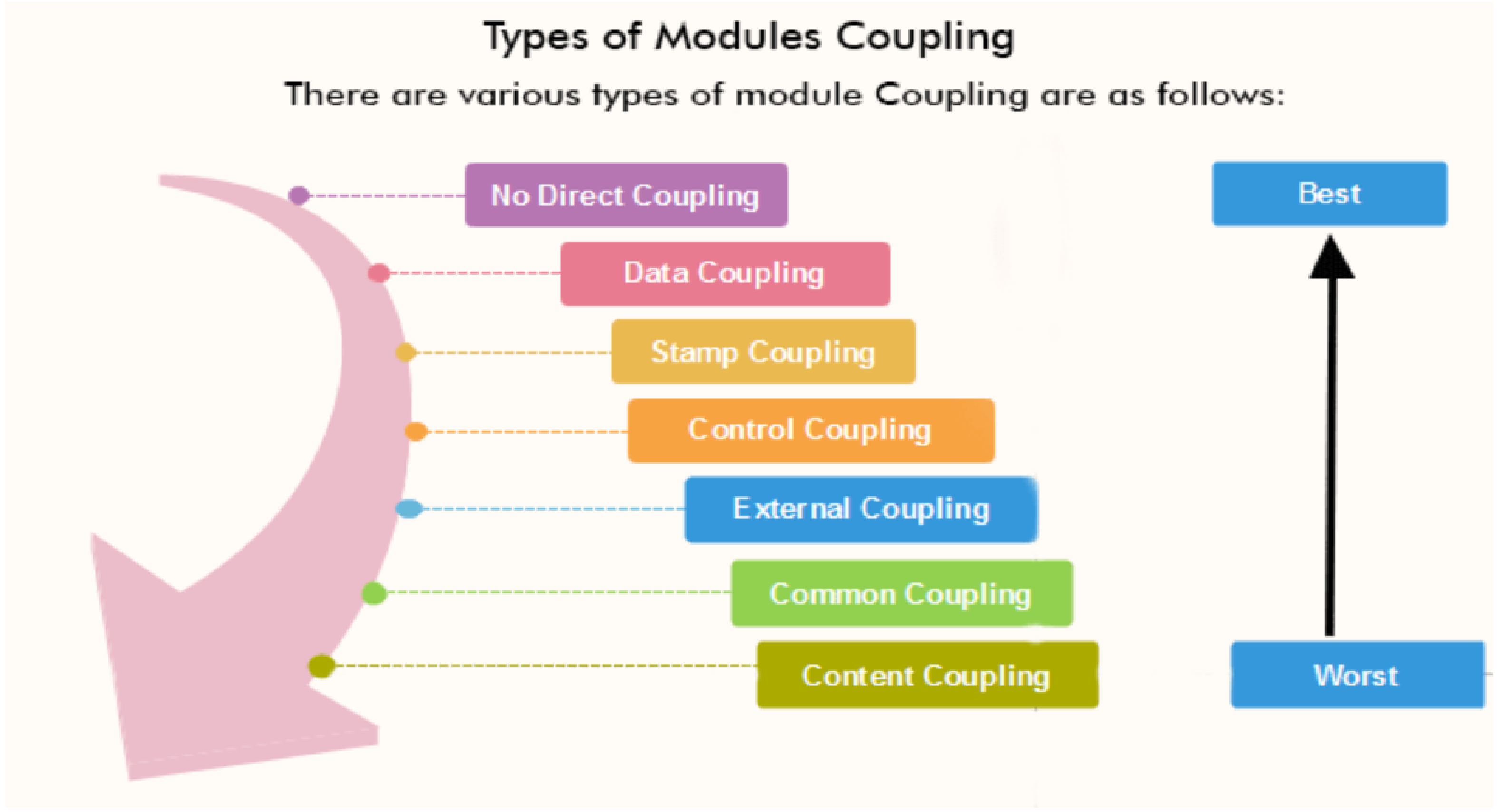
The various types of coupling techniques are shown in fig:


Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

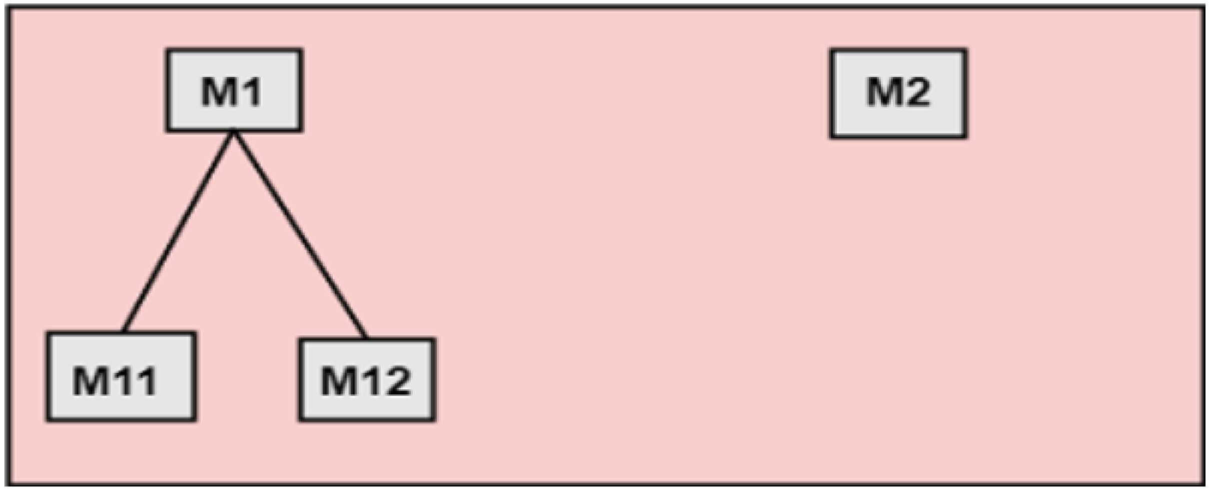
Coupling

- There are no ways to precisely determine coupling between two modules:
  - classification of different types of coupling will help us to approximately estimate the degree of coupling between two modules.
- Five types of coupling can exist between any two modules.

Types of Module Coupling

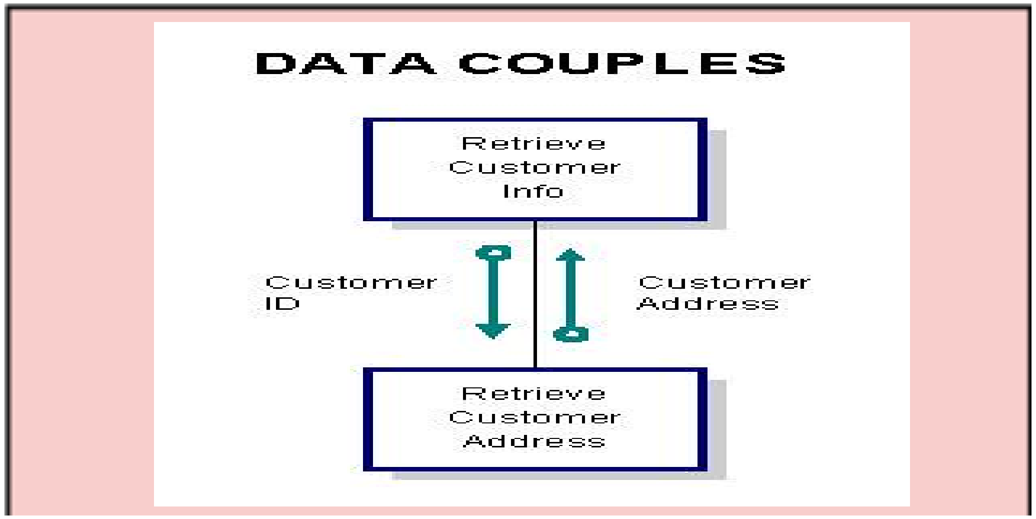


1. No Direct Coupling: There is no direct coupling between M1 and M2.



In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. Data Coupling: When data of one module is passed to another module, this is called data coupling.

