Software Design

Organization of this Lecture

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

Review of previous lectures

- Introduction to software engineering
- Life cycle models
- Requirements Analysis and Specification:
  - Requirements gathering and analysis
  - Requirements specification

Difference between analysis and design

- Aim of analysis is to understand the problem with a view to eliminate any deficiencies in the requirement specification such as incompleteness, inconsistencies, etc. The model which we are trying to build may be or may not be ready.

- Aim of design is to produce a model that will provide a seamless transition to the coding phase, i.e. once the requirements are analyzed and found to be satisfactory, a design model is created which can be easily implemented.

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.

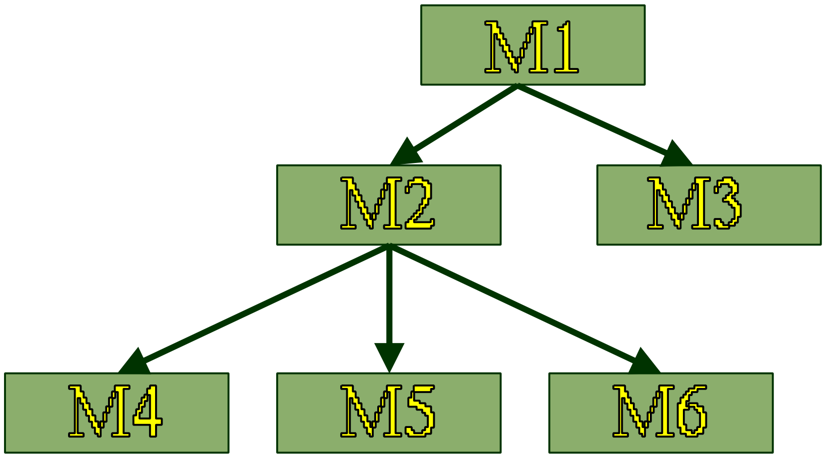
- A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps.

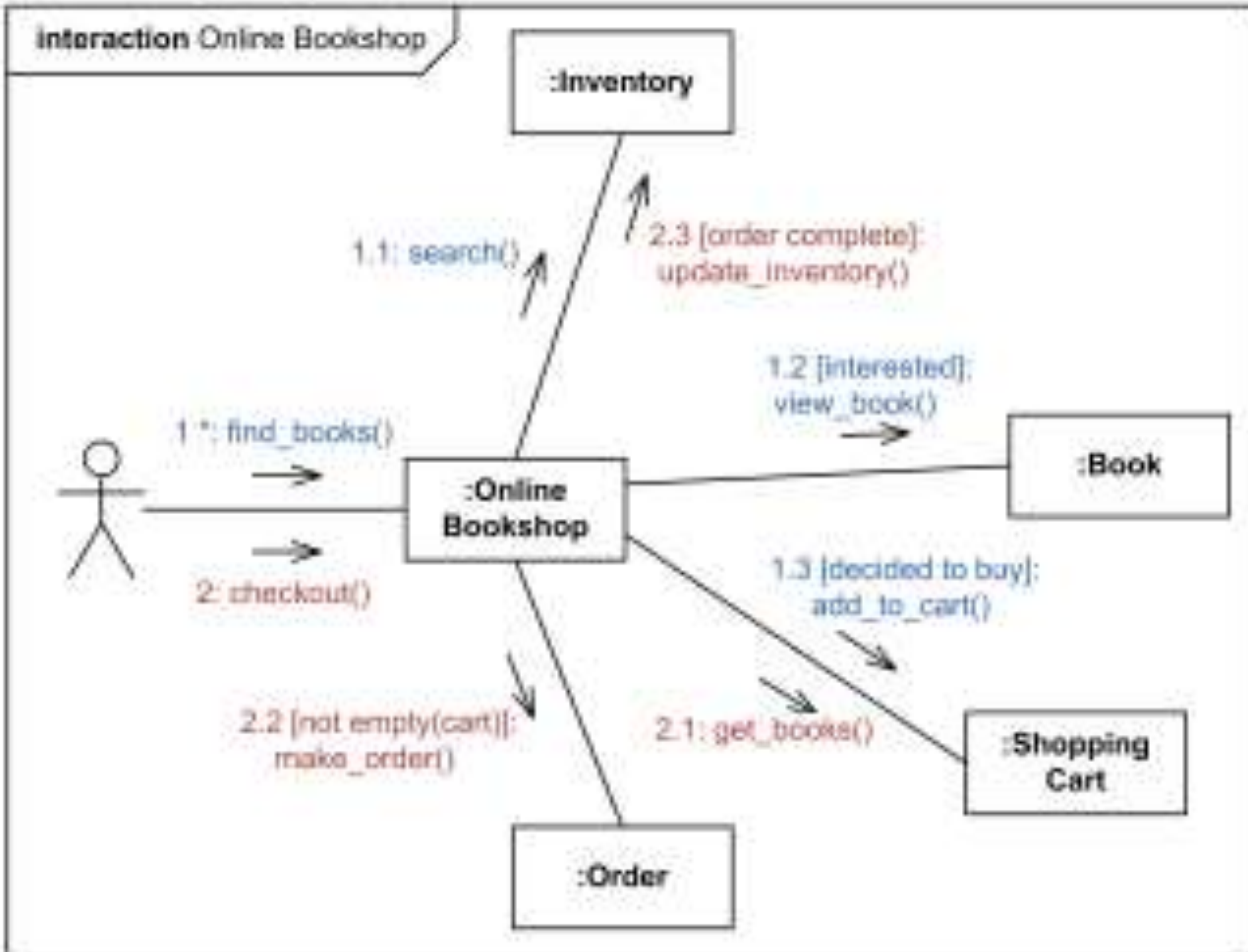
- Design activities can be broadly classified into two important parts:
  - Preliminary (or high-level) design and
  - Detailed design

Items Designed During Design Phase

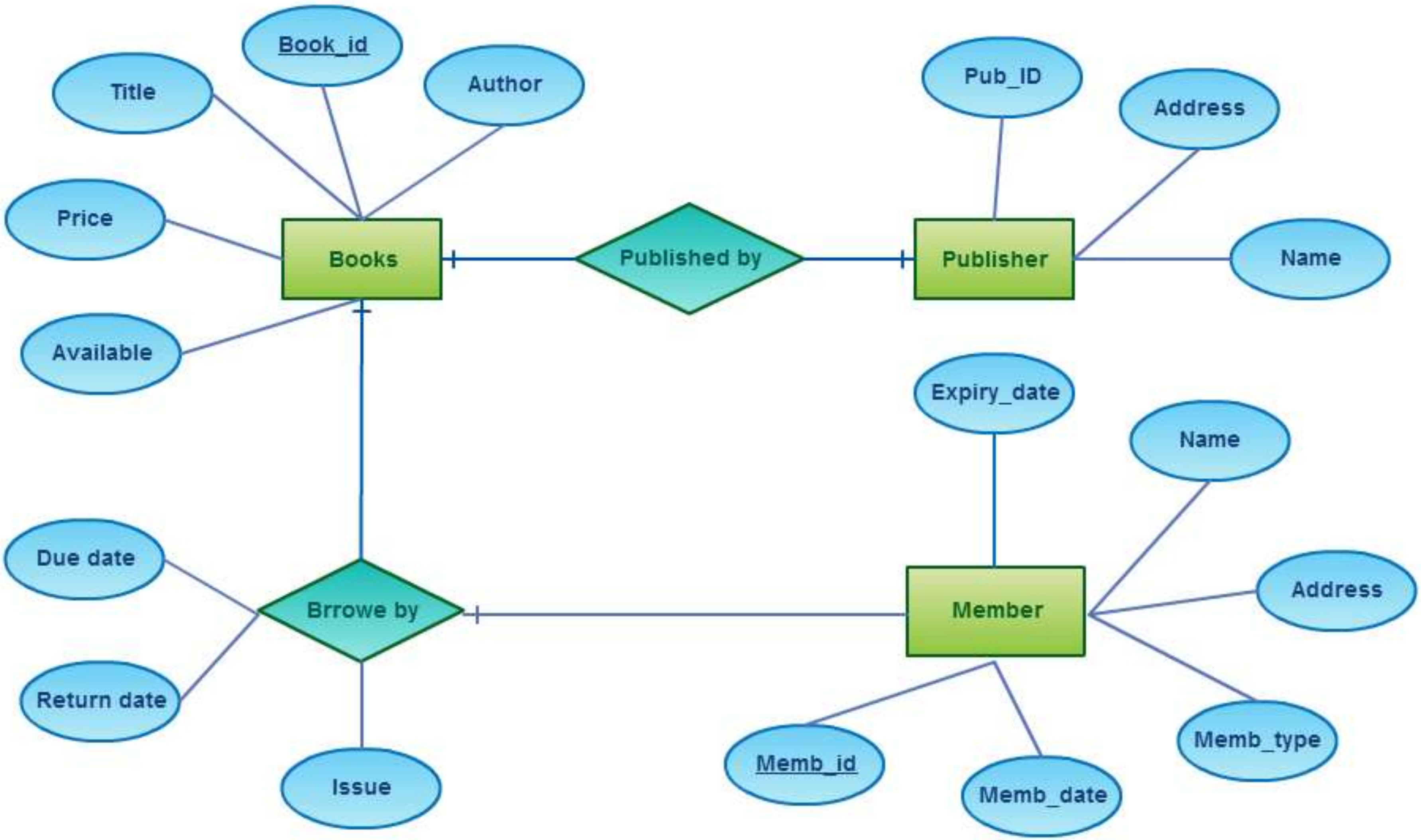
- module structure,
- control relationship among the modules
  - call relationship or invocation relationship
- interface among different modules,
  - data items exchanged among different modules,
- data structures of individual modules,
- algorithms for individual modules.