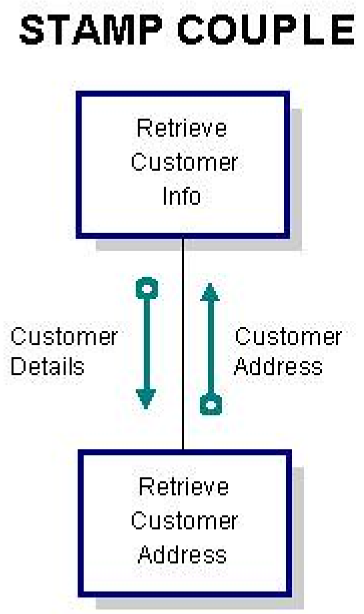


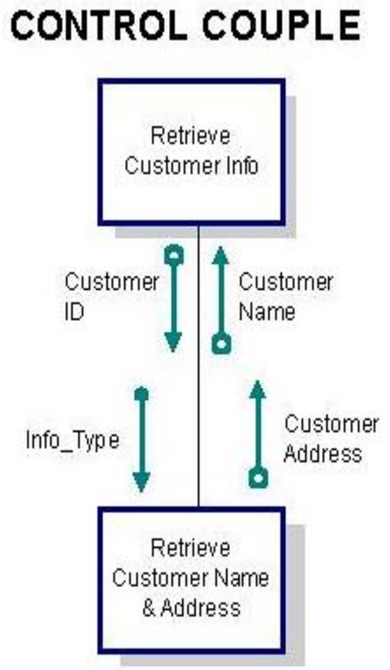
Data coupling occurs between two modules when data are passed by parameters using a simple argument list and every item in the list is used.

An example of data coupling is a module which retrieves customer address using customer id.

Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled.

An example of stamp coupling where a module that retrieves customer address using only customer id which is extracted from a parameter named customer details.



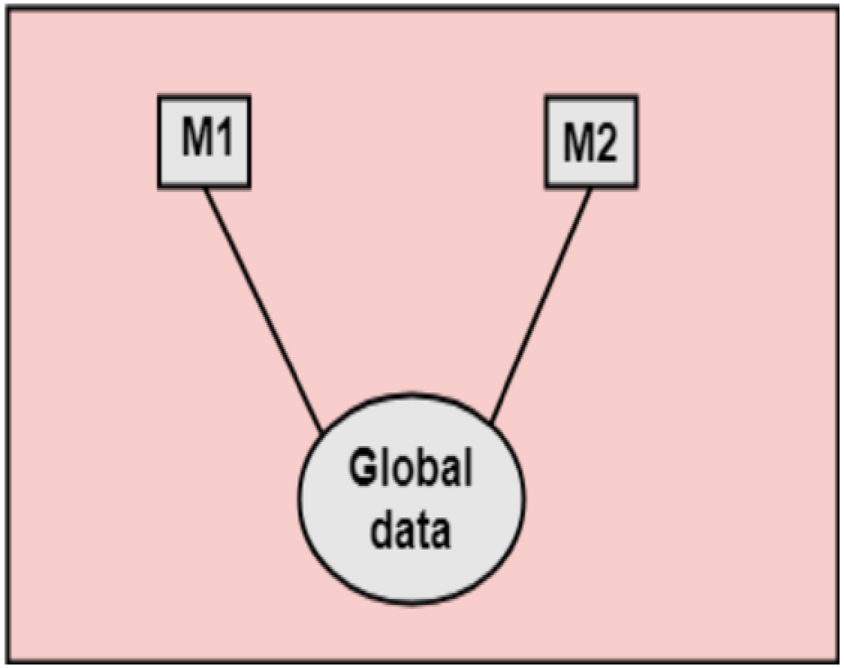


Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

example of control coupling, a module that retrieves either a customer name or an address depending on the value of a flag is illustrated.

External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. Common Coupling: Two modules are common coupled if they share information through some global data items.



7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Neat Hierarchy

- Control hierarchy represents:
  - organization of modules.
  - control hierarchy is also called program structure.
- Most common notation:
  - a tree-like diagram called structure chart.

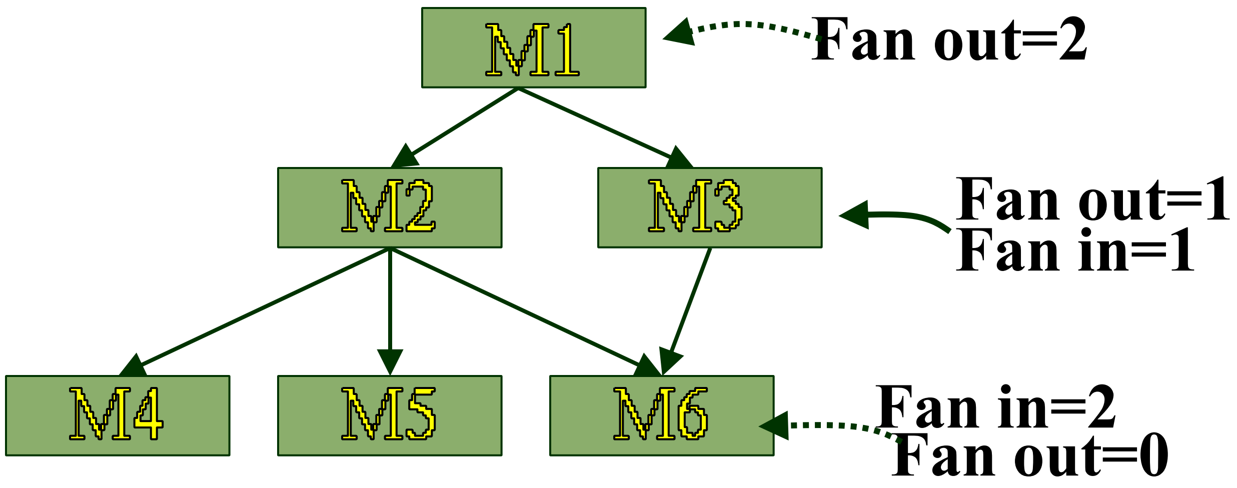
Neat Arrangement of modules

- Essentially means:
  - low fan-out
  - abstraction

Characteristics of Module Structure

- Depth:
  - number of levels of control
- Width:
  - overall span of control.
- Fan-out:
  - a measure of the number of modules directly controlled by given module.
- Fan-in:
  - indicates how many modules directly invoke a given module.
  - High fan-in represents code reuse and is in general encouraged.

Module Structure



Goodness of Design

- A design having modules:
  - with high fan-out numbers is not a good design:
  - a module having high fan-out lacks cohesion.

- A module that invokes a large number of other modules:
  - likely to implement several different functions:
  - not likely to perform a single cohesive function.

Control Relationships

- A module that controls another module:
  - said to be superordinate to it.

- Conversely, a module controlled by another module:
  - said to be subordinate to it.

Neat Hierarchy

- Control hierarchy represents:
  - organization of modules.
  - control hierarchy is also called program structure.
- Most common notation:
  - a tree-like diagram called structure chart.