Module Structure



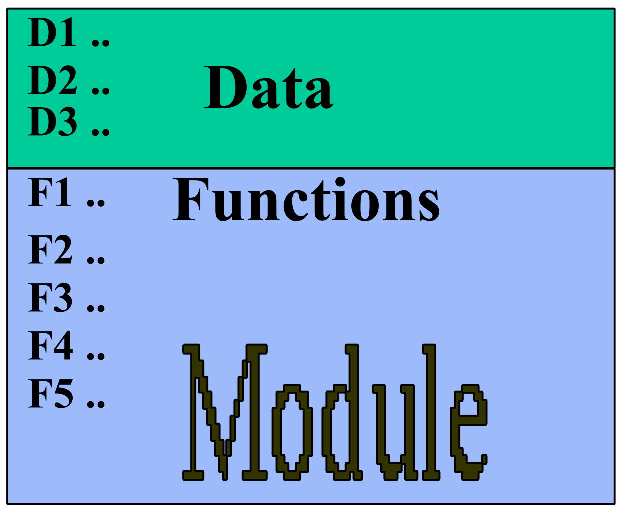

interaction Online Bookshop
:Inventory
1.1: search()
2.3 [order complete]:
update_inventory()
1.2 [interested]:
view_book()
1 *: find_books()
:Online
Bookshop
:Book
2: checkout()
1.3 [decided to buy]:
add_to_cart()
2.2 [not empty(cart)]:
make_order()
2.1: get_books()
:Shopping
Cart
:Order

E-R Diagram of Library Management System



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, Jackson diagram.

- 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 <u>cleanly decomposed</u> set of modules <u>(modularity),</u>
- Different modules should be neatly arranged in a hierarchy:
  - in a neat tree-like diagram.

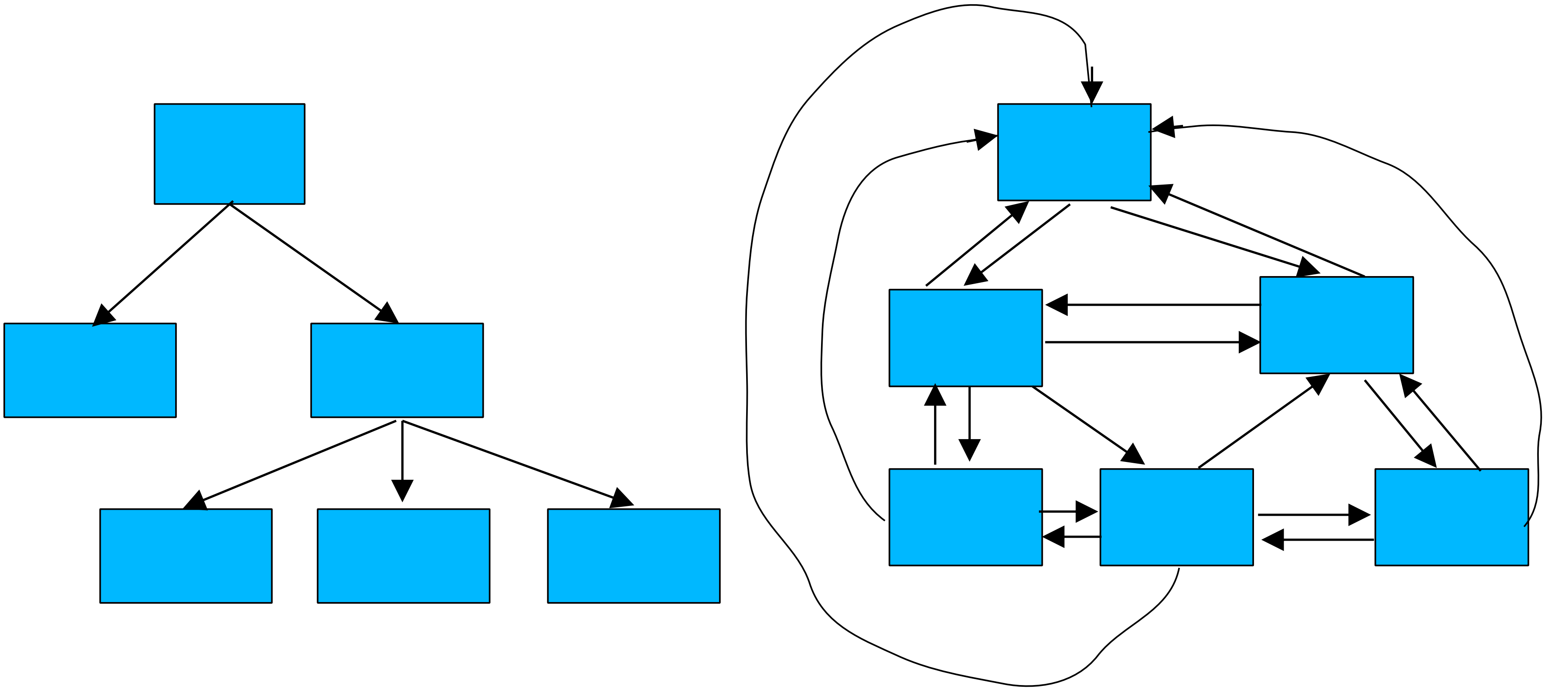
Modularity

- 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

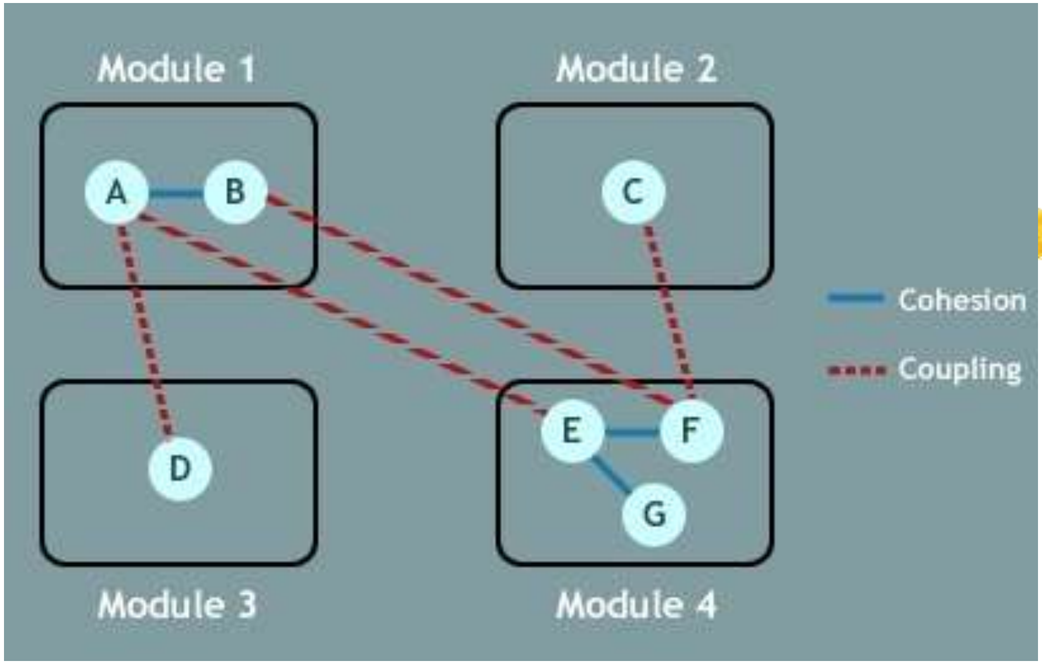
- In technical terms, modules should display:
  - high cohesion
  - low coupling.