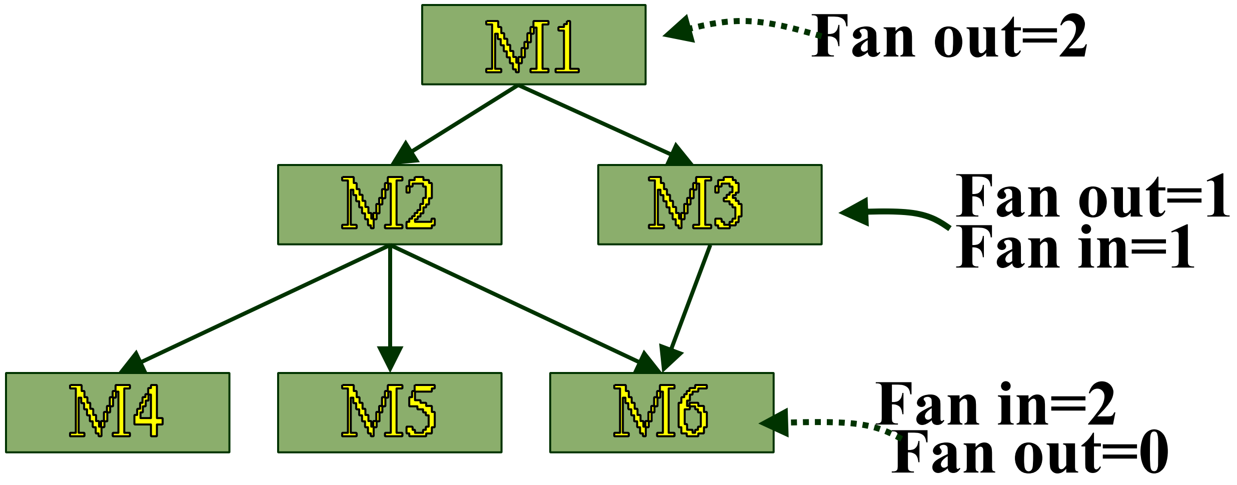
Neat Arrangement of modules

- Essentially means:
  - low fan-out
  - abstraction

Characteristics of Module Structure

- Depth:
  - number of levels of control
- Width:
  - overall span of control.
- Fan-out:
  - a measure of the number of modules directly controlled by given module.
- Fan-in:
  - indicates how many modules directly invoke a given module.
  - High fan-in represents code reuse and is in general encouraged.

Module Structure



Control Relationships

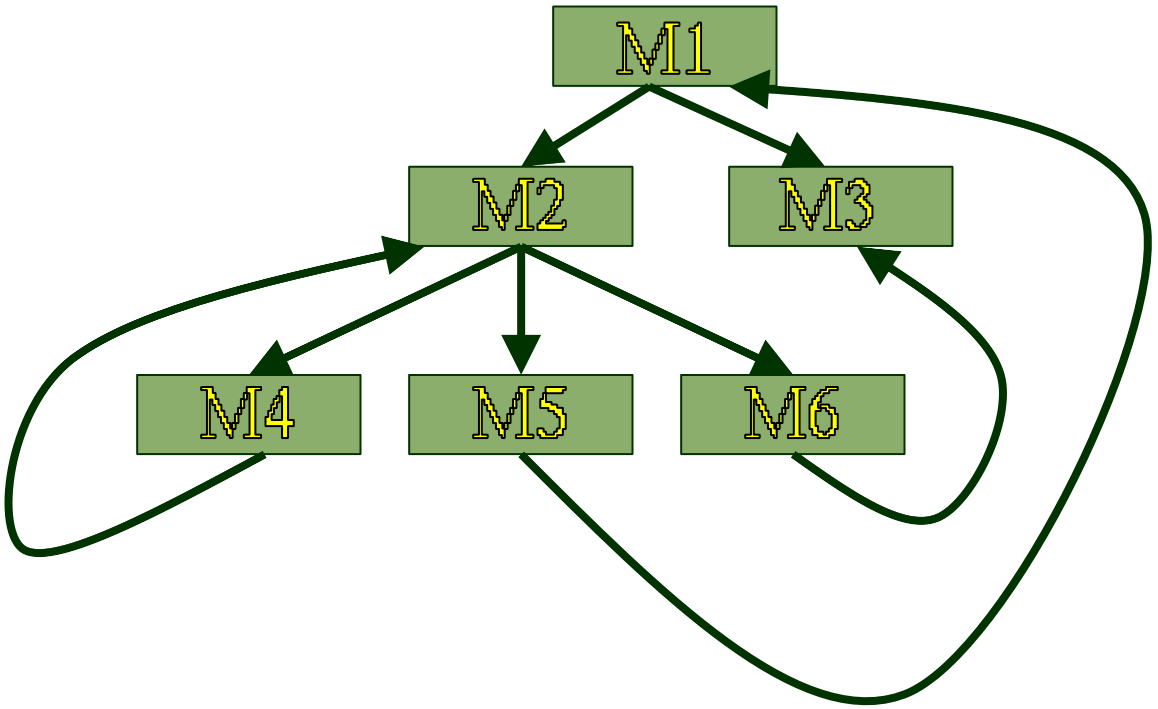
- A module that controls another module:
  - said to be superordinate to it.
- Conversely, a module controlled by another module:
  - said to be subordinate to it.