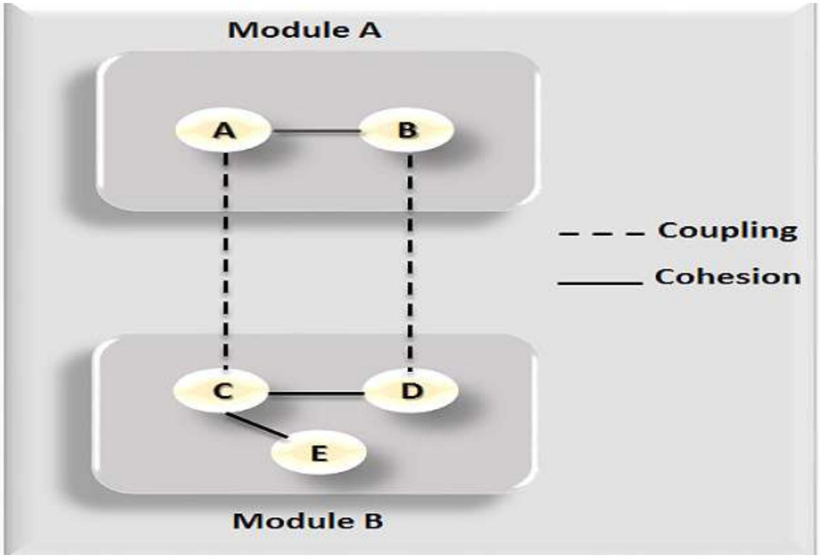
Modularity

- Neat arrangement of modules in a hierarchy means:
  - low fan-out
  - abstraction

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.

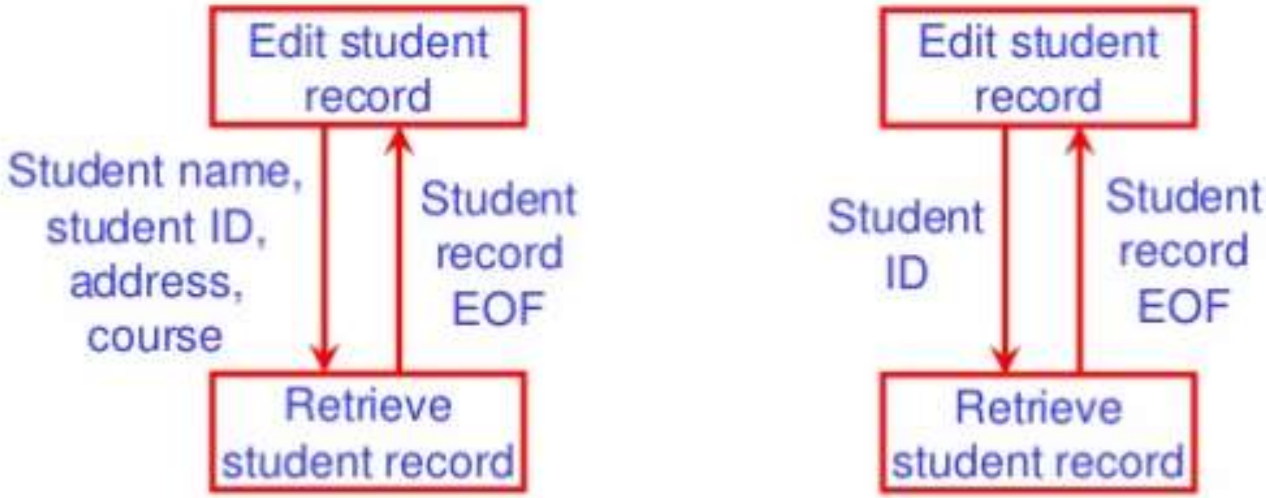




A, B, C are different data structures or functions calling each other

Software Design

Consider the example of editing a student record in a 'student information system'.



Poor design: Tight Coupling

Good design: Loose Coupling

Fig. 6 : Example of coupling

Cohesion and Coupling

- A module having high cohesion and low coupling:
  - <u>functionally independent</u> 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

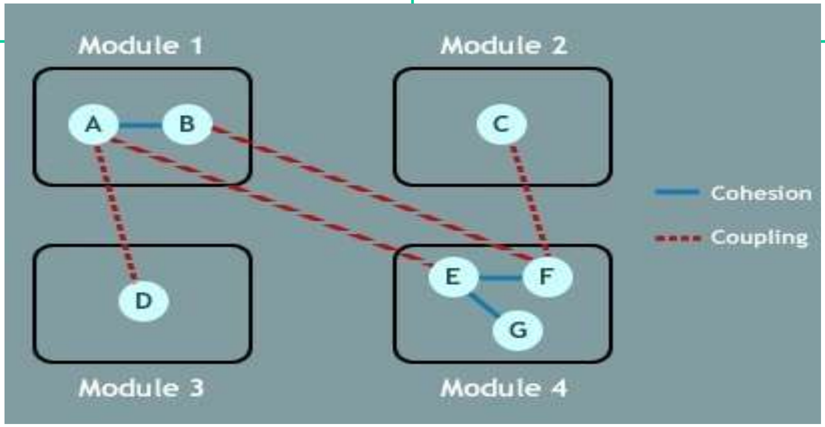
Advantages of Functional Independence

- 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.

Advantages of Functional Independence

- A functionally independent module:
  - can be easily taken out and reused in a different program.
    - each module does some well-defined and precise function
    - the interfaces of a module with other modules is simple and minimal.

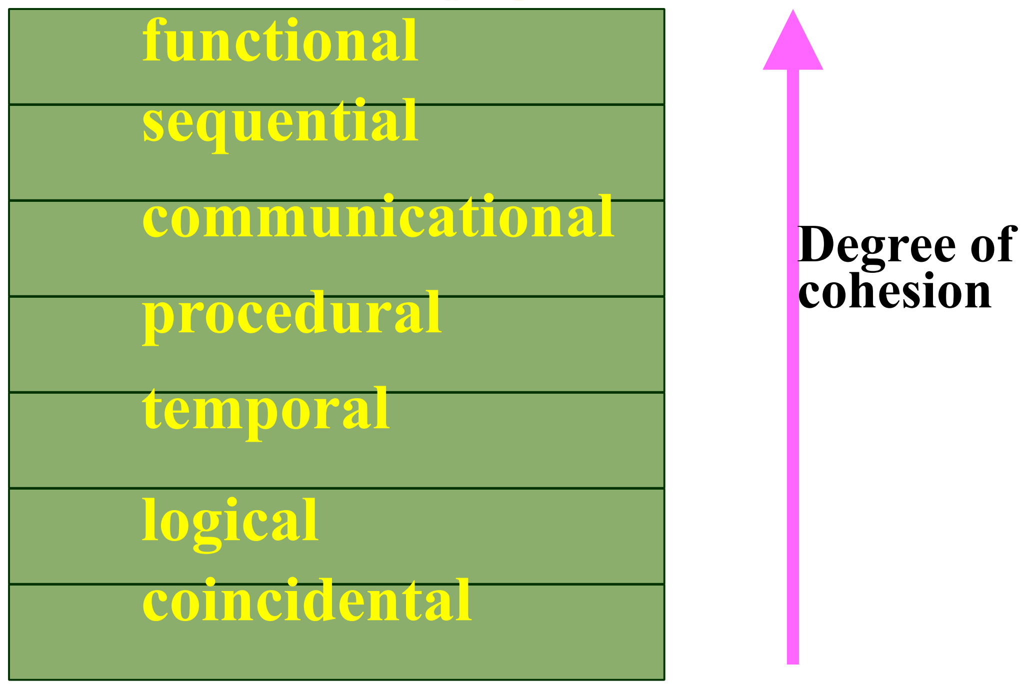
Cohesion	Coupling
Cohesion is the indication of the relationship **within** module.	Coupling is the indication of the relationships **between** modules.
Cohesion shows the module's **relative functional strength**.	Coupling shows the **relative independence among the modules**.
Cohesion is a degree (quality) 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.

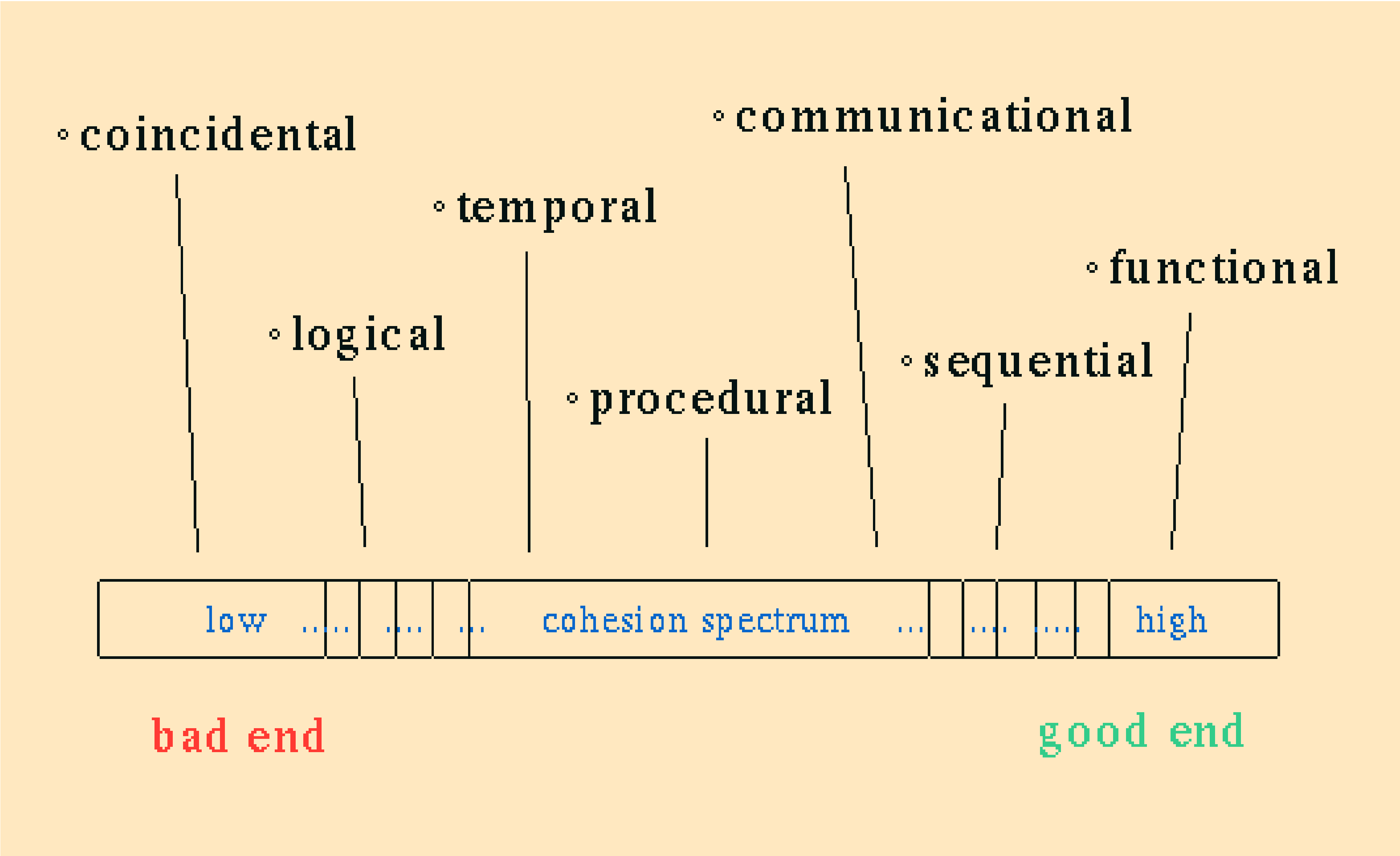


Classification of Cohesiveness

- 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.

Classification of Cohesiveness





◦ coincidental
◦ logical
◦ temporal
◦ procedural
◦ communicational
◦ sequential
◦ functional
low .... cohesion spectrum ... . high
bad end
good end

Coincidental cohesion

- The module performs a set of tasks:
  - which relate to each other very loosely, if at all.
    - the module contains a random collection of functions.
    - functions have been put in the module out of pure coincidence without any thought or design.
    - For example, in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module.

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.

Temporal cohesion

- The module contains tasks that are related by the fact:
  - 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.

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.

Communicational cohesion

ÑAll functions of the module:

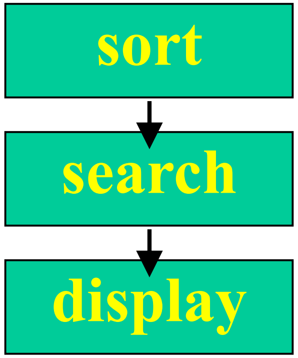
- yreference or update the same data structure,

ÑExample:

- ythe set of functions defined on an array or a stack.

Sequential cohesion

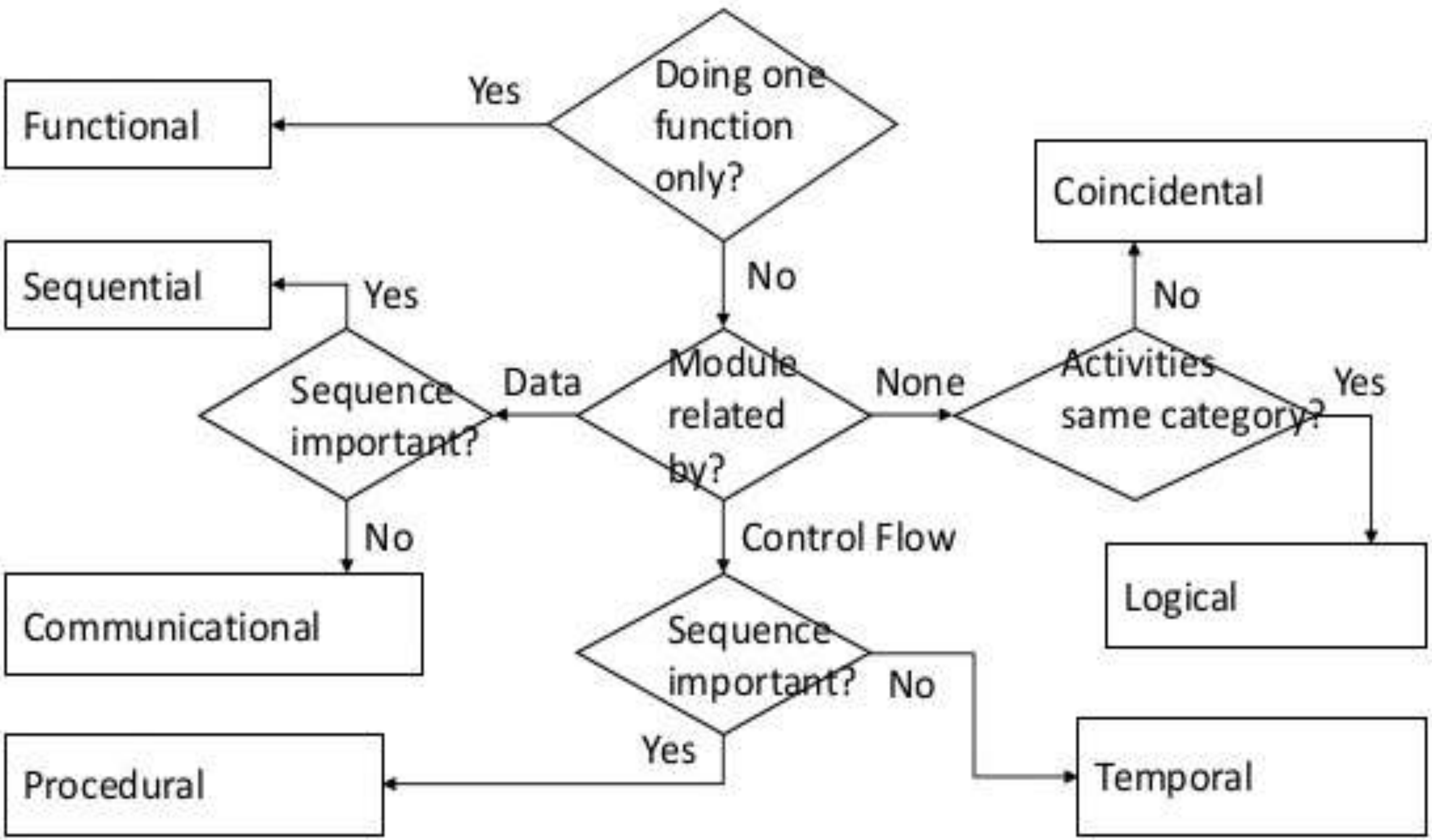
- Elements of a module form different parts of a sequence,
  - output from one element of the sequence is input to the next.
  - Example:



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.

Determining Module Cohesion



Determining Cohesiveness

- Write down a sentence to describe the function of the module
  - If the sentence is compound,
    - 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,
    - it probably has temporal cohesion.

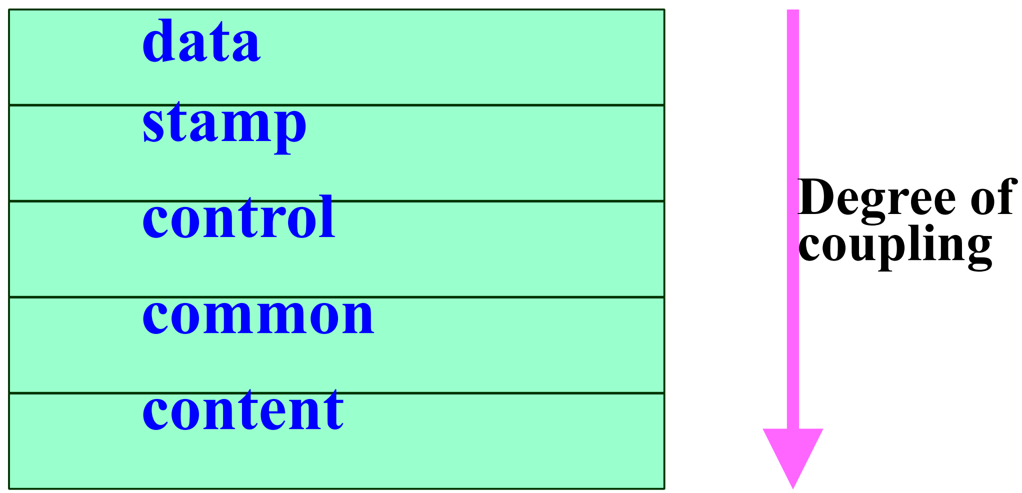
Coupling

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

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.