Visibility and Layering

- A module A is said to be visible by another module B,
  - if A directly or indirectly calls B (Embedding).

- The layering principle requires
  - modules at a layer can call only the modules immediately below it (Sequence).

Bad Design



Abstraction

- Lower-level modules:
  - do input/output and other low-level functions.
- Upper-level modules:
  - do more managerial functions.
- The principle of abstraction requires:
  - lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.

High-level Design

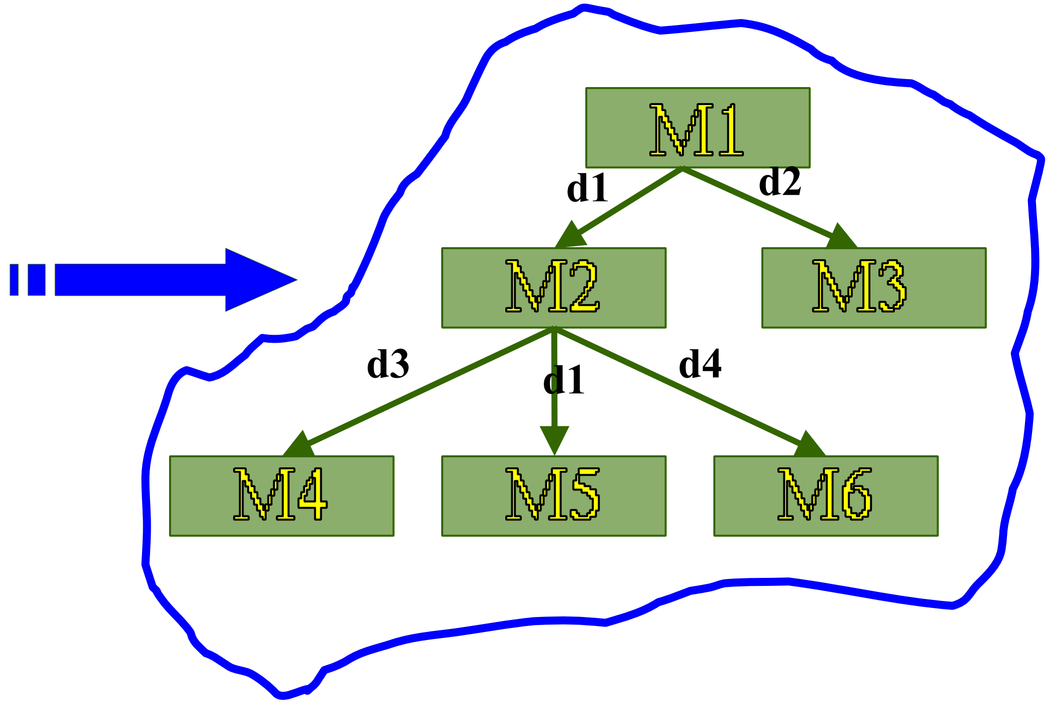
- High-level design maps functions into modules {fi} {mj} such that:
  - Each module has high cohesion
  - Coupling among modules is as low as possible
  - Modules are organized in a neat hierarchy

High-level Design

- f1
- f2
- f3

•
•
•

- fn



Design Approaches

- Two fundamentally different software design approaches:
  - Function-oriented design
  - Object-oriented design

- These two design approaches are radically different.
  - However, are complementary
    - rather than competing techniques.
  - Each technique is applicable at
    - different stages of the design process.