Classes of coupling

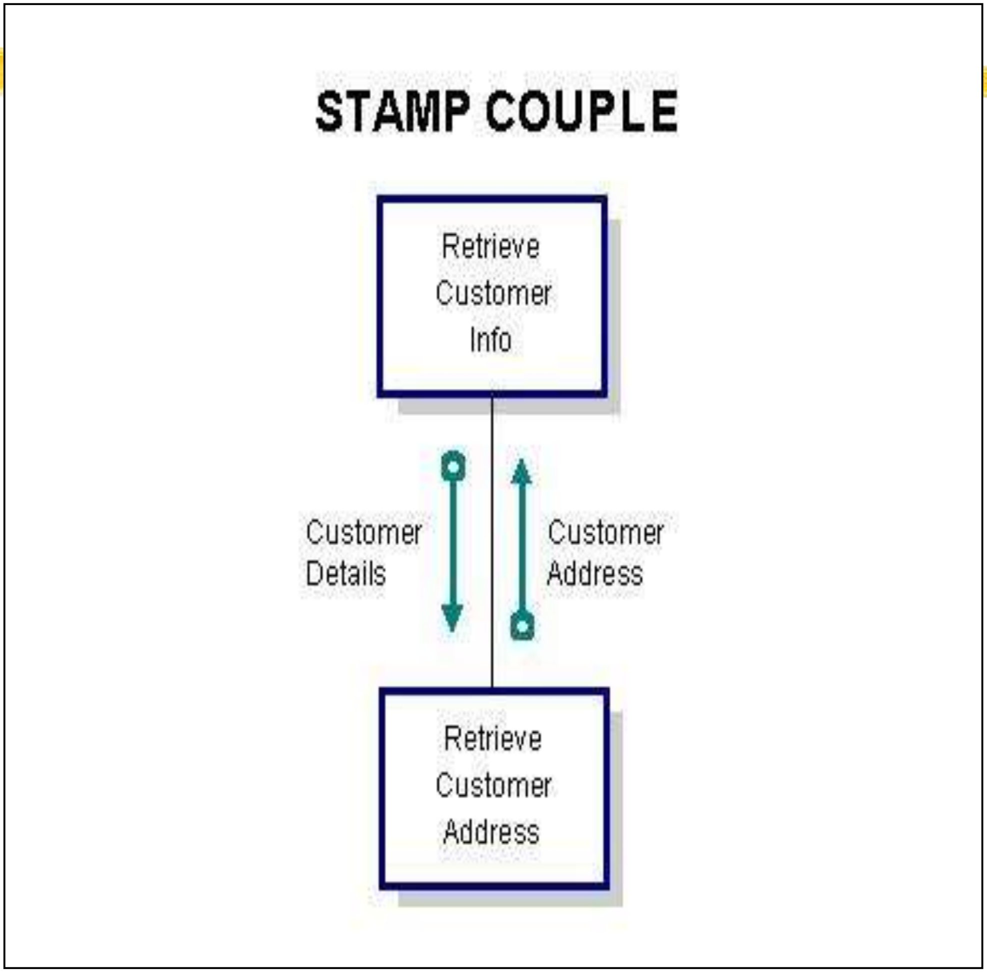


Data coupling

- Two modules are data coupled,
  - if they communicate via a parameter:
    - an elementary data item,
    - e.g an integer, a float, a character, etc.
  - The data item should be problem related:
    - not used for control purpose.

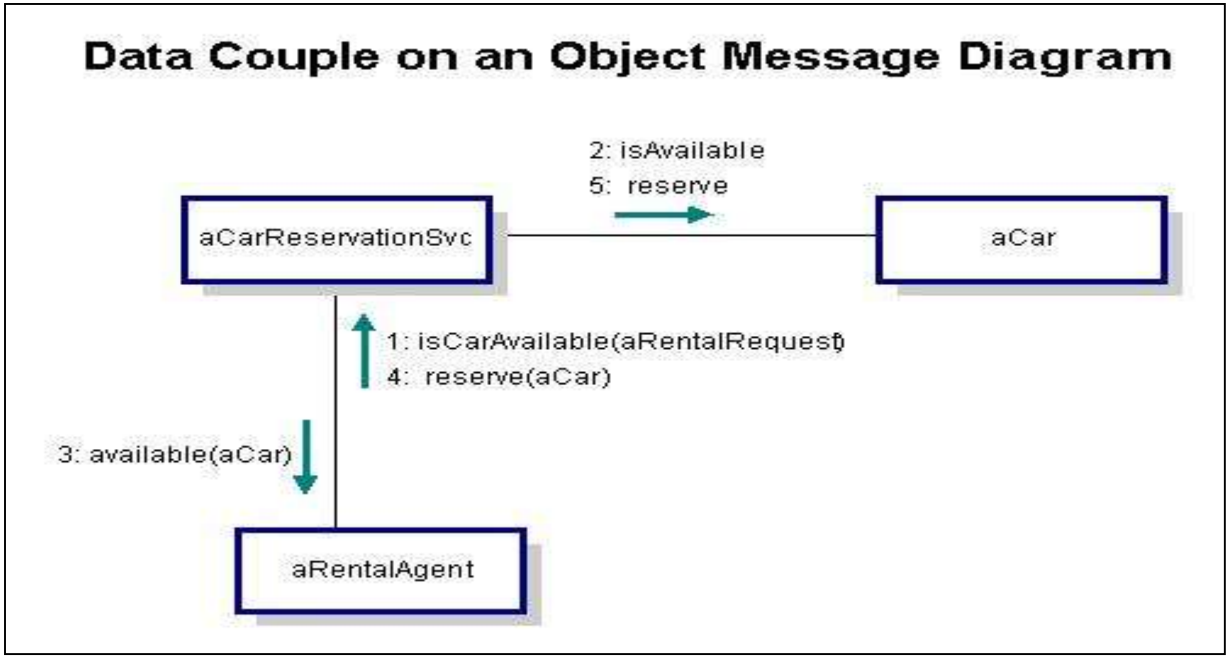
Stamp coupling

- Two modules are <u>stamp coupled</u>,
  - if they communicate via a composite data item
    - such as a record in PASCAL
    - or a structure in C.



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.

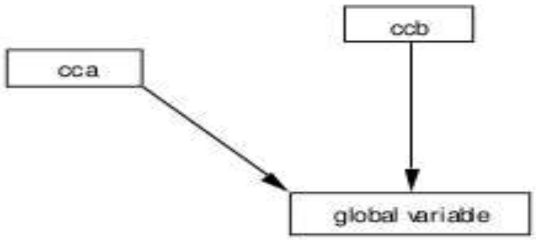


Common Coupling

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

Example of Common Coupling

```
while (global variable == 0)
   if (argument xyz > 25)
      module 3 ();
   else
      module 4 ();
```

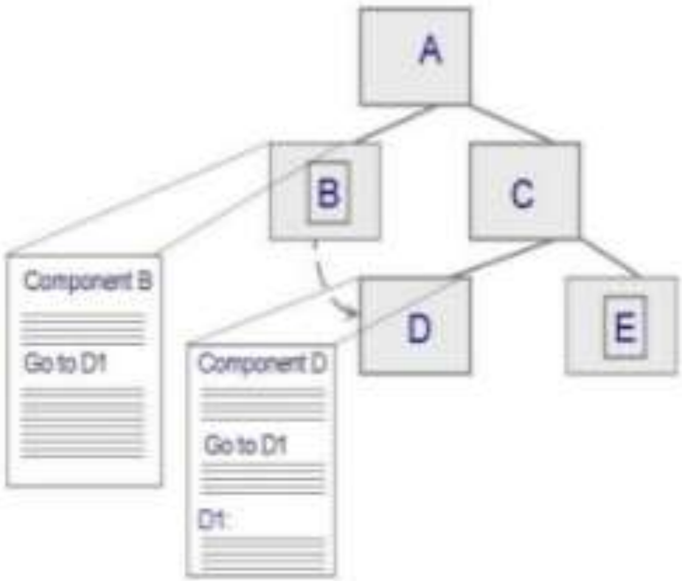


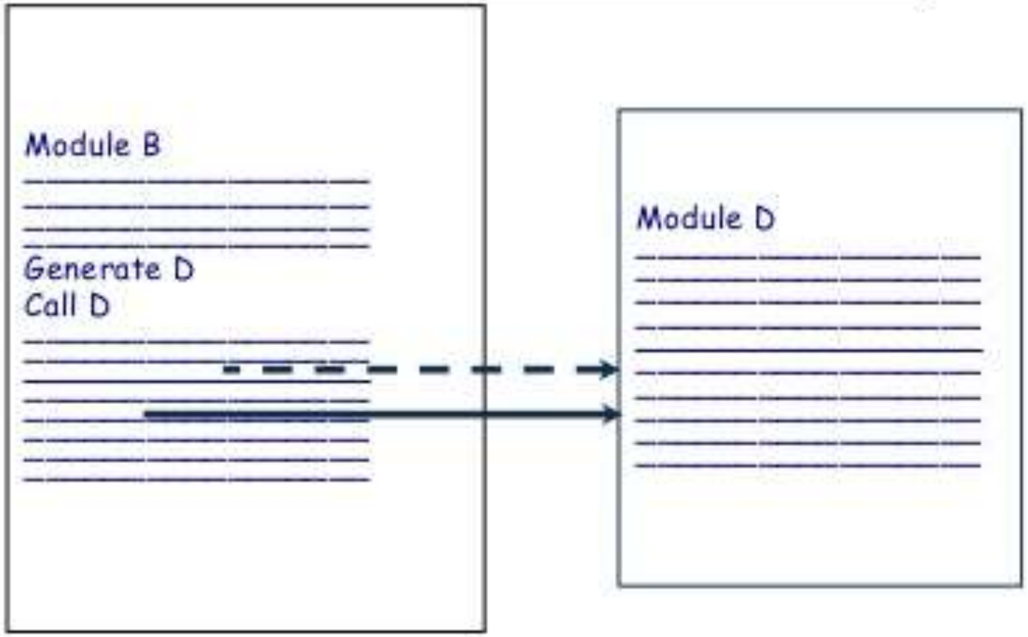
Content coupling

- 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.

Example of Content Coupling

- Occurs when one component modifies an internal data item in another component, or when one component branches into the middle of another component





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.

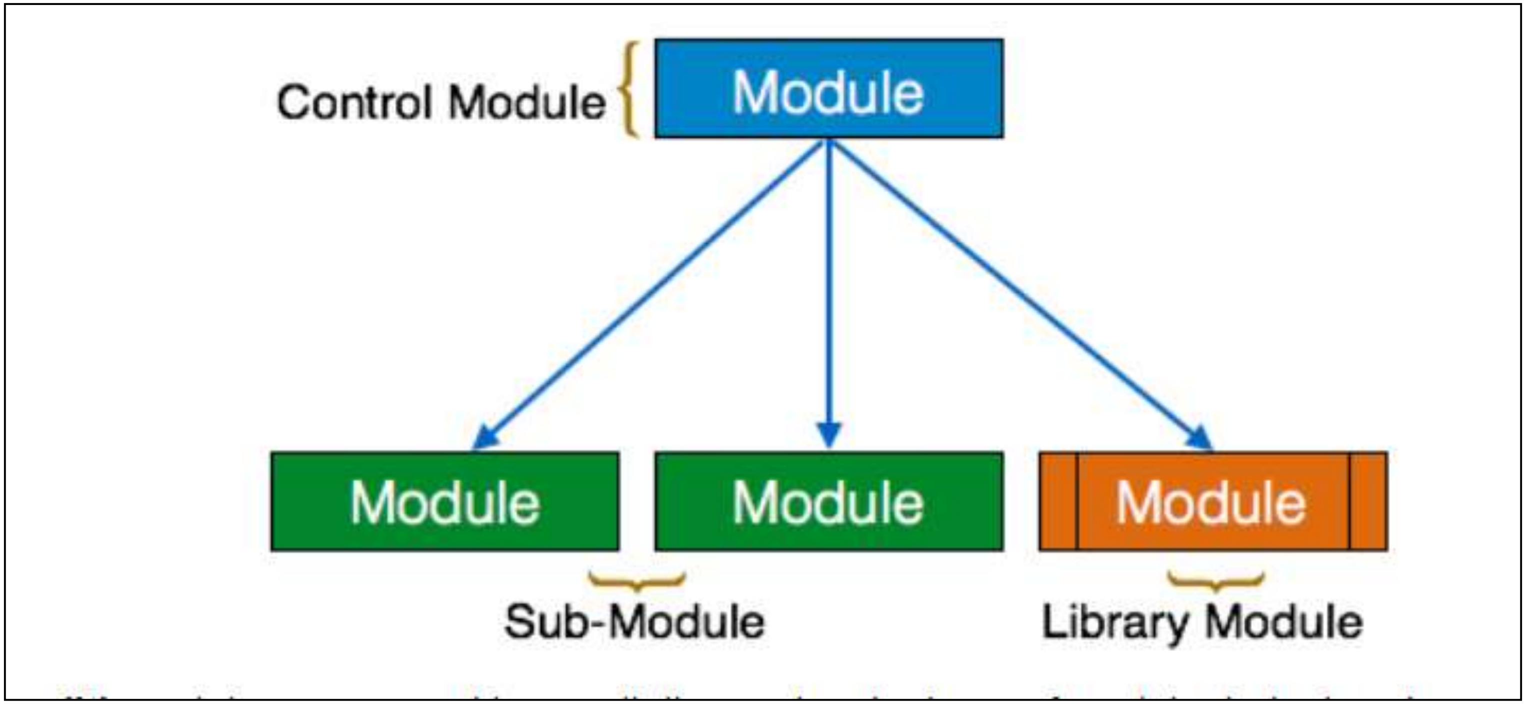
Structure Chart

A Structure Chart in software engineering and organizational theory is a chart which shows the breakdown of a system to its lowest manageable levels.

They are used in structured programming to arrange program modules into a tree.

Each module is represented by a box, which contains the module's name.