Function-Oriented Design

- A system is looked upon as something
  - that performs a set of functions.

- Starting at this high-level view of the system:
  - each function is successively refined into more detailed functions.
  - Functions are mapped to a module structure.

- Example : The function create-new-library- member:
  - **creates** the record for a new member,
  - **assigns** a unique membership number
  - **prints** a bill towards the membership

Example

- Create-library-member function consists of the following sub-functions:
  - assign-membership-number
  - create-member-record
  - print-bill

Function-Oriented Design

- Each subfunction:
  - split into more detailed subfunctions and so on.

- The system state is centralized:
  - accessible to different functions,
  - member-records:
    - available for reference and updation to several functions:
      - create-new-member
      - delete-member
      - update-member-record

Object-Oriented Design

- System is viewed as a collection of objects (i.e. entities).

- System state is decentralized among the objects:
  - each object manages its own state information.

Example:

- Library Automation Software:
  - each library member is a separate object
    - with its own data and functions.

  - Functions defined for one object:
    - cannot directly refer to or change data of other objects.

Object-Oriented Design

Ñ Objects have their own internal data:
  y defines their state.

Ñ Similar objects constitute a class.
  y each object is a member of some class.

Ñ Classes may inherit features
  y from a super class.

Ñ Conceptually, objects communicate by message passing.

Object-Oriented versus Function-Oriented Design

- Unlike function-oriented design,
  - in OOD the basic abstraction is not functions such as “sort”, “display”, “track”, etc.,
  - but real-world entities such as “employee”, “picture”, “machine”, “radar system”, etc.

- In OOD:
  - software is not developed by designing functions such as:
    - update-employee-record,
    - get-employee-address, etc.
  - but by designing objects such as:
    - employees,
    - departments, etc.

Example:

- In an employee pay-roll system, the following can be global data:
  - names of the employees,
  - their code numbers,
  - basic salaries, etc.

- Whereas, in object oriented systems:
  - data is distributed among different employee objects of the system.

Object-Oriented versus Function-Oriented Design

- Function-oriented techniques group functions together if:
  - as a group, they constitute a higher level function.
- On the other hand, object-oriented techniques group functions together:
  - on the basis of the data they operate on.
- To illustrate the differences between object-oriented and function-oriented design approaches,
  - let us consider an example ---
  - An automated fire-alarm system for a large building.

Fire-Alarm System:

- We need to develop a computerized fire alarm system for a large multi-storied building:
  - There are 80 floors and 1000 rooms in the building.
- Different rooms of the building:
  - fitted with smoke detectors and fire alarms.
- The fire alarm system would monitor:
  - status of the smoke detectors.
- Whenever a fire condition is reported by any smoke detector:
  - the fire alarm system should:
    - determine the location from which the fire condition was reported
    - sound the alarms in the neighboring locations.

Fire-Alarm System

Ñ The fire alarm system should:
  y flash an alarm message on the computer console:
    x fire fighting personnel man the console round the clock.

Ñ After a fire condition has been successfully handled,
  y the fire alarm system should let fire fighting personnel reset the alarms.

Function-Oriented Approach:

- /* Global data (system state) accessible by various functions */

```
BOOL  detector_status[1000];
int   detector_locs[1000];
BOOL  alarm-status[1000]; /* alarm activated when status set */
int   alarm_locs[1000]; /* room number where alarm is located */
int   neighbor-alarms[1000][10];/*each detector has at most*/
                          /* 10 neighboring alarm locations */
```

- The functions which operate on the system state:

```
interrogate_detectors();
get_detector_location();
determine_neighbor();
ring_alarm();
reset_alarm();
report_fire_location();
```

Object-Oriented Approach:

- class detector
  attributes: status, location, neighbors
  operations: create, sense-status, get-location, find-neighbors
- class alarm
  attributes: location, status
  operations: create, ring-alarm, get_location, reset-alarm
- In the object oriented program,
  appropriate number of instances of the class detector and alarm should be created.