Structure chart



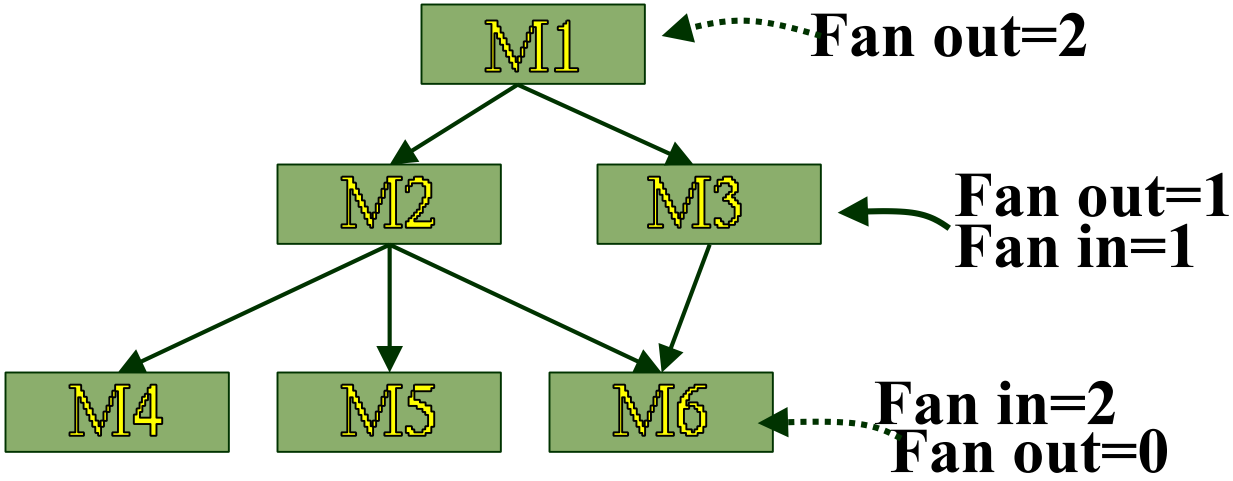
Neat Arrangement of modules

- Essentially means:
  - low fan-out
  - abstraction

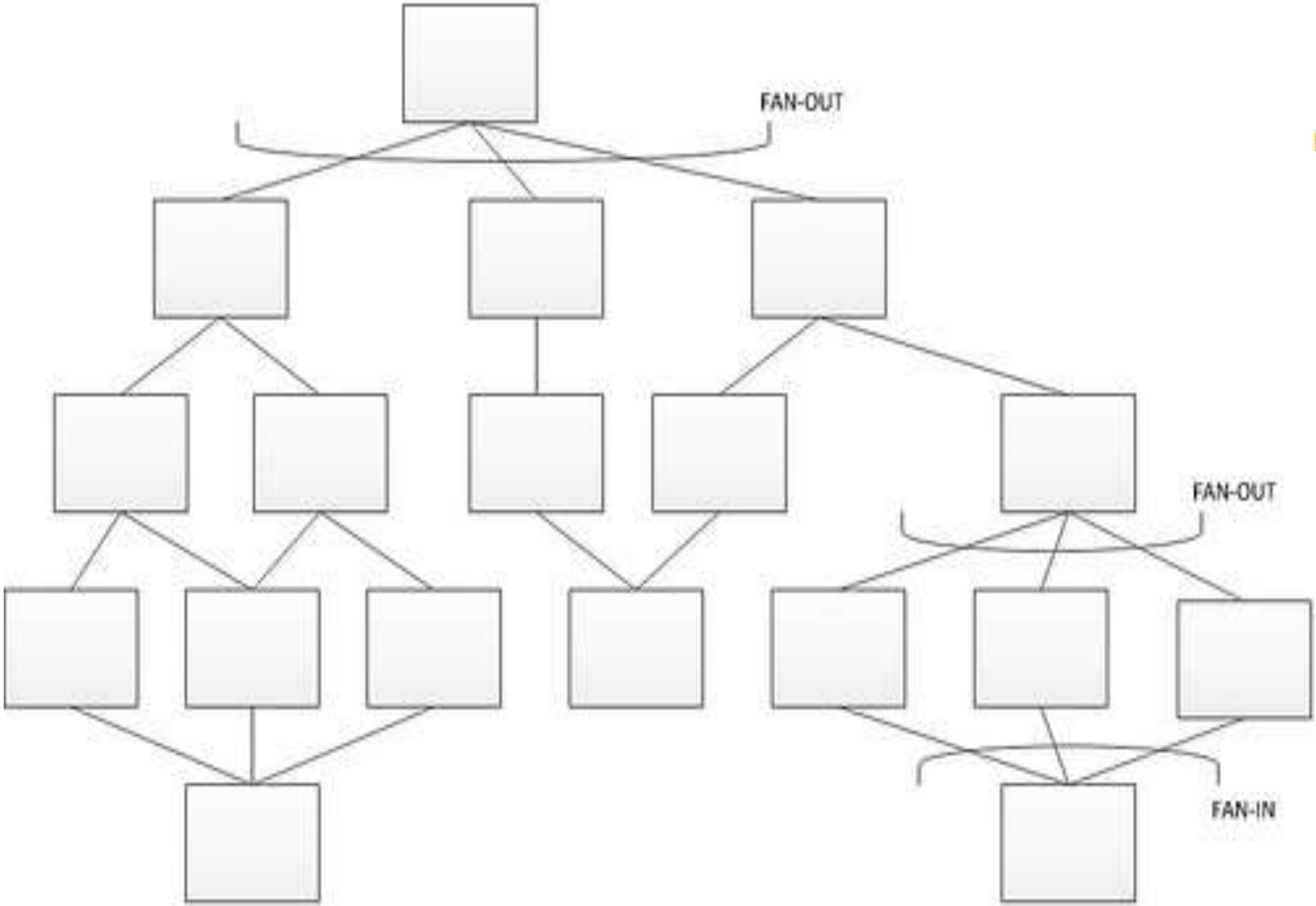
Characteristics of Module Structure

- 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



Fan-Out/Fan-In



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.

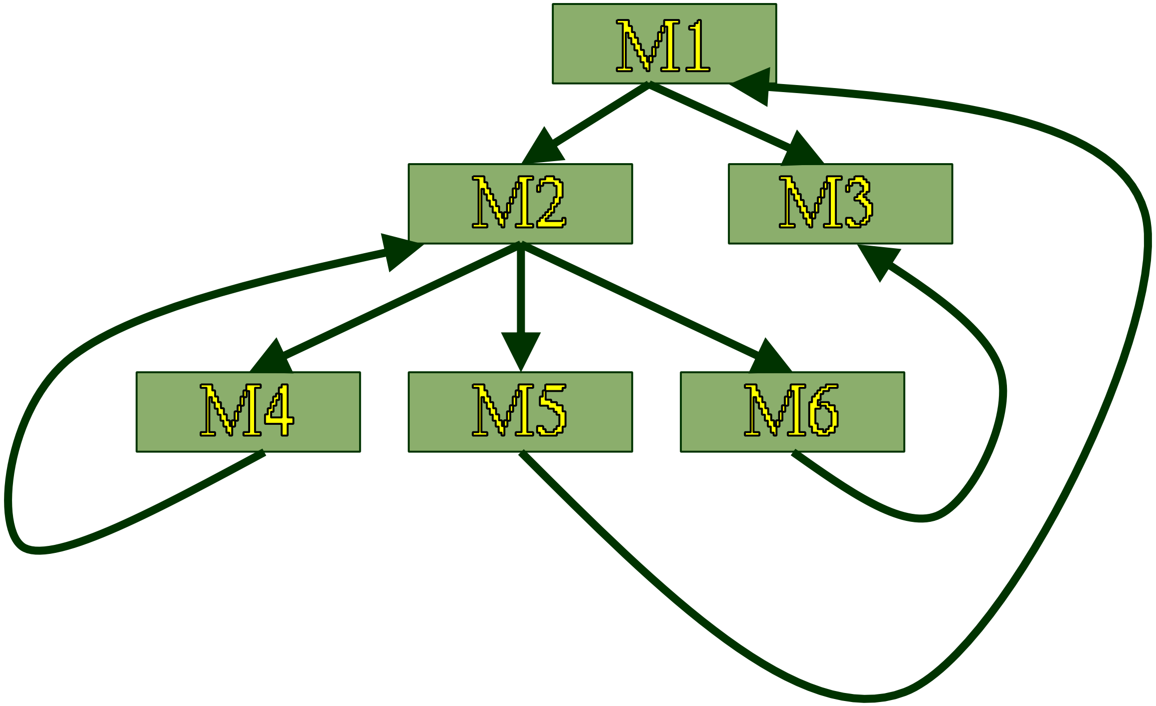
Control Relationships

- 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.

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



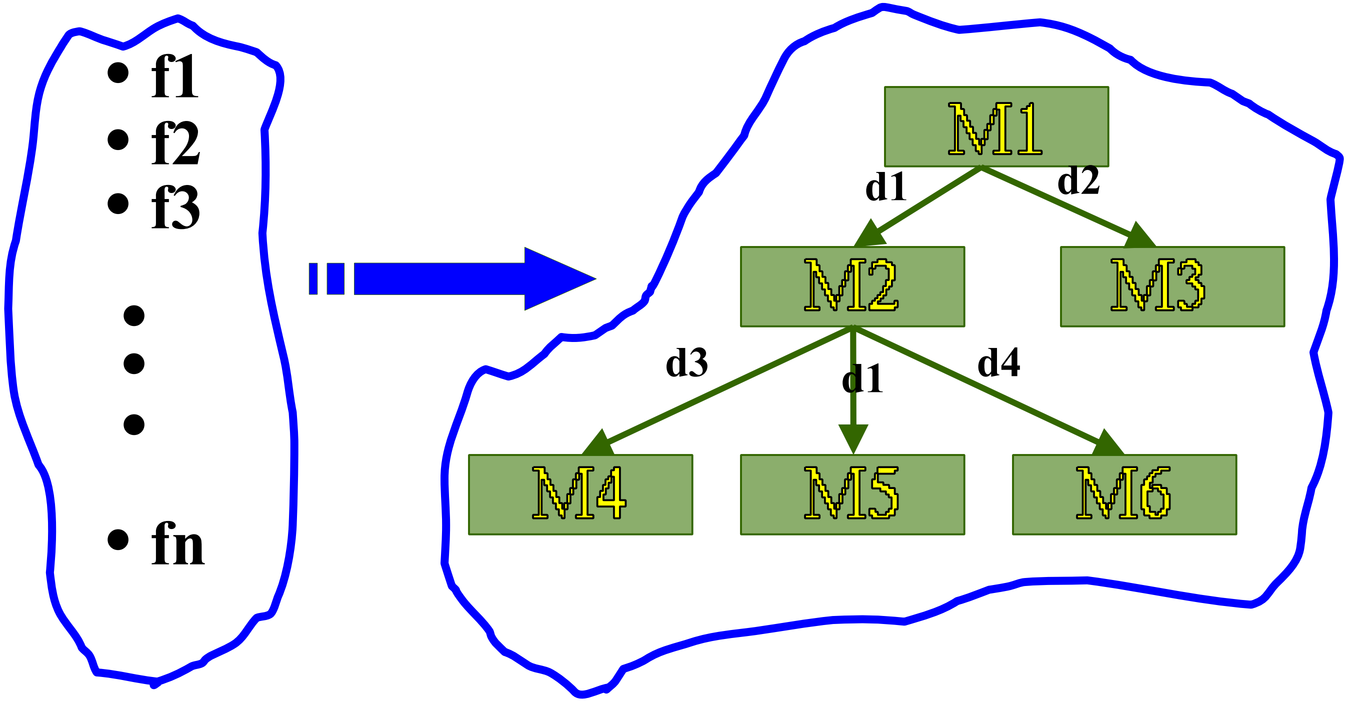
Abstraction

- 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



Design Approaches

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

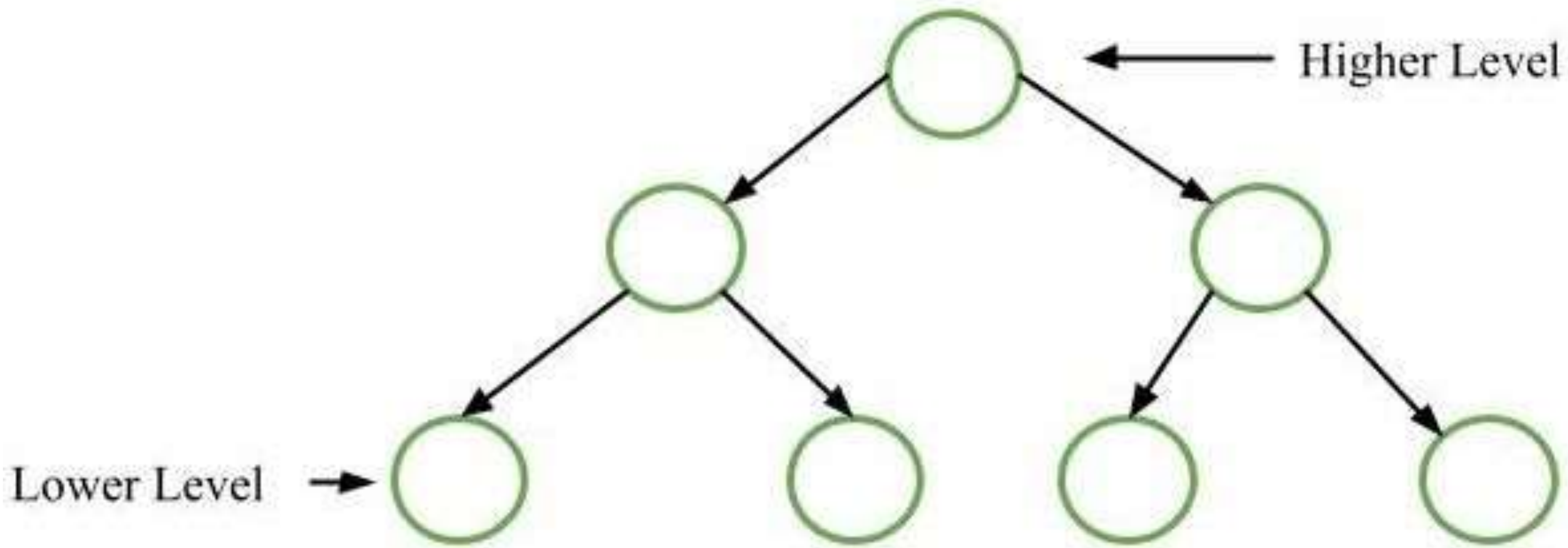
- These two design approaches are radically different.
  - However, are complementary
    - rather than competing techniques.
  - Each technique is applicable at
    - different stages of the design process.

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.

Function-Oriented Design



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
      - Renew-member-record

FUNCTION DESIGN

Main function

- **New member**
  - -Assign membershop number
  - -Print bills
  - -Create record
- **Renewal**
  - Update record
  - Get Details
  - Print bills
- **Delete Member**
  - - Get Details
  - - Print Cheque
  - - Delete record
- **Invalid option**
  - - Print error messag

Function-Oriented Design

- Several function-oriented design approaches have been developed:
  - Structured design (Constantine and Yourdon, 1979)
  - Jackson's structured design (Jackson, 1975)
  - Warnier-Orr methodology
  - Wirth's step-wise refinement
  - Hatley and Pirbhai's Methodology

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.

What is an Object?

- An object Is an unique, identifiable, self-contained entity that possesses operations and contains attributes
- Possesses all the know-how and information it needs to perform the services for which it was designed
- Is a **"black box"** which receives and sends messages

What is a Class ?

- A Class is a software template that defines the methods and variables to be included in a particular kind of Object.
- Is a blue print used to create objects. As it is a **blue print**, at runtime it will not occupy any memory.
- Examples :

  Animal, Human being, Automobiles

What can be objects ?

- Noun phase
- People
- Organization
- events
- places
- concepts

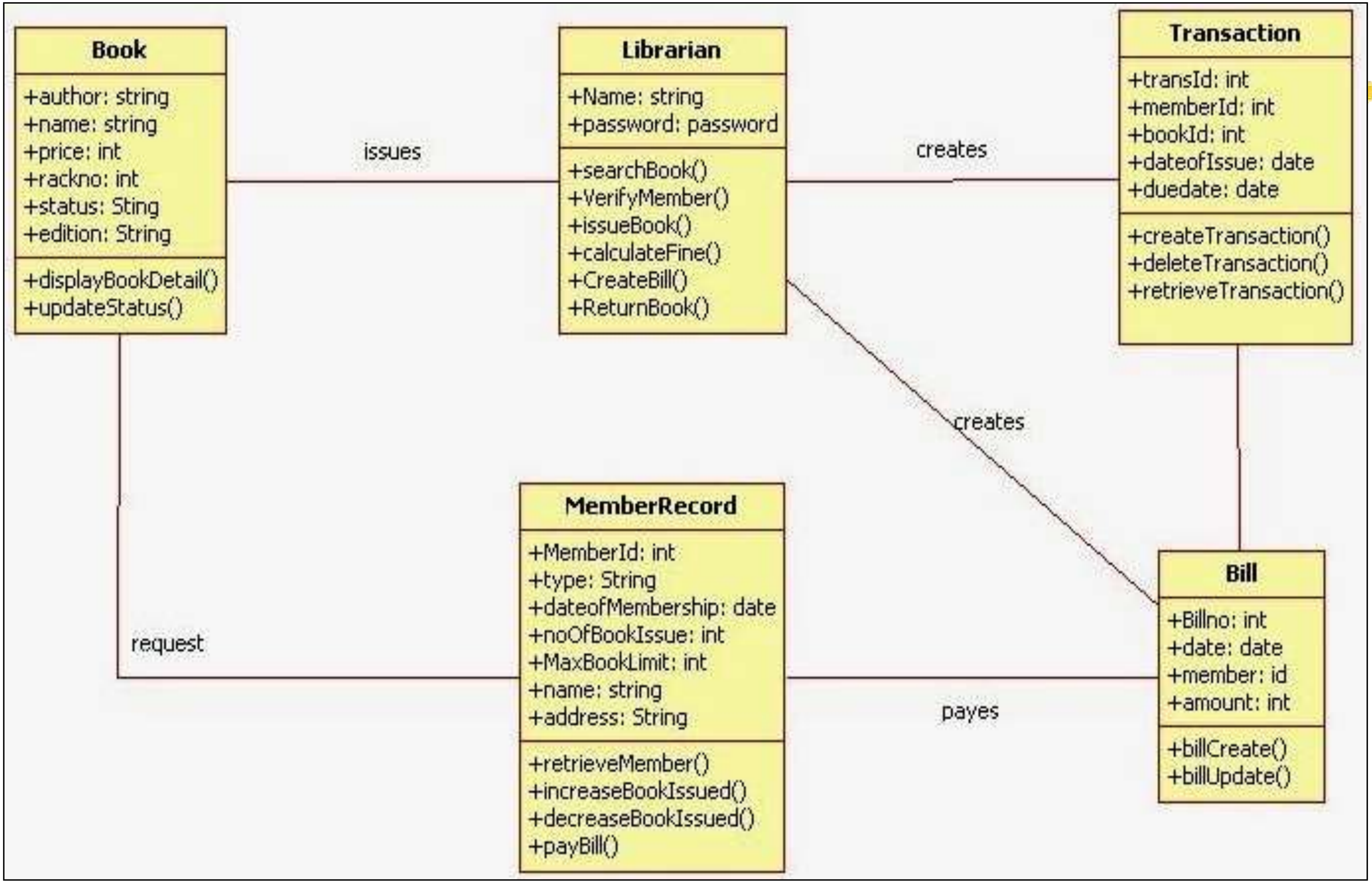
Object-Oriented Design Example

- 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

- Objects have their own internal data:
  - defines their state.
- Similar objects constitute a class.
  - each object is a member of some class.
- Classes may inherit features
  - from a super class.
- Conceptually, objects communicate by message passing.

Object Oriened Design



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.

Object-Oriented versus Function-Oriented Design

- Grady Booch sums up this fundamental difference saying:
  - "Identify verbs if you are after procedural design and nouns if you are after object-oriented design."

- In OOD:
  - state information is not shared in a centralized data.
  - but is distributed among the objects of the system.

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

- The fire alarm system should:
  - flash an alarm message on the computer console:
    - fire fighting personnel man the console round the clock.
- After a fire condition has been successfully handled,
  - 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();
```

Function-Oriented Approach:

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

BOOL detector_status[MAX_ROOMS];
int detector_locs[MAX_ROOMS];
BOOL alarm_status[MAX_ROOMS];
/* alarm activated when status is set */
int alarm_locs[MAX_ROOMS];
/* room number where alarm is located */
int neighbor-alarm[MAX_ROOMS][10];
/* each detector has atmost 10 neighboring locations */

The functions which operate on the system state are:

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.

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
```

Summary

- We started with an overview of:
  - activities undertaken during the software design phase.
- We identified:
  - the information need to be produced at the end of the design phase:
    - so that the design can be easily implemented using a programming language.

Summary

- We characterized the features of a good software design by introducing the concepts of:
  - fan-in, fan-out,
  - cohesion, coupling,
  - abstraction, etc.

Summary

- We classified different types of cohesion and coupling:
  - enables us to approximately determine the cohesion and coupling existing in a design.

Summary

- Two fundamentally different approaches to software design:
  - function-oriented approach
  - object-oriented approach

Summary

- We looked at the essential philosophy behind these two approaches
  - these two approaches are not competing but complementary approaches.